آنتونی سنتیس
عباس ریاضی - مهدی بنواری

ویراست دوم

آموزش

برنامه‌نویسی شیءگرا

در ۲۱ روز

با

آموزش

برنامه‌نویسی شیءگرا

در ۲۱ روز

آنتونی سنتیس

مهندس عباس ریاضی
مهندس مهدی بنواری

موسسه علمی فرهنگی

سنتیس، آنتونی
آموزش برنامه نویسی شیءگرا در ۲۱ روز/سینتس [ترجمه]عباس ریاضی، مهدی بنواری. تهران: نص، ۱۳۸۳.
۴۷۲ ص. مصور.

ISBN: 964-410-006-9 با CD ۴۰۰۰۰ ریال :

فهرستنویسی بر اساس فیپا.
عنوان اصلی:
Teach YourSelf Object Oriented Programming in 21 days. 2nd ed
۱. برنامه نویسی شیءگرا. الف ریاضی، عباس، ۱۳۵۷، مترجم ب. بنواری، مهدی، ۱۳۵۶، مترجم. ج. عنوان.
QA۷۶/۶۴/۴۳/ ۹آ۸س
۰۰۵/۱۱۷
۱۳۸۳
کتابخانه ملی ایران
م۸۱-۳۶۱۹۱

موسسه علمی فرهنگی

آموزش برنامه‌نویسی شیءگرا در ۲۱ روز

آنتونی سنتیس
عباس ریاضی /مهدی بنواری
چاپ اول : پاییز ۸۳
چاپ و صحافی: سازمان چاپ و انتشارات وزارت فرهنگ و ارشاد اسلامی
طراحی، آماده سازی: موسسه علمی فرهنگی «نص»
قیمت با CD: ۴۰۰۰ تومان

تهران: میدان انقلاب، خیابان اردیبهشت، بن بست مبین، شماره ۲۳۷
تلفکس: ۶۴۱۲۳۸۵- تلفن: ۴-۶۹۵۳۸۸۳ / ص.پ. ۱۳۱۴۵/۸۶۳

شابک: ۹-۰۰۶-۴۱۰-۹۶۴
ISBN:694-410-006-9

تقدیم به:
شریک زندگیم، سرکار خانم مهندس شفیعی و
فرزندمان محمد مهدی
عباس ریاضی

تقدیم به:
همراه و همسر عزیزم، به خاطر تمام یاریها و
دلگرمی دادنهایش
مهدی بنواری

مقدمه

در نظر داریم در این کتاب از طریقی عملی به آموزش برنامه‌نویسی شیءگرا (Object Oriented Programming) بپردازیم. اما به جای استفاده از روشهای آکادمیک، راهبردی قابل دسترس، آسان و کاربرپسند با استفاده از مثالهای فراوان در پیش خواهیم گرفت، در این صورت به جای درگیر شدن در مفاهیم پیچیده و بررسی یک به یک مفاهیم، ویژگیها و استانداردهای روش شیءگرا فقط به مباحثی خواهیم پرداخت که در برنامه‌نویسی روزمره به آنها نیاز پیدا می‌کنیم.

هدف این کتاب بنا نهادن پایه‌ای محکم برای برنامه نویسی شیءگرا در خواننده است. بعد از خواندن این کتاب باید چنان درک خوبی از مفاهیم بنیادین برنامه نویسی شیءگرا پیدا کرده باشید که بتوانید در کارهای روزمرهٔ خود از روش شیءگرا استفاده کنید و نیز بتوانید با مطالعهٔ مراجع و منابع تخصصی‌تر دانش و مهارت برنامه نویسی شیءگرای خود را افزایش دهید. مسلماً در ۲۱ روز نمی‌توان OOP را به طور کامل آموخت و درک کرد، اما می‌توان پایهٔ لازم برای ادامهٔ مطالعه در این زمینه را به دست آورد.

من این کتاب را به سه بخش تقسیم کرده‌ام. در هفتهٔ اول به معرفی و بررسی سه رکن اساسی برنامه نویسی شیءگرا خواهم پرداخت. درک مفهوم این سه رکن نقشی کلیدی در آموختن روش شیءگرا دارد. در درسهای هفتهٔ اول ابتدا به صورت تئوری و سپس به صورت عملی از طریق کارگاه‌های ارایه شده، به بررسی هر یک از سه رکن مزبور می‌پردازم.

هفتهٔ دوم به فرایند توسعهٔ نرم‌افزار اختصاص دارد. اگرچه مفاهیم معرفی شده در هفتهٔ اول بسیار مهم هستند، اما برای شروع برنامه‌نویسی، دانستن آنها کفایت نمی‌کند. در واقع در این مرحله این است که به فردی غیر ماهر مقادیری مصالح و ابزار کار داده شود و از او خواسته شود تا خانه بسازد. در درسهای هفتهٔ دوم به آموزش چگونگی استفاده از ابزارهای معرفی شده در هفتهٔ اول می‌پردازم.
در طی هفتهٔ سوم با هم یک مورد کامل ایجاد یک بازی ورق با استفده از روش شیءگرا را انجام خواهیم داد. در طی این پروژه، یک چرخهٔ کامل توسعهٔ نرم‌افزار شیءگرا را از ابتدا تا انتها با هم طی خواهیم کرد. در این صورت مجبور خواهید شد خود به کدنویسی بپردازید. امیدوارم این تمرین باعث جا افتادن تئوری برنامه‌نویسی شیءگرا در ذهن خواننده گردد.

دربارهٔ مثالها

کد منبع تمام مثالهای این کتاب به زبان Java است. اگر چه آشنایی با Java در درک مفهوم کدها مؤثر است اما اهمیت حیاتی ندارد. در صورتی که لازم دیدید به یک کتاب مقدماتی در مورد Java رجوع کنید. برای واقعی و ملموس بودن مثالها تلاش فراوانی صرف شده و سعی شده از استفاده و تاکید بر شیوه‌های خاص و ترفندهای مختص Java خودداری شود.

پیش‌نیازهای این کتاب

فرض بر این است که خوانندهٔ این کتاب تجربیات چندی در برنامه‌نویسی دارد. هدف این کتاب هم آموزش برنامه‌نویسی مقدماتی نیست. این کتاب از دانش خواننده در زمینهٔ برنامه‌نویسی بهره می‌گیرد تا نشان دهد چگونه می‌توان با استفاده از همین دانش نرم‌افزار شیءگرا نوشت. البته منظور این نیست که برای درک مفاهیم این کتاب باید استاد برنامه‌نویسی باشید، بلکه گذراندن یک دورهٔ مقدماتی یا خواندن یک کتاب خودآموز هم کفایت می‌کند.
برای اینکه بتوانید از مثالها و تمرین‌ها استفادهٔ کافی بکنید به یک کامپیوتر و یک اتصال اینترنت هم نیاز دارید. نوع سیستم عامل و ویرایشگر کاملاً به سلیقهٔ خواننده بستگی دارد و اهمیت خاصی در درک مفاهیم ندارد. تنها نیازمندی خاص نصب Java روی کامپیوتر است.
در پایان باید به خواننده یادآوری کنم که در راهی که در پیش دارد به قوهٔ تشخیص و ذهن باز نیازمند خواهد بود. برنامه‌نویسی شیءگرا آسان نیست و تسلط در آن بسیار بیشتر از ۲۱ روز زمان نیاز دارد، اما این کتاب شروع خوبی خواهد بود.
دنیای شگفت‌انگیز برنامه‌نویسی شیءگرا در انتظار شماست .

فهرست مطالب

روز ۱

مقدمه‌ای بر برنامه‌نویسی شیءگرا..... ۱۵

برنامه‌نویسی شیءگرا در گذر زمان ۱۶
مقدمه‌ای بر OOP ۱۶
برنامه‌نویسی شیءگرا..... ۱۸
مزایا و اهداف برنامه‌نویسی شیءگرا..... ۲۶
سادگی..... ۲۶
پایداری..... ۲۷
قابل استفاده مجدد..... ۲۷
قابلیت نگهداری ۲۷
قابلیت توسعه..... ۲۸
به موقع بودن ۲۸
دام‌ها ۲۸
دام اول: این تفکر که OOP زبانی ساده است.. ۲۸
دام دوم: سفسطه و مغلطه..... ۲۹
دام سوم: این تفکر که شیءگرایی همیشه شفابخش است..... ۲۹
دام چهارم: برنامه‌نویسی همراه با مخفی‌کاری ۲۹
هفته‌ای که در پیش رو دارید ۲۹
خلاصه ۳۰
پرسش ها و پاسخها ۳۰
کارگاه ۳۱
پرسشها..... ۳۱
تمرین‌ها ۳۱

روز ۲

کپسوله‌سازی: بیاموزید جزییات رانزد خود نگاه دارید..... ۳۳

سه رکن بنیادی برنامه‌نویسی شیءگرا..... ۳۴
کپسوله‌سازی: رکن اول ۳۴
مثالی از رابط و پیاده‌سازی ۳۶
سطح‌های عمومی، اختصاصی و محافظت شده..... ۳۷
چرا کپسوله‌سازی؟ ۳۷
تجرید: چگونه عام فکر کنیم و برنامه بنویسیم ... ۳۸
تجرید چیست؟..... ۳۸
دو نمونهٔ تجرید..... ۳۹
تجرید مؤثر ۴۰

حفظ اسرار با مخفی کردن پیاده‌سازی ۴۱
محافظت از شیء توسط ADTها ۴۱
محافظت از کاربران از طریق اختفای کد ۴۵
یک مثال اختفاء از دنیای واقعی ۴۶
تقسیم مسئولیت: فقط به کار خودتان برسید! ۴۸
نکته‌ها و دامهای کپسوله‌سازی ۵۱
نکته و دامهای تعمیم ۵۱
نکته‌ها و دامهای ADT ۵۳
نکاتی در مورد مخفی کردن پیاده‌سازی ۵۳
چگونه کپسوله‌سازی اهداف
برنامه‌نویسی شیءگرا را تأمین می‌کند ۵۳
هشدار ۵۴
خلاصه ۵۵
پرسش‌ها و پاسخ‌ها ۵۵
کارگاه ۵۶
پرسشها ۵۶
تمرین‌ها ۵۶

روز ۳

کپسوله‌سازی: زمان نوشتن کد.... ۵۷

کارگاه ۱: برپایی محیط جاوا ۵۷
تعریف مسأله ۵۸
کارگاه ۲: اصول و مبانی کلاسها ۵۸
تعریف مسأله ۶۰
حل و بحث ۶۱
کارگاه ۳ـافزایش کپسوله‌سازی ۶۳
تعریف مسأله ۶۴
حل و بحث ۶۴
کارگاه ۴: مطالعه موردی ـکلاسهای
ابتدایی Java (اختیاری) ۶۹
تعریف مسأله ۷۳
حل و بحث ۷۳
پرسشها و پاسخها ۷۵
کارگاه ۷۵
پرسشها ۷۵
تمرین‌ها ۷۶

روز ۴

وراثت: ساختن از هیچ..... ۷۷

وراثت چیست؟ ۷۷
چرا وراثت؟ ۸۰
گشت و گذار در دنیای پیچیده وراثت ۸۲
مکانیزمهای وراثت ۸۴
انواع وراثت ۸۸
وراثت در پیاده‌سازی ۸۹
وراثت برای ایجاد تغییر ۹۰
وراثت برای جانشینی نوع داده ۹۲
نکاتی در مورد وراثت صحیح ۹۵
خلاصه ۹۶
پرسش و پاسخ ۹۸
کارگاه ۹۹
پرسشها ۹۹
تمرین‌ها ۹۹

روز ۵

وراثت: زمان نوشتن کد..... ۱۰۱

کارگاه ۱: وراثت ساده ۱۰۱
تعریف مسأله ۱۰۲
حل و بحث ۱۰۳
کارگاه ۲: استفاده از کلاسهای مجرد برای
وراثت طرح‌ریزی شده ۱۰۴
تعریف مسأله ۱۰۶

حل و بحث ۱۰۷
کارگاه ۳: حساب بانکی ـ تمرین وراثت ساده ... ۱۰۸
حساب کلی (عمومی) ۱۰۸
حساب پس‌انداز ۱۰۸
حساب سپرده‌گذاری زمانی ۱۰۸
حساب جاری ۱۰۹
حساب اعتباری ۱۰۹
تعریف مسأله ۱۰۹
توسعه تعریف مسأله ۱۱۱
حل و بحث ۱۱۲
کارگاه ۴: مطالعه موردی: همانی، مالکیت
و java.util.Stack ۱۱۷
تعریف مسأله ۱۱۸
حل و بحث ۱۱۸
خلاصه ۱۱۹
پرسشها و پاسخها ۱۲۰
کارگاه ۱۲۰
پرسشها ۱۲۰
تمرین‌ها ۱۲۰

روز ۶

چند شکلی بودن: بیاموزیم

آینده را پیش‌بینی کنیم ۱۲۱

چندشکلی ۱۲۲
چندشکلی بودن درونی ۱۲۵
چندشکلی بودن پارامتری ۱۳۰
روالهای پارامتری ۱۳۰
انواع پارامتری ۱۳۱
جایگزینی ۱۳۲
سربارگذاری ۱۳۳
تحمیل ۱۳۴
چندشکلی بودن مؤثر ۱۳۴
دامهای چندشکلی ۱۳۶
دام ۱: بالا رفتن رفتارها در سلسله مراتب ... ۱۳۶
دام ۲: کاهش کارایی ۱۳۶
دام ۳: محدودکنندگی ـ چشم‌بندها ۱۳۶
هشدار ۱۳۷
چگونه چندشکلی بودن اهداف
برنامه‌نویسی شیءگرا ۱۳۷
را تأمین می‌کند ۱۳۷
خلاصه ۱۳۸
پرسش‌ها و پاسخ‌ها ۱۳۹
کارگاه ۱۳۹
پرسشها ۱۳۹
تمرین‌ها ۱۴۰

روز ۷

چند شکلی بودن: زمان نوشتن کد ۱۴۱

کارگاه ۱: اعمال کردن چند شکلی ۱۴۱
تعریف مسأله ۱۴۷
حل و بحث ۱۴۸
کارگاه ۲: حساب بانکی ـ اعمال چندشکلی
بر روی مثالی آشنا ۱۴۹
تعریف مسأله ۱۴۹
حل و بحث ۱۵۱
کارگاه ۳: حساب بانکی ـ استفاده از قابلیت
چندشکلی برای نوشتن ۱۵۳
کدهای آینده‌نگر ۱۵۳
تعریف مسأله ۱۵۴
حل و بحث ۱۵۵
حل مشکل ۱۵۹
تعریف مسأله ۱۶۰
حل و بحث ۱۶۲

خلاصه ۱۶۴
پرسشها و پاسخها ۱۶۴
کارگاه ۱۶۴
پرسشها ۱۶۴
تمرین‌ها ۱۶۴

روز ۸

آشنایی با UML ۱۶۵

آشنایی با زبان مدلسازی یکپارچه ۱۶۶
مدلسازی کلاس‌ها ۱۶۷
نمادهای بنیادی کلاس ۱۶۷
نمادهای پیشرفته کلاس ۱۶۸
مدلسازی هدفمند کلاس‌ها ۱۶۹
مدلسازی روابط یک کلاس ۱۷۰
وابستگی ۱۷۰
تشریک ۱۷۱
اجماع ۱۷۲
ترکیب ۱۷۳
تعمیم ۱۷۴
جمع‌بندی ۱۷۵
خلاصه ۱۷۶
پرسش‌ها و پاسخ‌ها ۱۷۶
کارگاه ۱۷۷
پرسشها ۱۷۷
تمرین‌ها ۱۷۷

روز ۹

مقدمه‌ای بر تحلیل شیءگرا

(OOA) ۱۷۹

مراحل توسعه نرم‌افزار ۱۸۰
فرایندهای تکراری ۱۸۱
یک متدولوژی سطح بالا ۱۸۳
تحلیل شیءگرا (OOA) ۱۸۳
استفاده از مدل کاربرد برای استنتاج
کاربردهای سیستم ۱۸۴
نامگذاری و تعریف مجدد موارد کاربرد ... ۱۸۷
ساخت مدل دامنه ۱۹۶
حال چه کنیم؟ ۱۹۷
خلاصه ۱۹۷
پرسشها و پاسخها ۱۹۸
کارگاه ۱۹۸
پرسشها ۱۹۸
تمرین‌ها ۱۹۹

روز ۱۰

آشنایی با روش طراحی شیءگرا

(OOD) ۲۰۱

روش طراحی شیءگرا ۲۰۲
چگونه OOD را اعمال کنیم؟ ۲۰۳
گام اول: تهیه فهرست اولیه اشیاء ۲۰۴
گام دوم: بازنگری مسئولیتهای اشیاء ۲۰۴
گام سوم: ایجاد نقاط تعامل ۲۱۰
گام چهارم: تعیین جزییات روابط
بین کلاسها ۲۱۱
گام پنجم: ایجاد مدل ۲۱۱
خلاصه ۲۱۲
پرسش‌ها و پاسخ‌ها ۲۱۲
کارگاه ۲۱۳
پرسشها ۲۱۳
تمرین‌ها ۲۱۳

روز ۱۱

استفاده مجدد از طرحها از طریق الگوهای طراحی ۲۱۵

استفاده مجدد از طراحی ۲۱۶
الگوهای طراحی ۲۱۶
نام الگو ۲۱۶
مسأله ۲۱۷
راه حل ۲۱۷
نتایج ۲۱۷
واقعیتهای الگوها ۲۱۸
الگوها ۲۱۸
الگوها از دریچه مثال ۲۱۸
الگوی وفق دهنده ۲۱۸
الگوی واسط ۲۲۲
الگوی تکرار ۲۲۴
خلاصه ۲۳۱
پرسشها و پاسخها ۲۳۱
کارگاه ۲۳۱
پرسشها ۲۳۱
تمرین‌ها ۲۳۲

روز ۱۲

الگوهای پیشرفته طراحی ۲۳۵

مثالهای دیگر از الگوها ۲۳۵
الگوی مجرد عامل ۲۳۶
الگوی تک‌برگ ۲۳۹
الگوی شمارش با نوع محافظت شده ۲۴۳
دامهای الگوها ۲۴۸
خلاصه ۲۴۸
پرسشها و پاسخها ۲۴۸
کارگاه ۲۴۹
پرسشها ۲۴۹
تمرین‌ها ۲۴۹

روز ۱۳

شیءگرایی و برنامه‌نویسی رابط کاربر ۲۵۱

شیءگرایی و رابط کاربر ۲۵۱
اهمیت رابطهای کاربری منقطع از هم ۲۵۲
مدل ۲۵۶
نما ۲۵۹
کنترل کننده ۲۶۲
مشکلات مرتبط با الگوی MVC ۲۶۵
تأکید بر روی داده‌ها ۲۶۵
اتصال محکم ۲۶۵
ناکارآمدی ۲۶۵
خلاصه ۲۶۶
پرسشها و پاسخها ۲۶۶
کارگاه ۲۶۶
پرسش‌ها ۲۶۷
تمرین‌ها ۲۶۷

روز ۱۴

آزمون: راه اعتماد به نرم‌افزار ۲۷۱

آزمون نرم‌افزار شیءگرا ۲۷۲
آزمون و فرایند توسعه نرم‌افزار تکراری ۲۷۲
اشکال آزمون ۲۷۴
آزمون واحد ۲۷۴
آزمون مجتمع ۲۷۴

آزمون سیستم ۲۷۴
آزمون واپسگرد ۲۷۵
راهنمای نوشتن کد قابل اعتماد ۲۷۵
ترکیب توسعه و آزمون ۲۷۶
نوشتن کد استثناء ۲۸۹
نوشتن مستندات مؤثر ۲۹۰
خلاصه ۲۹۲
پرسشها و پاسخها ۲۹۲
کارگاه ۲۹۳
پرسشها ۲۹۳
تمرین‌ها ۲۹۳

روز ۱۵
ادغام تئوری و عمل ۲۹۵
چرا Backjack؟ ۲۹۶
چشم‌انداز ۲۹۶
جایگزینی نیازمندیها ۲۹۶
تحلیل مقدماتی Blackjack ۲۹۷
قوانین Blackjack ۲۹۷
تشخیص عاملها ۲۹۹
ساخت لیستی مقدماتی از موارد کاربردی .. ۳۰۰
طراحی تکرارها ۳۰۰
تکرار اول: بازی کردن مقدماتی ۳۰۱
تکرار دوم: قوانین ۳۰۱
تکرار سوم: شرط‌بندی ۳۰۲
تکرار چهارم: رابط کاربری (UI) ۳۰۲
تکرار اول: بازی کردن مقدماتی ۳۰۲
تحلیل Blackjack ۳۰۲
مدلسازی موارد کاربردی ۳۰۵
طراحی Blackjack ۳۰۷
پیاده‌سازی ۳۱۰
اشتباهات در برنامه‌نویسی رویه‌ای ۳۲۳
آزمایش سیستم ۳۲۴
خلاصه ۳۲۴
پرسشها و پاسخها ۳۲۵
کارگاه ۳۲۵
پرسشها ۳۲۵
تمرین‌ها ۳۲۵

روز ۱۶
تکرار دوم Blackjack:
افزودن قوانین ۳۲۷
قوانین Blackjack ۳۲۷
تحلیل قوانین ۳۲۷
طرح قوانین ۳۳۱
پیاده‌سازی قوانین ۳۳۷
خلاصه ۳۵۰
پرسشها و پاسخها ۳۵۰
کارگاه ۳۵۰
پرسشها ۳۵۰
تمرین‌ها ۳۵۱

روز ۱۷
تکرار سوم Blackjack:
اضافه کردن شرط‌بندی ۳۵۳
شرط‌بندی در بازی ۳۵۳
تحلیل شرط‌بندی ۳۵۴
طراحی شرط‌بندی ۳۵۷
پیاده‌سازی شرط‌بندی ۳۶۰
یک آزمایش کوچک: یک شیء کاذب ۳۶۶
خلاصه ۳۶۷

پرسشها و پاسخها ۳۶۷
کارگاه ۳۶۸
پرسشها ۳۶۸
تمرین‌ها ۳۶۸

روز ۱۸

تکرار چهارم Blackjack:

افزودن رابط گرافیکی کاربر ۳۶۹

نمایش Blackjack ۳۶۹
تکمیل رابط خط فرمان ۳۷۰
تحلیل GUI بازی ۳۷۱
حالتهای GUI ۳۷۱
مدل نمایشی GUI ۳۷۳
طراحی GUI بازی ۳۷۴
کارتهای CRC برای GUI ۳۷۴
ساختار GUI ۳۷۵
دیاگرام کلاس GUI ۳۷۷
پیاده‌سازی GUI بازی ۳۷۷
نهایی کردن کار در BlackjackGUI ۳۸۴
خلاصه ۳۸۴
پرسشها و پاسخها ۳۸۵
کارگاه ۳۸۵
پرسشها ۳۸۵
تمرین‌ها ۳۸۵

روز ۱۹

اعمال روشی متفاوت با..... ۳۸۷

یک رابط گرافیکی دیگر برای Blackjack ۳۸۷
لایه‌های PAC ۳۸۸
فلسفه PAC ۳۸۸
چه زمانی از الگوی طراحی PAC
استفاده کنیم ۳۸۸
تشخیص اجزای لایه نمایش ۳۸۹
طراحی اجزای لایه انتزاعی ۳۹۰
طراحی کنترل ۳۹۱
استفاده از الگوی Factory جهت پیشگیری از
خطاهای مشترک ۳۹۲
پیاده‌سازی VCard و VHand ۳۹۳
پیاده‌سازی VBettingPlayer ۳۹۵
پیاده‌سازی VBlackjackDealer ۳۹۷
پیاده‌سازی GUIPlayer ۳۹۸
ترکیب همه کلاسها با هم به همراه کنترل ... ۳۹۸
خلاصه ۴۰۰
پرسشها و پاسخها ۴۰۰
کارگاه ۴۰۱
پرسشها ۴۰۱
تمرین‌ها ۴۰۱

روز ۲۰

کمی تفریح با بازی ۴۰۳

تفریح با قابلیت چندشکلی ۴۰۳
ایجاد یک بازیکن ۴۰۳
کلاس SafePlayer ۴۰۴
اضافه کردن SafePlayer به GUI ۴۰۴
OOP و شبیه‌سازی ۴۰۵
بازیکنان Blackjack ۴۰۶
خلاصه ۴۱۱
پرسشها و پاسخها ۴۱۱
کارگاه ۴۱۱
پرسشها ۴۱۱
تمرین‌ها ۴۱۱

روز ۲۱

آخرین قدم ۴۱۳

نهایی کردن ۴۱۳
بازسازی طراحی بازی Blackjack
برای استفاده..... ۴۱۴
مجدد در دیگر سیستمها ۴۱۴
مزایایی که OOP برای بازی Blackjack
به ارمغان آورده است..... ۴۱۸
حقایق مربوط به صفت نرم‌افزار و OOP.... ۴۱۹
خلاصه ۴۱۹
پرسشها و پاسخها..... ۴۲۰
کارگاه..... ۴۲۰
پرسشها ۴۲۰
تمرین‌ها..... ۴۲۰

پیوست الف

پاسخ به تمرینها ۴۲۱

روز ۱

Object Oriented Programming

مقدمه‌ای بر برنامه‌نویسی شیءگرا

اگرچه زبان‌های شیءگرا از دهه ۱۹۶۰ پدید آمدند، تنها ۱۰ سالی است که فناوریهای شیءگرا به صورت غیرموازی رشد کرده و توسط صنایع نرم‌افزاری مقبول افتاده‌اند. موفقیتهای اخیری همچون CORBA, Java و C++ تکنیک‌های شیءگرایی (OO) را به سطوح جدیدی از مقبولیت پیش برده است. این امر اتفاقی نیست. پس از سالها شاگردی در مراکز علمی و مبارزه با تجربیات غیرعلمی، برنامه‌نویسی شیءگرا (OOP) به نقطه‌ای صعود کرده است که برنامه‌نویسان می‌توانند تکنیکهایی را که شیءگرایی قول آنها را داده بود، تحقق بخشند. در گذشته، مجبور بودید که از رییستان جهت نوشتن برنامه‌ای با تکنیک شیءگرا اجازه بگیرید. امروزه بسیاری از شرکتها استفاده از این تکنیک را اجباری کرده‌اند. دیگر با خیال راحت می‌توان گفت که شیءگرایی فراگیر شده است. ممکن است شما کسی باشید که تجربیات متوسطی در زمینه برنامه‌نویسی داشته باشد. جدای از آنکه، پیش‌زمینه‌ای در C، ویژوال بیسیک یا فرترن داشته‌اید، ممکن بوده نیم نگاهی به این تکنیک انداخته ولی اکنون تصمیم گرفته‌اید که برنامه‌نویسی شیءگرا را به صورتی جدی دنبال کنید و درواقع آن را به عنوان یکی از قابلیتها و مهارتهای خود درآورید.

اگرچه ممکن است تجربیات چندی با زبان‌های شیءگرا داشته باشید، این

کتاب کمک می‌کند درک خود را از شیءگرایی بهبود ببخشید. ولی اگر با زبان‌های شیءگرا آشنا نیستید دستپاچه نشوید. اگرچه این کتاب از زبان برنامه نویسی Java برای آموزش مفاهیم شیءگرایی استفاده می‌کند با این حال نیازی به داشتن تجربه و دانش در این زبان نخواهید داشت. با این حال اگر احساس می‌کنید کمی گیج شده‌اید و یا آنکه می‌خواهید شیوه دستوری تابعی را بدانید کافی است به راهنمای Java مراجعه کنید.

جدای از آنکه می‌خواهید شیءگرایی را برای تطبیق خود با صنعت نرم‌افزار فراگیرید و یا آنکه می‌خواهید جدیدترین پروژه خود را از این طریق شروع کنید، جای درستی آمده‌اید. اگرچه هیچ کتابی نمی‌تواند همه آنچه را که برای دانستن شیءگرایی لازم است به شما یاد دهد، ولی این کتاب می‌تواند بنیانی محکم از شیءگرایی را در شما ایجاد کند. با این بنیان، می‌توانید شیءگرایی را به صورت عملی آغاز کنید. مهمتر آنکه، این بنیان، آخرین اصولی را که نیاز است، به آنها دست پیدا کرده و بر آنها مسلط شوید به شما منتقل می‌کند.

آنچه امروز خواهید آموخت

- برنامه‌نویسی شیءگرا در گذر زمان
- پایه‌های برنامه‌نویسی شیءگرا
- مزایا و اهداف برنامه‌نویسی شیءگرا
- اشتباهات و اشکالات معمول در برنامه‌نویسی شیءگرا

برنامه‌نویسی شیءگرا در گذر زمان

برای فهم وضعیت فعلی OOP، بهتر است تاریخچه‌ای از برنامه‌نویسی را بدانید. هیچ کس یک شبه تصوری از OOP پیدا نمی‌کند. در عوض OOP جایگاه خاصی در متحول کردن توسعه نرم‌افزار دارد. در طول زمان OOP تبدیل به انتخابی آسان در کار عملی شده است. OOP روشهای عملی تجربه شده در طول زمان را به صورتی بهینه تا آنجا که ممکن باشد، ارایه می‌کند.

واژه جدید

OO کوتاه شده Object-Oriented به معنای شیءگرا است. درواقع OO چتری است که همه شیوه‌های توسعه که مبتنی بر مفهوم شیء - چیزی که مشخصه و رفتاری را از خود بروز می‌دهد - باشد، را در بر می‌گیرد. هم می‌توان روشهای شیءگرایی را برای برنامه‌نویسی به خدمت گرفت همچنانکه می‌توان از آن برای تحلیل و طراحی نیز استفاده کرد.

می‌توان گفت OO یک شیوه تفکر، یک روش نگریستن به دنیا و دیدن همه انواع اشیاء است.

به طور ساده، OO شامل همه چیزهایی است که پیشوند شیءگرایی به آنها تعلق می‌گیرد. عبارت OO را به کرات در این کتاب خواهید دید.

مقدمه‌ای بر OOP

امروزه، زمانی که از یک کامپیوتر استفاده می‌نمایید، درواقع از ۵۰ سال تجربه بهره می‌گیرید. برنامه‌نویسی در آن سالها نیاز به تبحر خاصی داشت: برنامه‌نویسان برنامه را از طریق تعدادی کلید (سوئیچ) مستقیماً وارد حافظه اصلی کامپیوتر می‌کردند. درواقع برنامه‌نویسان، برنامه‌های خود را با زبان ماشین وارد کامپیوتر

می‌نمودند. نوشتن برنامه به زبان ماشین مولد خطاهای بی‌شمار بوده و به دلیل فقدان یک ساختار مناسب، نگهداری و پشتیبانی از آن تقریباً غیرممکن بود. به علاوه آن‌که دسترسی به کدهای ماشین نیز غیرممکن می‌نمود.

با همه‌گیر شدن کامپیوترها، زبان‌های سطح بالاتر و ماژولار به وجود آمدند. اولین آنها فرترن (Fortran) بود. در هر حال زبان‌های ماژولار بعدی همچون الگول (ALGOL) تأثیر بیشتری بر روی OO گذاشتند. زبان‌های رویه‌ای به برنامه‌نویس اجازه می‌دادند که برنامه را به تعدادی از اجزایی مناسب جهت پردازش بر روی داده‌ها تقسیم کند. این رویه‌ها درواقع ساختار کل برنامه را تعیین می‌کردند. صدا کردن (فراخوانی) ترتیبی این رویه‌ها اجرای برنامه‌ای ماژولار را به دنبال داشت. برنامه در انتها پس از اتمام فراخوانی فهرستی از رویه‌ها به پایان می‌رسید.

این الگو بهبودهای چندی را نسبت به زبان ماشین نشان می‌داد، از جمله ساختارمندی برنامه از طریق رویه‌ها. توابع کوچکتر تنها برای فهمیدن آسان نیستند، بلکه برای اشکالزدایی نیز آسان‌تر هستند. از سوی دیگر، برنامه‌نویسی رویه‌ای استفاده مجدد از کد نوشته شده را محدود می‌کرد و غالباً باعث می‌شد که برنامه‌نویسان کدهای اسپاگتی تولید کنند. کدهایی که مسیر و روند اجرای آنها همچون وجود ظرفی پر از اسپاگتی است. در آخر آنکه طبیعت ذاتی برنامه‌نویسی رویه‌ای خود مشکلاتی را به بار می‌آورد. چرا که داده‌ها و رویه‌ها از یکدیگر جدا بودند. درواقع هیچگونه کپسوله‌سازی (Encapsulation) از داده‌ها وجود نداشت. این امر باعث می‌شد که هر رویه‌ای نداند چگونه به طور صحیح با داده‌ها برخورد کند. متأسفانه یک تابع در صورتی که نحوه رفتار درست با داده‌ها را نمی‌دانست، می‌توانست باعث تولید خطاهایی گردد. از آنجا که هر رویه می‌بایست دانش نحوه رفتار با داده‌ها را برای خود نگهدارد، هر تغییری در داده‌ها نیازمند تغییر در تمام رویه‌هایی بود که به داده‌ها دسترسی داشتند. به همین دلیل یک تغییر کوچک موجب تغییرات سلسله مراتبی در دیگر رویه‌ها می‌شد. به عبارت دیگر کابوسی برای نگهداری و پشتیبانی از برنامه!

برنامه‌نویسی ماژولار، با زبانی نظیر Modula2 سعی در رفع مشکلاتی داشت که در برنامه‌نویسی رویه‌ای ایجاد شده بود. برنامه‌نویسی ماژولار برنامه‌ها را به تعدادی جزء (Component) یا ماژول تقسیم می‌کند. برخلاف برنامه‌نویسی رویه‌ای که داده‌ها و رویه‌ها را از هم جدا می‌کند، ماژولها این دو را با هم ترکیب می‌کند. یک ماژول درواقع شامل داده و رویه‌هایی است که بر روی داده عمل می‌کنند. هرگاه نیاز باشد که قسمتی از برنامه از ماژولی استفاده نماید، به سادگی رابط (Interface) آن ماژول را فراخوانی خواهد کرد. از آنجا که ماژول تمام داده‌های درونی را از باقی برنامه مخفی می‌نماید، به راحتی می‌توان ایده حالت (وضعیت - State) را معرفی کرد: ماژولی حالتی را در خود نگه می‌دارد که ممکن است در طول زمان تغییر کند.

واژه جدید وضعیت یک شی درواقع مجموعه‌ای از متغیرهای درونی آن شیء است.

واژه جدید یک متغیر درونی مقداری است که در یک شیء نگهداری می‌شود.

با این حال ماژولها هم دچار کمبودهایی هستند. درواقع ماژولها قابل توسعه نیستند. این بدان معناست که نمی‌توان تغییرات افزایشی را بر روی یک ماژول اعمال کرد بدون آنکه کدهای آن را به طور مستقیم باز کرده و تغییر داد. همچنین نمی‌توان مبنای یک ماژول را بر ماژول دیگری گذاشت. همچنین یک ماژول

می‌باید نوعی را تعریف کند و یک ماژول نمی‌تواند نوع یک ماژول دیگر را به اشتراک گذارد.

در زبان‌های ماژولار و رویه‌ای، داده‌های ساختاریافته و غیرساختاریافته دارای یک نوع هستند(Type). یک نوع می‌تواند به سادگی قالب(فرمت) درون حافظه‌ای یک داده باشد. زبان‌های مبتنی بر نوع نیازمند آن هستند که هر شیء دارای یک نوع بوده و یا نوعی را تعریف نمایند. با این حال انواع داده‌ای نمی‌توانند به نحوی توسعه یابند، به گونه‌ای که دیگر انواع داده‌ای را ایجاد نمایند مگر به شیوه‌ای که ما آن را اجتماع(Aggregation) می‌نامیم. برای مثال، در زبان C، دو نوع داده‌ای مرتبط با هم به صورت زیر داریم:

```
typedet struct
{
   int a;
   int b;
} aBaseType;
typedet struct
{
   aBaseType Base;
   int c;
} aDerviedType;
```

در این مثال، aDerivedType مبتنی بر aBaseType است، ولی ساختمان aDerivedType نمی‌تواند به طور مستقیم همچون ساختمان aBaseType رفتار کند. تنها می‌توان به اعضای پایه ساختمان aDerivedType رجوع کرد. متأسفانه، در این حالت کدهای برنامه بسیار مبتنی بر دستورات سوئیچ و بلوک‌های شرطی if/else خواهند بود، چراکه برنامه باید بداند چگونه با هر ماژول رفتار کند.

در آخر، برنامه‌نویسی ماژولار، خود یک نوع برنامه‌نویسی رویه‌ای است که برنامه را به تعدادی رویه تقسیم می‌کند. حال به جای آنکه بر روی داده‌ها به صورت مستقیم عملی صورت گیرد، رویه‌ها با ماژولها سر و کار پیدا می‌کنند.

برنامه‌نویسی شیءگرا

OOP قدم منطقی بعدی پس از برنامه‌نویسی ماژولار بود که با اضافه کردن وراثت و پلی مورفیسم (چندشکلی) بر ماژولها به وجود آمد. OOP یک برنامه را به تعدادی از اشیاء سطح بالا تقسیم می‌کند. هر شیء قسمتی از مسأله‌ای که به دنبال حل آن هستید، را مدل می‌کند. نوشتن یک فهرست ترتیبی از فراخوانی رویه‌ها در روند اجرای برنامه دیگر جزیی از برنامه‌نویسی به صورت شیءگرا نیست. در عوض اشیاء با یکدیگر در حال تعامل هستند به نحوی که اجزای برنامه همگی با یکدیگر برنامه را پیش می‌برند. در این صورت، یک برنامه شیءگرا به صورت مدل و شبیه‌سازی زنده‌ای از مسأله‌ای که در حال حل آن هستید، در می‌آید.

روش OOP برای نرم‌افزارها با استفاده از اشیاء

تصور کنید در حال توسعه یک برنامه شیءگرا هستید تا یک سیستم فروش روی خط (Online) و یا پایانه فروش را پیاده‌سازی نماید. یک برنامه شیءگرا شامل آیتمهایی نظیر کارت خرید، کوپن و صندوق است. هریک از این اشیاء با دیگری در حال تعامل است تا برنامه را به حالت اجرا درآورند. برای مثال زمانی که صندوقدار هزینه سفارشات را جمع می‌زند، باید قیمت هر آیتم را چک نماید.

تعریف یک برنامه به عباراتی از اشیاء یک روش و دیدی از نرم‌افزار ارایه می‌دهد. در واقع اشیاء باعث می‌شوند که شما به هر چیزی به عنوان یک سطح مفهومی نگاه کنید که بدانید آن شیء قرار است چه کاری انجام دهد؛ به عبارت دیگر رفتارهای آن شیء. نگاه کردن به یک شیء از منظر مفهومی بسیار متفاوت است با دیدن آن شیء از این منظر که چیزی انجام شده است، به عبارت دیگر پیاده‌سازی. این عمل باعث می‌شود به برنامه خود به صورتی طبیعی و با عباراتی که در دنیای واقعی وجود دارد نگاه کنید. به جای آنکه برنامه‌تان را با مجموعه‌ای از رویه‌ها و داده‌های جدا از هم مدل کنید، (عبارات موجود در دنیای کامپیوتر)، آن را با اشیاء مدل می‌کنید.

اشیاء اجازه می‌دهند برنامه را با اسامی، افعال و صفات حوزه مسأله‌تان مدل کنید.

واژه جدید

پیاده‌سازی(Implementation) بیان می‌کند چگونه چیزی انجام شود. به زبان برنامه‌نویسی، پیاده‌سازی به معنای نوشتن کد است.

واژه جدید

حوزه یا دامنه(Domain) درواقع فضایی است که مسأله در آن سیر می‌کند. حوزه مجموعه‌ای از مفاهیمی است که دیدگاههای مهم مسأله‌ای را که سعی در حل آن دارید، نشان می‌دهد.

زمانی که به گذشته برمی‌گردید و به مسأله‌ای که می‌خواهید آن را حل کنید، فکر می‌کنید، دیگر خودتان را در گرداب جزئیات پیاده‌سازی (کدنویسی) غرق نمی‌کنید. بله، تعدادی از اشیاء سطح بالای نیاز خواهند داشت که در بعضی از سطوح پایین از طرق مختلف در حال تعامل باشند. با این حال، اشیاء از باقی سیستم ایزوله خواهند بود. (درس روز دوم مزایای این تفکر را تشریح خواهد کرد.)

نکته

در مسأله خرید و فروش، مخفی کردن پیاده‌سازی، بدین معناست که صندوقدار نیازی ندارد که نگاهی به مجموعه داده‌ها کند تا سفارشات را جمع بزند. درواقع صندوقدار چیزی از محل قرارگیری آرایه‌ها، شماره آیتمها و کلاً دیگر متغیرها و یا کوپن نمی‌داند. در عوض، صندوقدار با هریک از آن اشیاء در حال تعامل است. او تنها می‌داند که چگونه از اشیاء سؤال کند، هزینه و قیمت فلان آیتم چقدر است.

از این لحظه به بعد می‌توانید، شیء را تعریف کنید:

واژه جدید

یک شیء، جزیی از نرم‌افزار است که رفتار و حالت را در خود کپسوله (نگهداری) کرده است. اشیاء به برنامه‌نویس اجازه می‌دهند تا نرم‌افزار را با عبارات و تجریدهای دنیای واقعی مدل کند.

واضح‌تر که بخواهیم صحبت کنیم، یک شیء در واقع یک نمونه از یک کلاس است. قسمت بعدی مفهوم کلاس را در OOP روشن می‌کند.

همانگونه که دنیای واقعی از اشیاء ساخته شده است، به همین صورت هم می‌توان گفت نرم افزارهای شیءگرا مبتنی بر اشیاء هستند. در زبان‌های برنامه‌نویسی شیءگرای خالص، همه چیز توسط اشیاء صورت می‌گیرد. از انواع داده‌ای پایه نظیر اعداد صحیح(Integer) و یا بولی(Boolean) تا کلاسهای پیچیده توسط اشیاء تعریف می‌گردند. البته تمامی زبان‌های شیءگرا تا این حد پیش نمی‌روند. در بعضی از زبان‌ها (نظیر Java)، انواع داده‌ای ساده‌ای نظیر int و یا float همچون دیگر اشیاء رفتار نمی‌کنند.

یک کلاس چیست؟

همچون اشیاء در دنیای واقعی، دنیای OOP، اشیاء را با رفتارها و خواص مشترکشان دسته‌بندی می‌کند.

زیست‌شناسی، سگها، گربه‌ها، فیلها و انسانها را همگی جز پستانداران تقسیم‌بندی می‌کند. مشخصه‌های مشترک باعث می‌شود که این آفریده‌های متفاوت در یک دسته قرار گیرند. در دنیای نرم‌افزار، کلاسها، اشیاء مرتبط با هم را به همان شیوه، دسته‌بندی می‌کنند.

یک کلاس تمام مشخصه‌های مشترک از یک نوع شیء را تعریف می‌کند. به ویژه آنکه، کلاس همه رفتارها و خواص(Attribute) یک شیء را نیز بیان می‌نماید. همچنین پیغامهایی که شیء باید به آنها پاسخ دهد، در کلاس تعریف می‌شوند. زمانی که شیء بخواهد رفتار شیء دیگری را مورد آزمایش قرار دهد، به طور مستقیم با شیء هدف درگیر نمی‌شود، بلکه از آن شیء می‌خواهد تا خود را تغییر دهد، این امر عموماً از طریق یکسری اطلاعات اضافی صورت می‌گیرد که ما آنها را ارسال پیغام می‌نامیم.

واژه جدید یک کلاس هم رفتارها و خواص مشترک را که یک نوع شیء به اشتراک می‌گذارد، تعریف می‌کند. اشیاء از یک نوع مشخص، رفتارها و صفات یکسانی از خود نشان می‌دهند. کلاسها بسیار شبیه الگوها (قالبها) عمل می‌کنند، بدین معنی که شما می‌توانید از کلاسها برای ایجاد نمونه‌ای از یک شیء استفاده نمایید.

واژه جدید صفات، مشخصه‌های قابل مشاهده یک کلاس هستند. رنگ چشم یا رنگ مو نمونه‌ای از صفات است. یک شیء می‌تواند صفتی را با در اختیار گذاردن یکسری متغیرهای درونی یا باز گرداندن مقداری توسط یک تابع، به معرض نمایش گذارد.

واژه جدید یک رفتار(Behavior)، عملی است که توسط شیء زمانی که پیغامی به آن ارسال می‌شود و یا در هنگام پاسخ به یک تغییر حالت، انجام می‌دهد. مثل این است که بگوییم، شیء کاری انجام داد.

نکته

> بکار بردن اصطلاحات ارسال پیغام، عملیات، فراخوانی متد و فراخوانی تابع بستگی به پیش‌زمینه شما دارد. در این حالت تفکر ارسال پیغام یک روش تفکر شیءگرایی است. ارسال پیغام پویا است. از نظر مفهومی شی و پیغام از یکدیگر جدا هستند.

یک شیء می‌تواند رفتار دیگر اشیاء را از طریق انجام دادن عملی بر روی آن شیء تجربه کند. در این صورت اصطلاح فراخوانی تابع یا فراخوانی متد یا روال(Method) یا ارسال یک پیغام(Message) را برای این منظور به کار خواهیم بست. آنچه که مهم است بیان این نکته است که هریک از موارد فوق رفتار شیء را بیان می‌کند.

زبان‌هایی نظیر Java و C++ که میراث دوران برنامه‌نویسی رویه‌ای هستند (زمانی که فراخوانی توابع به صورت ایستا-Static- صورت می‌گرفت) بیشتر از اصطلاح متد استفاده می‌کنند تا دیگر اصطلاحات.

این کتاب عموماً از اصطلاح متد استفاده خواهد کرد (چرا که نگاه ویژه‌ای به Java دارد). اگرچه مواقعی هم از اصطلاح پیغام در این رابطه استفاده شده است.

جمع مباحث: کلاسها و اشیاء

شیء آیتم را به عنوان مثال در نظر بگیرید. هر آیتم شامل شماره مشخصه (ID)، توضیحات (Description)، قیمت واحد (Unit Price)، تعداد (Quantity) و در صورت لزوم میزان تخفیف (Discount) است. هر آیتم

می‌داند چگونه تخفیف را اعمال کند.

در دنیای OOP همه اشیاء از نوع آیتم، نمونه هایی از کلاس Item هستند. یک کلاس Item می‌تواند چنین شکل و شمایلی داشته باشد:

```
public class Item {

    private double unit_price;
    private double discount;    // a percentage discount to apply to the price
    private int    quantity;
    private String description;
    private String id;

    public Item( String id, String description, int quantity, double price ) {
        this.id = id;
        this.description = description;

        if( quantity >= 0 ) {
            this.quantity = quantity;
        }
        else {
            this.quantity = 0;
        }

        this.unit_price = price;
    }
     public double getAdjustedTotal() {
        double total = unit_price * quantity;
        double total_discount = total * discount;
        double adjusted_total = total - total_discount;

        return adjusted_total;
    }

    // applies a percentage discount to the price
    public void setDiscount( double discount ) {
        if( discount <= 1.00 ) {
            this.discount = discount;
        }
        else {
            this.discount = 0.0;
        }
    }

    public double getDiscount() {
        return discount;
```

```
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity( int quantity ) {
        if( quantity >= 0 ) {
            this.quantity = quantity;
        }
    }

    public String getProductID() {
        return id;
    }

    public String getDescription() {
        return description;
    }
}
```

متدهایی همچون

```
public Item(String id, String description, int quantity, double price)
```

توابع سازنده(Constructor) نامیده می‌شود. این توابع عموماً جهت مقداردهی اولیه به متغیرهای درونی شیء به هنگام ایجاد آن، به کار می‌روند.

واژه جدید

سازنده‌ها متدهایی هستند که برای آماده کردن اشیاء و مقداردهی اولیه متغیرهای درونی آنها به هنگام ساخت آن شیء، به کار می‌روند. فرایند ساخت یک شیء را نمونه‌سازی می‌گویند. چراکه شیء نمونه‌ای از کلاس است.

نکته

در تابع سازنده و همچنین در طول مثالی که برای کلاس Item آوردیم، چندین بار از this استفاده کردیم. this درواقع مرجعی است که به شیء اشاره می‌کند. هر شیء شامل مرجعی است که به خودش اشاره می‌کند. درواقع هر نمونه از کلاس (شیء) از this برای دسترسی به متدها و متغیرهای خودش استفاده می‌کند.

متدهایی همچون getDescription(),setDiscount() و getAdjustedTotal() همگی رفتارهای کلاس Item هستند که صفتی (خاصیتی) را تنظیم کرده و یا آن را از طریق دستور return باز می‌گردانند. زمانی که صندوقدار بخواهد جمع کل همه چیزهایی که خریداری شده‌اند را، محاسبه کند، کافی است هر شیء (آیتم) را در نظر گرفته و پیغام getAdjustedTotal() را به آن شیء بفرستد.

description, quantity, discount, until_price و id همگی متغیرهای درونی(Internal Variables) کلاس Item هستند. این مقادیر حالت (وضعیت) شیء را در بر می‌گیرند. وضعیت یک شیء می‌تواند در طول زمان

تغییر کند. برای مثال خریدار در زمان خرید ممکن است کوپنی را برای آیتمی اعمال کند. اعمال کوپن به آیتم حالت (وضعیت) آیتم را به دلیل تغییر در مقدار تخفیف تغییر می‌دهد.

رفتارهایی همچون getDiscount(), getAdjustedTotal()، دست یابنده(Accesor) نیز نامیده می‌شوند، چراکه اجازه می‌دهند از طریق آنها به مقدار داده‌های درونی شیء، دسترسی داشته باشید. این دسترسی ممکن است همچون حالت getDiscount() به صورت مستقیم صورت گیرد. به عبارت دیگر، ممکن است شیء، قبل از اجازه دسترسی به مقدار یک متغیر درونی، بر روی آن پردازشهایی را صورت دهد. دقیقاً همانند آنچه که در getAdjustedTotal() اتفاق افتاده است.

واژه جدید دست یابنده اجازه دسترسی به داده‌های درونی یک شیء را ممکن می‌سازد. با این حال، دست یابنده اینکه داده در متغیری ذخیره شده است و یا در مجموعه‌ای از متغیرها و یا از روی متغیری محاسبه شده است، را از دید کاربر مخفی می‌کند. دست یابنده اجازه می‌دهد که مقدار متغیری را تغییر داده و یا دریافت کنید. در نتیجه می‌توانند بر روی حالت درونی شیء تأثیرگذار باشند.

رفتارهایی همچون setDiscount() تغییردهنده(Mutator) یا دگرگون کننده نامیده می‌شوند، چراکه اجازه می‌دهند حالت درونی شیء را تغییر دهید. یک تغییر دهنده می‌تواند ابتدا ورودیهای خود را پردازش کرده و در صورت الزام حالت درونی شیء را تغییر دهد. برای مثال به setDiscount() نگاهی داشته باشید. در این حالت setDiscount() ابتدا مطمئن می‌شود مقدار تخفیف (آرگومان ورودی) بیش از ۱۰۰٪ نیست و سپس تخفیف را اعمال می‌کند.

واژه جدید دگرگون کننده‌ها اجازه می‌دهند، حالت درونی شیء را تغییر دهید.

در زمان اجرا، برنامه شما از کلاسهایی نظیر Item استفاده کرده و اشیاء را ایجاد می‌کند. از این طریق برنامه (Application) ساخته می‌شود. هر نمونه جدید رونوشتی از کلاس مورد نظر است. پس از ایجاد، هر نمونه به طور مستقل رفتار خود را به نمایش گذاشته و وضعیت و حالت خود را دنبال می‌کند.

برای مثال اگر دو شیء آیتم از کلاس Item را ایجاد کرده باشید، یکی از آنها ممکن است دارای ۱۰٪ تخفیف و دیگری شامل تخفیف نباشد. بعضی از آیتمها ممکن است از دیگر آیتمها گرانتر باشند. یکی ممکن است ۱۰۰۰ دلار قیمت و حال آنکه دیگری تنها ۱/۹۸ دلار قیمت داشته باشد. اگرچه حالت هر آیتم در طول زمان ممکن است متفاوت باشد، نمونه اجرا شده به هر حال از نوع Item (کلاس Item) می‌باشد. دوباره به مثال زیست‌شناسی برگردیم: یک پستاندار خاکستری به هر حال نوعی پستاندار است همچون پستانداری که رنگ بدنش قهوه‌ای باشد.

جمع اشیاء برای آغاز کار

متد main() در زیر را در نظر بگیرید:

```
public static void main( String [] args ) {
   // create the items
   Item milk   = new Item( "dairy-011", "1 Gallon Milk", 2,2.50 );
   Item yogurt = new Item( "dairy-032", "Peach Yogurt", 4, 0.68 );
   Item bread  = new Item( "bakery-023", "Sliced Bread", 1, 2.55 );
   Item soap   = new Item( "household-21", "6 Pack Soap", 1, 4.51 );
```

```
    // apply coupons
    milk.setDiscount( 0.15 );
    // get adjusted prices
    double milk_price   = milk.getAdjustedTotal();
    double yogurt_price = yogurt.getAdjustedTotal();
    double bread_price  = bread.getAdjustedTotal();
    double soap_price   = soap.getAdjustedTotal();

    // print receipt
    System.out.println( "Thank You For Your Purchase." );
    System.out.println( "Please Come Again!" );
    System.out.println( milk.getDescription()   + "\t $" + milk_price   );
    System.out.println( yogurt.getDescription() + "\t $" + yogurt_price );
    System.out.println( bread.getDescription()  + "\t $" + bread_price  );
    System.out.println( soap.getDescription()   + "\t $" + soap_price   );

    // calculate and print total
    double total = milk_price + yogurt_price + bread_price + soap_price;
    System.out.println( "Total Price\t $" + total );
}
```

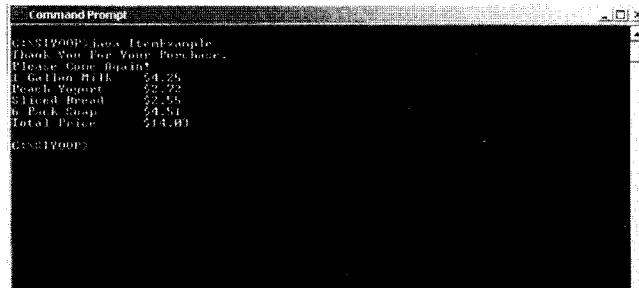
این متد نشان می‌دهد چگونه یک برنامه کوچک می‌تواند از کلاس Item استفاده کند. اول از همه برنامه چهار نمونه از کلاس Item ایجاد می‌کند. در یک برنامه حقیقی، برنامه باید نمونه‌ها را به هنگامی که کاربر کاتالوگ محصولات را مرور می‌کند، ایجاد نماید.

برنامه تعدادی از آیتمها را ایجاد کرده، مقدار تخفیف را اعمال کرده و سپس فاکتور فروش را چاپ می‌کند. برنامه تمام تعاملات با اشیاء را از طریق ارسال پیغامهای متفاوت به آنها انجام می‌دهد. برای مثال، برنامه ۱۵٪ تخفیف برای شیر را با ارسال پیغام setDiscount() به آن درخواست می‌کند. بالاخره برنامه با ارسال پیغام getAdjustedTotal() به هریک از آیتمها هزینه نهایی سفارش هر محصول را محاسبه می‌کند.

در آخر، برنامه اقدام به چاپ فاکتور فروش می‌نماید. شکل ۱ـ۱ خروجی مثال فوق را نمایش می‌دهد. این نکته حائز اهمیت است که تمام قسمتهای برنامه توسط اشیاء Item و تعاملات آنها بر اساس متدهای Item صورت می‌گیرد، درواقع اسامی و افعال موجود در حوزه فروش!

ارتباط اشیاء

اینکه چگونه اشیاء به یکدیگر مرتبط می‌شوند، جزء بسیار مهمی از OOP است. اشیاء می‌توانند به دو شیوه با یکدیگر مرتبط شوند: اول آنکه اشیاء می‌توانند کاملاً مستقل از یکدیگر وجود داشته باشند. دو شیء



شکل ۱ـ۱
چاپ فاکتور فروش

Item در کارت خرید در یک زمان واحد می‌توانند حضور داشته باشند. اگر این دو شیء مستقل و جدا از هم نیاز به تعامل با یکدیگر داشته باشند، می‌توانند از طریق ارسال پیغام به یکدیگر با هم ارتباط برقرار کنند.

واژه جدید

اشیاء از طریق ارسال پیغام(Message) با یکدیگر ارتباط برقرار می‌کنند. پیغامها باعث می‌شوند که اشیاء مبادرت به انجام کاری نمایند.

«ارسال پیغام» همچون فراخوانی یک متد باعث تغییر در حالت شیء شده و باعث انجام رفتاری در شیء می‌گردد.

دوم آنکه یک شیء می‌تواند شامل شیء دیگری باشد. همانگونه که اشیاء باعث ایجاد برنامه‌های OOP می‌شوند، می‌توانند جهت ایجاد دیگر اشیاء نیز به کار روند. از مثال Item ممکن است به این نکته پی برده باشید که شیء Item خود شامل دیگر اشیاء بوده است. برای مثال، شیء Item شامل توضیحات (Description) و شماره مشخصه (ID) است. این دو هریک شیء از نوع رشته‌ای (String) هستند. هریک از این اشیاء دارای رابطی(Interface) هستند که از طریق آن متدها و خواص در دسترس قرار می‌گیرند. به خاطر داشته باشید، در OOP، همه چیز یک شیء است. حتی آن قسمتهایی که در ساخت یک شیء دخالت دارند!

ارتباطات میان اشیاء درون یک شیء دیگر نیز به همین طریق صورت می‌گیرد. زمانی که شیء نیاز به تعامل داشته باشد، این کار از طریق ارسال پیغام صورت می‌گیرد.

پیغامها یک مفهوم مهم در شیءگرایی هستند. پیغامها اجازه می‌دهند که اشیاء مستقل از یکدیگر باقی بمانند. زمانی که یک شیء پیغامی به شیء دیگر می‌فرستد، در حالت کلی برای شیء مهم نیست که رفتار شیء داخلی چگونه است. شیء درخواستی باید واکنش مناسبی از خود بروز دهد.

هفته آینده در مورد ارتباطات میان اشیاء چیزهای بیشتری را فرا خواهید گرفت.

توجه

تعریف شیء (Object) همچنان برای مباحثه بازمانده است. بعضی از مردم یک شیء را به عنوان نمونه اجرا شده از کلاس تعریف نمی‌کنند. در عوض آنها همه چیز را به عنوان نوعی شیء تعریف می‌کنند: از این منظر، یک کلاس شیئی است که دیگر اشیاء را ایجاد می‌کند. رفتار کردن یک کلاس به عنوان یک شیء برای مفاهیمی همچون فوق کلاس(Meta Class) بسیار مهم است.

اگرچه در این کتاب با مجموعه اصطلاحات متضادی ممکن است روبرو شوید، با این حال ما یک تعریف را برمی‌گزینیم و بر روی همان تأکید می‌کنیم. انتخاب ما عموماً آن مفهومی است که در عمل بیشتر مورد استفاده قرار می‌گیرد. در اینجا ما تعریف شیء را به عنوان نمونه‌ای از یک کلاس برمی‌گزینیم. این تعریف، تعریفی است که توسط زبان مدلسازی واحد (UML) ارایه می‌شود و همانی است که در صنعت بیشترین مورد استفاده را دارد. (با UML در بخشهای بعدی بیشتر آشنا خواهید شد). متأسفانه، تعریف دوم، تعریفی خالصتر است. با این حال به دلیل آنکه مفهوم فوق کلاس از حوصله این کتاب خارج است، با آن سر و کار نخواهید داشت.

چگونه برنامه‌نویسی شیءگرا مبتنی برگذشته ساخته می‌شود؟

به عنوان یک مسأله دیگر که سعی دارد از قدرت مفاهیم گذشته استفاده کرده و خطاهای آن را جبران کند، OOP بر اساس برنامه‌نویسی ماژولار و رویه‌ای ساخته می‌شود.

برنامه‌نویسی ماژولار یک برنامه را بر اساس تعدادی ماژول ساختاربندی می‌کند. به همین صورت، OOP برنامه را به تعدادی از اشیاء در حال تعامل می‌شکند. همچون ماژول که نمایش داده‌ها را پشت رویه‌ها پنهان می‌کند، اشیاء حالت و وضعیت خود را در پشت رابطشان کپسوله می‌کنند. درواقع OOP مفهوم کپسوله‌سازی را مستقیماً از برنامه‌نویسی ماژولار به عاریت گرفته است.

کپسوله‌سازی با برنامه‌نویسی رویه‌ای بسیار متفاوت است. برنامه‌نویسی رویه‌ای داده‌ها را کپسوله نمی‌سازد. بلکه، داده‌ها برای همه روال‌هایی که می‌خواهند به آنها دسترسی داشته باشند، باز هستند. برخلاف برنامه‌نویسی رویه‌ای، برنامه‌نویسی شیءگرا داده‌ها و رفتارهایی که بر روی آنها اثر می‌گذارند را از طریق اشیاء به هم مرتبط می‌کند. مفهوم کپسوله‌سازی را در روزهای دوم و سوم بیشتر بررسی خواهید کرد.

اگرچه اشیاء از لحاظ مفهومی به ماژولها شبیه هستند، در تعدادی از مسایل دیگر با یکدیگر تفاوت دارند. اول آنکه ماژولها به طور واضح توسعه یافتگی را پشتیبانی نمی‌کنند. OOP مفهوم وراثت را به همین منظور معرفی کرده است. وراثت یا اشتقاق اجازه می‌دهد که به راحتی کلاسهای خود را توسعه داده و بهبود بخشید. وراثت همچنین اجازه می‌دهد که کلاسهای خود را دسته‌بندی نمایید. با مفهوم وراثت در روز چهارم و روز پنجم آشنا خواهید شد.

OOP همچنین مفهوم چندشکلی(Polymorphism) را ارایه می‌دهد، که ایجاد برنامه‌های منعطف را ممکن می‌سازد. چند شکلی قابلیت انعطاف را با روشن کردن نوع محدود شده ماژول فراهم می‌نماید. شما با چند شکلی در روز ششم و روز هفتم آشنا خواهید شد.

OOP مفاهیم چند شکلی‌بودن و کپسوله‌سازی را یقیناً ابداع نکرده است. آنچه واضح است آن است که OOP این دو مفهوم را در یکجا ترکیب کرده است. از طریق تعریف OOP برای اشیاء، شما این فن‌آوریها را از راهی که تاکنون بدان نپرداخته‌اید به کار خواهید بست.

مزایا و اهداف برنامه‌نویسی شیءگرا

برنامه‌نویسی شیءگرا شش هدف را در توسعه نرم‌افزار دنبال می‌کند. OOP کوشش می‌کند تا نرم‌افزاری را تولید نمایید که در بر گیرنده مشخصه‌های زیر باشد:

۱. سادگی یا طبیعی بودن(Natural)
۲. پایداری(Reliable)
۳. قابل استفاده مجدد(Reusable)
٤. قابل نگهداری(Maintainable)
۵. قابل توسعه(Extedable)
۶. به موقع(Timely)

اجازه دهید نگاهی داشته باشیم به هریک از اهداف فوق و اینکه چگونه OOP برای رسیدن به آنها تلاش می‌کند.

سادگی

OOP برنامه‌های طبیعی تولید می‌کند. برنامه‌های ساده بیشتر قابل فهم هستند. به جای آنکه برنامه‌نویسی را عبارتی همچون آرایه‌ای از نواحی حافظه بدانیم، می‌توانید برنامه‌نویسی را مجموعه اصطلاحات مسأله

(مشکل) مشخص خودتان بدانید. نیازی نیست که در جزییات کامپیوتر غوطه‌ور شوید تا برنامه‌تان را طراحی کنید. در عوض برنامه‌تان را محدود به زبان دنیای کامپیوتر بکنید. شیءگرایی شما را آزاد می‌کند تا برنامه‌تان را مجموعه عبارات مسأله‌تان توصیف کنید.

برنامه‌نویسی شیءگرا اجازه می‌دهد که مسأله را در سطح تابعی مدل کنید نه در سطح پیاده‌سازی. بنابراین نیازی نیست که بدانید قسمتهایی از نرم‌افزار چگونه کار می‌کنند تا از آن استفاده نمایید، لذا می‌توانید تنها بر روی آنچه که انجام می‌دهد، متمرکز شوید.

پایداری

برای ایجاد نرم افزارهای مفید، نیاز دارید که نرم‌افزاری بسازید که همچون دیگر محصولات نظیر یخچال و تلویزیون از پایداری خوبی برخوردار باشند. چه زمانی مایکروفر شما دچار مشکل شد؟

طراحی مناسب، پیاده‌سازی خوب و برنامه‌های شیءگرا، پایدار هستند. ذات ماژولار بودن اشیاء اجازه می‌دهد که قسمتهایی از برنامه را بدون آنکه دیگر اجزاء تحت تأثیر قرار گیرند، تغییر دهید. درواقع اشیاء حالت (وضعیت فعلی) و وظایفشان را از دیگر اشیاء ایزوله می‌کنند.

یکی از راههای افزایش پایداری، از طریق آزمایش کامل(Through Testing) به دست می‌آید. شیءگرایی نحوه آزمایش را از طریق ایزوله کردن اشیاء از یکدیگر، بهبود می‌بخشد. این ایزولاسیون اجازه می‌دهد هر جز را به طور مستقل مورد آزمایش قرار دهید. زمانی که یک جزء را آزمایش کرده و از آن اطمینان حاصل کردید، می‌توانید از آن با اعتماد کامل استفاده نمایید.

قابل استفاده مجدد

آیا بنا هر بار که می‌خواهد خانه‌ای را بنا کند، آجر جدیدی اختراع می‌کند؟ آیا مهندس الکترونیک هر بار که می‌خواهد مداری طراحی کند، مقاومت جدیدی را اختراع می‌کند؟ پس چرا برنامه‌نویسان می‌خواهند «دوباره چرخ را اختراع کنند؟» زمانی که مسأله‌ای یکبار حل شد، باید راه حل را دوباره استفاده کرد.

می‌توانید به راحتی از کلاسهای شیءگرا دوباره استفاده نمایید. مثل ماژولها می‌توانید از اشیاء در برنامه‌های مختلف استفاده مجدد کنید. برخلاف ماژولها، OOP وراثت را معرفی می‌نماید که از این طریق می‌توان اشیاء موجود را توسعه داده و از طریق چند شکلی می‌توانید کدهای عمومی و جامعی بنویسید.

شیءگرایی کد جامع را تضمین نمی‌کند. ایجاد کلاسهای با طراحی خوب اندکی مشکل است و نیاز به توجه و تمرکز بر روی تجرید(Abstraction) دارد. برنامه‌نویسان عموماً این روش را روشیء ساده نمی‌یابند.

از طریق OOP می‌توانید ایده‌های کلی را مدل کنید و از آن ایده‌ها جهت حل مسایل مشخص بهره جویید. همچنین اشیاء ساخته می‌شوند تا مسأله‌ای مشخص را حل کنند. عموماً برای ساخت این اشیاء مشخص از کدهای جامع و عمومی استفاده می‌شود.

قابلیت نگهداری

چرخه زندگی(Life Cycle) یک برنامه پس از ارایه آن به بازار به پایان نمی‌رسد. در عوض، باید نگهداری و پشتیبانی از کدهای خود را ادامه دهید. در حقیقت ۶۰ تا ۸۰ درصد زمانی که بر روی یک برنامه صرف می‌شود، صرف نگهداری آن می‌گردد. توسعه نرم‌افزار و نوشتن کدهای آن ۲۰ درصد این معادله را شامل می‌شود!

کدهای شیءگرای با طراحی خوب قابلیت نگهداری خوبی دارند. برای برطرف کردن خطایی در برنامه، کافی است تنها به یک قسمت از آن مراجعه کنید. از آنجا که تغییر در پیاده‌سازی کد بسیار شفاف صورت می‌گیرد، دیگر اشیاء به طور خودکار از این مزیت سود می‌جویند، زبان طبیعی برنامه (کد) باید به نحوی باشد که دیگر برنامه‌نویسان به راحتی آن را بفهمند.

قابلیت توسعه

همچنان که از کدهای خود نگهداری و پشتیبانی می‌نمایید، کاربران برای اضافه کردن قابلیتهای جدید به سیستم با شما تماس می‌گیرند. همچنانکه به ساخت مجموعه‌های اشیاء و کتابخانه‌های مرتبط با آنها می‌پردازید، باید قابلیتها و کارکردهای خود اشیاء را نیز توسعه دهید.

OOP درواقع به این درخواست جامه عمل می‌پوشاند. نرم‌افزار چیزی ایستا نیست. نرم‌افزار باید رشد کرده و در طول زمان تغییر یابد تا همچنان قابل استفاده باقی بماند. OOP خواصی را به برنامه‌نویس ارایه می‌دهد که از طریق آن می‌توان کدها را توسعه داد. این قابلیتها شامل وراثت، چند شکلی‌بودن، سربارگذاری، جایگزینی و تعدادی دیگر از الگوهای طراحی می‌باشند.

به موقع بودن

چرخه زندگی پروژه‌های نرم‌افزاری مدرن عموماً بر اساس هفته‌ها سنجیده می‌شود. OOP هدفش کوتاه کردن این چرخه توسعه است. درواقع OOP با استفاده از فراهم کردن توانایی‌هایی نظیر پایداری، قابلیت استفاده مجدد و قابلیت توسعه چرخه زمانی توسعه نرم‌افزار را کوتاه می‌کند.

نرم‌افزارهای طبیعی (ساده) طراحی سیستمهای پیچیده را ساده می‌کنند. تا زمانی که از طراحی دقیق چشم‌پوشی نکنید، نرم‌افزارهای (ساده) طبیعی می‌توانند چرخه طراحی را کوتاه نمایند، چراکه با OOP تنها بر روی مسأله‌ای که برای حل آن اقدام کرده‌اید، متمرکز شده‌اید.

زمانی که برنامه‌ای را به تعدادی از اشیاء می‌شکنید، توسعه هریک از آن قطعات می‌تواند به صورت موازی پیش رود. چند برنامه‌نویس می‌توانند بر روی کلاسهای مختلف به صورت مستقل کار کنند. توسعه نرم‌افزار به صورت موازی، زمان کمتری برای اتمام نرم‌افزار می‌طلبد.

دام‌ها

زمانی که برای اولین بار با شیءگرایی آشنا می‌شوید، چهار دام در کمین شماست که باید مراقب باشید در آنها گرفتار نشوید.

دام اول: این تفکر که OOP زبانی ساده است.

عموم مردم OOP را با زبان‌های شیءگرا یکی می‌گیرند. این اشتباه از آنجا نشأت می‌گیرد که شما به روش شیءگرایی برنامه می‌نویسید.

درواقع OOP چیزی فراتر از دانستن زبانی شیءگرا و آشنا بودن به مجموعه‌ای از تعریف‌های مشخص است. می‌توان با استفاده از زبانی شیءگرا، کدهای غیرشیءگرا نوشت. OOP واقعی تفکری است که باعث می‌شود بتوان مسأله را مانند مجموعه‌ای از اشیاء دید و از کپسوله‌سازی، وراثت و چند شکلی‌بودن به شیوه‌ای صحیح استفاده کرد. متأسفانه بسیاری از شرکتها و برنامه‌نویسان می‌پندارند اگر از زبانی شیءگرا

استفاده نمایند از تمام مزایای OOP بهره‌مند خواهند شد. زمانی که به مشکلی برمی‌خورند، بر تکنولوژی خرده می‌گیرند حال آن که این حقیقت را درک نکرده‌اند که باید کارمندان خود را به شیوه‌ای صحیح آموزش دهند و اینکه تا زمانی که عمق مفاهیم زبان را درک نکرده‌اند، گرفتار مفاهیم عامیانه آن زبان نشوند.

دام دوم: سفسطه و مغلطه

باید بیاموزید کدهای خود را بعدا دوباره مورد استفاده مجدد قرار دهید. یادگیری این امر که از کدهای خود استفاده مجدد کنید بدون آنکه گرفتار سفسطه و مغلطه شوید یکی از دروس سختی است که با آن مواجه خواهید شد. دو مشکل ممکن است در این رابطه رخ دهد. اول آنکه برنامه‌نویس دوست دارد چیزی را خلق نماید. اگر به «استفاده مجدد» به شیوه‌ای نادرست بنگرید، از خلاقیت و ابتکار به دور خواهید ماند. اگرچه باید به خاطر داشته باشید که استفاده مجدد برای آن است تا قطعه کدهای بزرگتری را بر اساس قطعات قبلی ایجاد نمایید. اینکه از یک جز استفاده مجدد نمایید چیز هیجان‌انگیزی نیست ولی شما را قادر می‌کند تا چیز بهتری را ایجاد نمایید.

دوم آنکه بسیاری از برنامه‌نویسان، به نرم‌افزارهایی که خود آنها ننوشته‌اند، اعتماد ندارند. اگر قسمتی از نرم‌افزار به درستی مورد آزمایش قرار گرفته باشد و در ضمن نیازهای شما را نیز برطرف نماید، شما باید از آن استفاده مجدد نمایید. هیچگاه به این دلیل که قسمتی از برنامه را خود ننوشته‌اید، از آن قسمت صرفنظر نکنید. به خاطر داشته باشید که استفاده مجدد از یک جزء، دست شما را باز می‌گذارد تا نرم‌افزارهای جالب دیگری بنویسید.

دام سوم: این تفکر که شیءگرایی همیشه شفابخش است

اگرچه OOP مزایای بسیار زیادی را در اختیار می‌گذارد، با این حال داروی همه دردهای دنیای برنامه‌نویسی نیست. زمانهایی وجود دارد که نباید از OO استفاده نمایید. بنابراین نیاز دارید که در هر زمانی ابزار درست را به دست بگیرید و به شیوه‌ای صحیح قضاوت نمایید. مهمتر آنکه OOP موفقیت پروژه شما را تضمین نمی‌کند. پروژه به طور خودکار به این دلیل که از زبانی شیءگرا استفاده کرده‌اید موفق نمی‌شود. موفقیت تنها زمانی حاصل می‌شود که با دقت طرح‌ریزی کرده و سپس کدنویسی نمایید.

دام چهارم: برنامه‌نویسی همراه با مخفی‌کاری

زمانی که برنامه می‌نویسید مخفی‌کاری نکنید. همانگونه که باید یاد بگیرید که از کدهای دیگران استفاده نمایید، باید کدهایی را که ایجاد کرده‌اید، به اشتراک بگذارید. به اشتراک‌گذاری بدین معناست که دیگر برنامه‌نویسان را نیز تشویق کنید از کلاسهایی که ایجاد کرده‌اید، استفاده نمایند. همچنین اشتراک‌گذاری بدین معناست که برای دیگران نیز استفاده مجدد از آن کلاسها آسان شود.

زمانی که برنامه می‌نویسید دیگر برنامه‌نویسان را از آنچه انجام می‌دهید، مطلع کنید. واضح و قابل فهم بنویسید. مهمتر آنکه سند فنی پروژه را تهیه کنید. تا آنجا که می‌توانید فعالیتهای خود را مستند سازید. هیچکس از چیزی که نمی‌تواند پیدا کند و یا آن را بفهمد نمی‌تواند استفاده مجدد کند.

هفته‌ای که در پیش رو دارید

در هفته‌ای که در پیش رو دارید، آشنایی شما با OOP، با یادگیری مطالبی که اساس تئوری OOP را شکل می‌دهند بیشتر خواهد شد: کپسوله‌سازی، وراثت و چندشکلی‌بودن.

هریک از این سه مورد خود به دو درس تقسیم می‌شود. درس اول تکنیک را شرح داده و تئوریهای پشت آن را بیان می‌کند. درس دوم به صورت عملی با آنچه که در روز قبل آموخته‌اید، شما را وارد معرکه می‌نماید. درواقع این روش، روشی موفق برای همپوشانی مطالب کلاسی و آزمایشگاهی است که در بسیاری از دانشگاهها اجرا گردیده است.

تمام مطالب عملی را با استفاده از زبان Java شرکت سان میکروسیستم(Sun Microsystem) تکمیل خواهید کرد. می‌توانید تمام ابزارهایی راکه در طول این کتاب استفاده شده است از طریق وب به طور رایگان دریافت نمایید. روز سوم شما را قدم به قدم جهت نصب و راه‌اندازی محیط توسعه(Development Environment) یاری می‌رساند.

نکته

چرا جاوا؟

دو دلیل برای آموزش Java وجود دارد. اول آنکه Java به طرز زیبایی شما را از جزئیات سیستم عامل و ماشین دور نگه می‌دارد. به جای آنکه نگران آن باشید که چگونه کارهای مربوط به تخصیص حافظه و یا رهاسازی آن را انجام دهید، به راحتی می‌توانید فکر خود را متمرکز یادگیری اشیاء نمایید. در آخر آنکه یادگیری خوب این زبان شیءگرا، عملی است. می‌توانید این دانش را فراگرفته و کاری برای خود دست و پاکنید.

تعدادی از زبان‌های دیگر، بیش از Java شیءگرا هستند. با این حال پیداکردن کاری مرتبط با Java ساده‌تر است.

خلاصه

امروز نگاهی به برنامه‌نویسی شیءگرا داشتیم. با نگاهی به انقلاب صورت گرفته در زمینه برنامه‌نویسی شروع کردیم و تعدادی از اصول OOP را آموختیم. هم اکنون می‌توانید ایده‌های مفهومی پشت سر شیءگرایی را همچون اینکه کلاس چیست و چگونه اشیاء با یکدیگر کار می‌کنند، بفهمید.

تعاریف مهم هستند ولی، نباید مسیری راکه به دنبال حل مسأله است با چراها و چگونه‌ها گم کنیم. شش مزیت و هدف برنامه‌نویسی شیءگرا به طور خلاصه عبارتند از:

۱. سادگی
۲. پایداری
۳. قابلیت استفاده مجدد
٤. قابلیت نگهداری
۵. قابلیت توسعه
۶. به موقع و به جا بودن

هیچگاه نباید از رسیدن به این اهداف گمراه شوید.

پرسش ها و پاسخها

برای مسلط شدن بر روی OOP چه کاری باید انجام دهم؟

کتابهایی نظیر این کتاب، شروع خوبی برای تسلط بر شیءگرایی است. این نکته مهم است که پایه‌ای محکم

در این زمینه بنا کنید.

پس از بنا کردن پایه‌ای محکم نیاز دارید که به صورتی فعال شیءگرایی را تجربه کنید. تسلط واقعی تنها با کار عملی میسر می‌شود. به عنوان یک برنامه‌نویس پروژه شیءگرایی جدیدی را شروع کنید. همچنانکه با جنبه‌های مختلف شیءگرایی آشنا می‌شوید، شروع کنید به تحلیل و طراحی پروژه‌تان.

همچنین خوب است که راهنمایی هم پیدا کنید. کسی را پیدا کنید تا بتواند اندکی برای شما وقت بگذارد. یادگیری از دیگران مهمترین و سریعترین راه یادگیری OOP است.

در آخر، مطالعات شخصی خود را ادامه دهید. کتابها و مقالات مختلف را مطالعه کرده و در کنفرانسها شرکت کنید. همواره نیازمند مطالعه و افزایش مهارتهای خود خواهید بود.

کارگاه

سؤالات و جوابهای آنها تنها برای افزایش درک و فهم شما تهیه شده‌اند.

پرسشها

۱. **برنامه‌نویسی رویه‌ای چیست؟**
۲. **مزیت برنامه‌نویسی رویه‌ای نسبت به برنامه‌نویسی غیرساختار یافته چیست؟**
۳. **برنامه‌نویسی ماژولار چیست؟**
٤. **مزیت برنامه‌نویسی ماژولار نسبت به برنامه‌نویسی رویه‌ای چیست؟**
۵. **تعدادی از برنامه‌های ماژولار و رویه‌ای را نام ببرید.**
۶. **برنامه‌نویسی شیءگرا چیست؟**
۷. **شش مزیت و هدف برنامه‌نویسی شیءگرا کدامند؟**
۸. **یکی از اهداف برنامه‌نویسی شیءگرا را تشریح کنید.**
۹. **عبارات زیر را تعریف کنید:**
   - **کلاس (Class)**
   - **شی (Object)**
   - **رفتار (Behavior)**
۱۰. **چگونه اشیاء با یکدیگر ارتباط برقرار می‌کنند؟**
۱۱. **تابع سازنده چیست؟**
۱۲. **دست یابنده چیست؟**
۱۳. **دگرگون‌کننده چیست؟**
۱٤. **this چیست؟**

تمرین‌ها

برای امروز، تمرین نوشتنی ندارید. در عوض می‌توانید اندکی قدم بزنید!

روز ۲

Object Oriented Programming

کپسوله‌سازی: بیاموزید جزییات را نزد خود نگاه دارید

درس روز اول (آشنایی با برنامه‌نویسی شیءگرا) باید علاقه شما را برانگیخته باشد و احتمالاً پرسش‌های زیادی در ذهنتان جرقه زده است.

همانطور که خود می‌توانید حدس بزنید، برنامه‌نویسی شیءگرا بسیار گسترده‌تر از چند تعریف ساده‌ای است که با هم دیدیم. هنگامی که با دید شیءگرا به طراحی و توسعهٔ نرم‌افزار می‌پردازید، نمی‌توانید بی‌مقدمه به کدنویسی مشغول شوید بلکه باید برنامه ریزی و طراحی دقیق به علاوهٔ درکی عمیق از اصول و تئوریهای برنامه‌نویسی شیءگرا داشته باشید. متأسفانه هیچ راه عملی برای تبدیل شدن به یک متخصص برنامه‌نویسی شیءگرا حتی در طی چند سال هم وجود ندارد، چه رسد به ۲۱ روز! در عوض باید قدمی واپس گذارید و از خود بپرسید: «می‌خواهم چکار کنم؟» آیا می‌خواهید در تئوری وارد شوید یا در نظر دارید به فردی عملگرا و در اصطلاح خبره در زمینهٔ برنامه‌نویسی شیءگرا مبدل شوید؟

به زودی متوجه خواهید شد که برای آموختن برنامه‌نویسی شیءگرا در حدی که بتوانید در کارتان از آن استفاده کنید، باید دیدگاه و روش عملی‌تری در برخورد با هر مسأله مورد بررسی داشته باشید.

خوشبختانه برای درک و کاربری مؤثر روش برنامه‌نویسی شیءگرا در پروژه‌های نرم‌افزاری احتیاجی‌به درجهٔ دکتری نیست. آنچه در این راه

نیازمند آن خواهید بود، ذهنی باز و شوق آموختن ـ گاهی هم بازآموزی و یا فراموش کردن آموزه‌های قبلی ـ است.

طی امروز و بقیه این هفته، نظری به تئوریهای موجود در پس زمینهٔ برنامه‌نویسی شیءگرا خواهیم انداخت. آنچه که از آنها تحت عنوان ابزارهای برنامه‌نویسی شیءگرا یاد می‌شود.

این تئوریها زمینهٔ لازم برای شروع کار برنامه‌نویسی شیءگرا را شکل خواهند داد. مهارت در کاربرد این ابزارها به سادگی و سرعت فراهم نمی‌شود، بلکه مانند هر مهارت دیگری، فقط با تمرین و کار مداوم افزایش می‌یابند.

اما بپردازیم به آنچه امروز خواهید آموخت:

- سه رکن بنیادی برنامه‌نویسی شیءگرا
- کپسوله‌سازی
- تعمیم مسأله
- *ADT*ها که بنیان کپسوله‌سازی را شکل می‌دهند
- تفاوتهای پیاده‌سازی و رابط
- اهمیت تقسیم مسئولیت
- کپسوله‌سازی چگونه اهداف برنامه‌نویسی شیءگرا را تأمین می‌کند

سه رکن بنیادی برنامه‌نویسی شیءگرا

برای حصول درک و مهارت در زمینه برنامه‌نویسی شیءگرا، در ابتدا باید پایه‌ای محکم و استوار بنا کنید تا بتوانید با تکیه بر آن دانش خود را گسترش دهید. نخست لازم است مفاهیم اولیهٔ برنامه‌نویسی شیءگرا را کاملاً بشناسید و برای درک کامل آنها تلاش کنید. تنها وقتی خواهید توانست تکنیک شیءگرا را به درستی اعمال کنید که درک کاملی از اصول و مبانی آن پیدا کرده باشید.

با این مقدمه، طبیعتاً به سه مفهومی می‌رسیم که باید در مورد زبانی که شیءگرا فرض می‌شود صادق باشند. از این سه مفهوم غالباً تحت عنوان سه رکن برنامه‌نویسی شیءگرا یاد می‌شود.

واژه جدید

سه‌رکن برنامه‌نویسی شیءگراعبارتنداز: کپسوله‌سازی(Encapsulation)، وراثت (Inheritance)و چند شکلی بودن(Polymorphism)

از آنجایی که برنامه‌نویسی شیءگرا بر اساس این سه مفهوم بنا نهاده شده، سه رکن اساسی بسیار به ستونی متشکل از چند بلوک شبیهند. یعنی کافیست بلوک پایینی را بیرون بکشید تا تمام ستون فرو بریزد. کپسوله‌سازی که امروز به آن خواهیم پرداخت، بخش بسیار مهمی از این پازل است. زیرا پایه وراثت و چند شکلی‌بودن بر آن نهاده شده است.

کپسوله‌سازی: رکن اول

کپسوله‌سازی به ما این امکان را می‌دهد که به جای برخورد با برنامه به صورت موجودیتی منفرد و یک تکه، آن را به اجزای مستقل کوچکتری تفکیک کنیم. هر جز خودبسنده است و کار خود را مستقل از سایر اجزا انجام می‌دهد. کپسوله‌سازی با مخفی کردن جزئیات داخلی(Implementation Hiding) هر جز در پس یک رابط(Interface) خارجی این استقلال را ایجاد می‌کند.

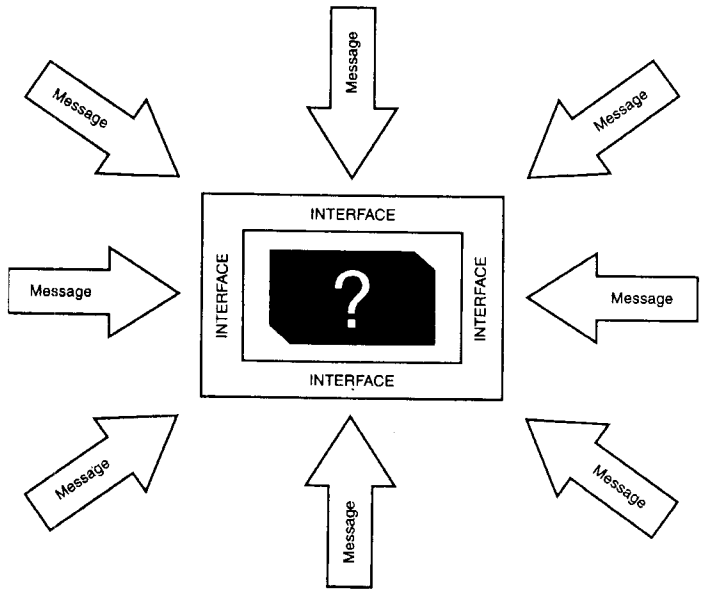
واژه جدید

کپسوله‌سازی، مشخصهٔ خودبسندگی شیءگرایی است. کپسوله‌سازی با مخفی کردن جزئیات بخشی از عملیات از جهان خارج، این اجازه را می‌دهد اجزای نرم‌افزاری مستقلی ایجاد کنیم.

نکته

اگر با اصطلاح کپسوله‌سازی آشنا نیستید، شاید اصطلاحات ماژول یا قطعه را شنیده باشید. می‌توان هریک از این کلمات و عبارات را به جای «تکهٔ کپسوله شده‌ای از نرم‌افزار» به کار برد.

وقتی کپسوله‌سازی اعمال شده باشد، می‌توان هر موجودیت نرم‌افزاری را به صورت یک جعبهٔ سیاه در نظر گرفت. هرگاه رابط خارجی جعبه را بشناسید و بدانید چه کار می‌کند، می‌دانید که جعبه چه کاری انجام می‌دهد. چنانکه شکل ۲ ـ ۱ نشان می‌دهد، برنامه‌نویس تنها با جعبهٔ سیاه سروکار دارد و به آن پیام ارسال می‌کند. آنچه درون جعبه اتفاق می‌افتد اهمیتی ندارد. آنچه مهم است وقوع اتفاق است.



شکل ۲ ـ ۱ یک جعبه سیاه

واژه جدید

یک رابط تمام سرویسهایی که توسط قطعه ارایه می‌شوند را لیست می‌کند. رابط درواقع قراردادی با دنیای خارج است که مقرر می‌دارد یک موجودیت خارجی دقیقاً چه کاری با شیء می‌تواند انجام دهد. یک رابط درواقع پانل کنترل شیء است.

نکته

اهمیت رابط در آن است که آنچه با شیء می‌توانید انجام دهید را مشخص می‌کند. اما جالب‌تر آن چیزی است که رابط نمی‌گوید. یعنی اینکه شیء چگونه کار خود را انجام می‌دهد. در عوض رابط پیاده‌سازی کد را از دید جهان خارج مخفی می‌کند. این موضوع شیء را برای تغییر کد خود هر زمان که لازم باشد آزاد می‌گذارد. تا زمانی که رابط بدون تغییر بماند، تغییرات کد باعث تغییر کدی که از کلاس استفاده می‌کند نمی‌شود. در عوض تغییر رابط، تغییر هر کدی راکه آن رابط را به کار گرفته باشد را ضروری می‌سازد.

نکته

شاید عبارت رابط برنامه‌نویسی کاربردی(Application Programming Interface) یا API برای شما آشنا باشد. یک رابط همانند API برای شیء صاحب آن عمل می‌کند. یعنی تمام متدها و آرگومانهایی که شیء درک می‌کند و می‌شناسد را لیست می‌کند.

واژه جدید

پیاده‌سازی یا شیوه کدنویسی مشخص می‌کند که یک جزء یا شیء چگونه یک سرویس را فراهم می‌کند. درواقع پیاده‌سازی جزییات داخلی قطعه را مشخص می‌کند.

مثالی از رابط و پیاده‌سازی

کلاس Log که تعریف آن در زیر آمده است را در نظر بگیرید.

```
public class Log {
    public void debug( String message ) {
        print( "DEBUG", message );
    }

    public void info( String message ) {
        print( "INFO", message );
    }

    public void warning( String message ) {
        print( "WARNING", message );
    }

    public void error( String message ) {
        print( "ERROR", message );
    }

    public void fatal( String message ) {
        print( "FATAL", message );
        System.exit( 0 );
    }

    private void print( String message, String severity ) {
        System.out.println( severity + ": " + message );
    }
}
```

کلاس Log برای اشیاء موجود در برنامه این امکان را فراهم می‌آورد که پیغامهای مختلفی را در زمان اجرای برنامه گزارش دهد. رابط کلاس Log تمام شیوهٔ رفتار کلاس با محیط خارجی را مشخص می‌کند. رفتارهایی که کلاس با دنیای خارج دارد تحت عنوان رابط عمومی(Public Interface) شناخته می‌شوند. رابط عمومی کلاس Log شامل متدهای زیر است.

```
public void warning(String message)
public void debug(String message)
public void error(String message)
public void fatal(String message)
```

هر چیز دیگری در تعریف کلاس به جز این متدها پیاده‌سازی محسوب می‌شوند. به یاد بیاورید که پیاده‌سازی مشخص می‌کند هر عملی چگونه انجام می‌شود. در اینجا «چگونه» عمل چاپی است که توسط کلاس Log صورت می‌پذیرد. ولی رابط چگونگی عملیات را کاملاً پنهان می‌کند. در عوض رابط قراردادی با دنیای خارج برای ارتباط منعقد می‌کند. مثلاً public void debug (String message) راهی است برای گفتن این حقیقت به دنیای خارج که اگر یک رشته متنی به آن ارسال کنید، تابع debug یک پیغام دیباگ گزارش خواهد کرد.

نکته‌ای که اهمیت به خاطر سپردن دارد آن چیزی است که رابط نمی‌گوید. ()debug نمی‌گوید که لزوماً پیغام را روی صفحه نمایش چاپ خواهد کرد. در عوض آنچه این تابع با پیام انجام خواهد داد به پیاده‌سازی واگذار می‌شود. پیاده‌سازی ممکن است برای نوشتن روی صفحه نمایش، یا ذخیره پیام در فایل و یا نوشتن در یک پایگاه داده انجام شده باشد.

سطح‌های عمومی، اختصاصی و محافظت شده

شاید توجه کرده باشید که رابط عمومی شامل تابع Private void print (String message, String severity)نمی‌باشد. بلکه شیء Log حق دسترسی به print را فقط برای خودش محفوظ می‌دارد.

آنچه در رابط عمومی ظاهر خواهد شد توسط چند کلمهٔ کلیدی معلوم می‌شود. هر زبان برنامه‌نویسی شیءگرا مجموعه کلمات کلیدی خاص خود را دارد. ولی از لحاظ ساختاری این کلمات کلیدی همگی اثرات مشابهی ایجاد می‌کنند.

بیشتر زبان‌های برنامه‌نویسی شیءگرا سطح دسترسی به متدها و متغیرها را در اختیار برنامه‌نویس قرار می‌دهد:

- **عمومی ـ اجازه دسترسی به آن را به تمام اشیاء دیگر می‌دهد.**
- **اختصاصی ـ دسترسی فقط به خود موجودیت صاحب روال یا خاصیت منحصر می‌شود.**
- **محافظت شده ـ دسترسی به آن را برای خود موجودیت و کلاسهای مشتق شده از آن فراهم می‌کند.**

انتخاب سطح دسترسی در طراحی بسیار اهمیت دارد. هر رفتاری که می‌خواهید دنیای خارج بتواند آن را ببیند نیاز به دسترسی عمومی دارد. هر چیزی که بخواهید آن را از دنیای خارج محفوظ نگه دارید باید دسترسی حفاظت شده یا اختصاصی داشته باشد.

چرا کپسوله‌سازی؟

اگر بتوانید با دقت از کپسوله‌سازی استفاده کنید، شیء تبدیل به موجودیتی قابل اتصال می‌شود. برای اینکه شیء دیگری بتواند از شیء مزبور استفاده کند، فقط لازم است بداند چگونه رابط عمومی آن را به کار بگیرد، چنین استقلالی سه مزیت ارزشمند دارد:

- **استقلال به آن معنی است که می‌توانید از شیء در هر جایی استفاده کنید. آن هم به صورت مکرر. وقتی بتوانید اشیاء خود را به طرز صحیح کپسوله کنید، اشیاء به هیچ برنامه خاصی مرتبط و بسته نخواهد بود. در عوض می‌توانید آنها را هر جاکه استفاده از آنها قابل توجیه باشد به کار ببرید. برای استفاده از شیء در هر جای دیگری تنها با رابط آن سروکار دارید.**

- کپسوله‌سازی این امکان را فراهم می‌کند که شیء خود را هر چقدر لازم باشد تغییر دهید. تا زمانی که رابط را تغییر ندهید، تمام تغییرات در شیء برای اشیای دیگری که از شیء استفاده می‌کنند نامرئی خواهد بود. کپسوله‌سازی به شما اجازه می‌دهد که شیء خود را به روز کنید، پیاده‌سازی آن را بهبود بخشید یا خطاهای آن را رفع کنید، بدون اینکه نیاز باشد سایر اشیای موجود در برنامه تغییر کنند. به این صورت کاربران شیء خودبه‌خود از هر تغییری که ایجاد کنید سود خواهند برد.
- استفاده از یک شیء کپسوله شده مانع تأثیرات متقابل ناخواسته بین شیء و بقیهٔ برنامه خواهد شد. از آنجایی که شیء کپسوله شده خودبسنده است، هیچگونه رابطه‌ای با بقیهٔ برنامه جز از طریق رابط خود نخواهد داشت. اکنون در نقطه‌ای هستیم که می‌توانیم چند قانون عمومی دربارهٔ کپسوله‌سازی را بیان کنیم.

دیدید که کپسوله‌سازی اجازه می‌دهد که اجزای نرم‌افزاری خودبسنده بنویسید. سه خاصیت مهم کپسوله‌سازی مؤثر عبارتند از:

- تجرید یا عمومیت دادن که پایه قابلیت استفاده مجدد است.
- مخفی کردن پیاده‌سازی یا کد برنامه که از آن تحت عنوان اختفای کد یا اختفا هم یاد خواهیم کرد.
- تقسیم مسئولیت

اجازه دهید تا نگاه دقیق‌تری به هریک از خواص فوق بیاندازیم تا بتوانیم بیاموزیم چگونه به بهترین کپسوله‌سازی دست پیدا کنیم.

تجرید: چگونه عام فکر کنیم و برنامه بنویسیم

با اینکه زبان‌های شیءگرا کپسوله‌سازی را تشویق می‌کنند هیچگونه ضمانتی برای آن نمی‌دهند. نوشتن کد وابسته و ضعیف بسیار ساده است. کپسوله‌سازی مناسب فقط با طراحی دقیق، تعمیم و تجربه حاصل می‌شود. یکی از مهمترین قدمها به سوی کپسوله‌سازی آن است که بیاموزیم چگونه نرم‌افزار و مفاهیم موجود در آن را درست عمومیت دهیم.

تجرید چیست؟

تجرید عبارت است از فرایند ساده‌سازی یک مسأله پیچیده. هنگامی که می‌خواهید مسأله‌ای را حل کنید، لزومی ندارد به تمام جزییات توجه کنید. در عوض مسأله را فقط با در نظر گرفتن جزییاتی که برای رسیدن به یک راه حل مورد نیاز هستند، تبدیل به مسأله‌ای ساده‌تر می‌کنید.

فرض کنید که می‌خواهید برنامه‌ای برای شبیه‌سازی جریان ترافیک بنویسید. می‌توان تصور کرد که چراغهای راهنما، وسایل نقلیه، شرایط راه، بزرگراهها، خیابان‌های دوطرفه و یک طرفه و وضعیت آب‌وهوا را به صورت کلاسهایی مدل کنید. چون هریک از این عوامل جریان ترافیک را تحت تأثیر قرار خواهند داد. در عوض حشرات و پرندگان وارد سیستم نخواهند شد، هرچند این عوامل در جاده‌های واقعی وجود داشته باشند. همچنین مدل مزبور انواع و مدل‌های ماشین‌ها و وسایل نقلیه را جداگانه مورد توجه قرار نخواهد داد. با این حساب دنیای واقعی را خلاصه کرده‌اید و فقط قطعاتی از واقعیت را در نظر می‌گیرید که شبیه‌سازی را تحت تأثیر قرار دهند. یک خودرو اهمیت زیادی در شبیه‌سازی دارد ولی در نظر گرفتن مدل ماشین (مثلاً یک کادیلاک) یا میزان بنزین آن در شبیه‌سازی ترافیک هیچ تأثیری ندارند.

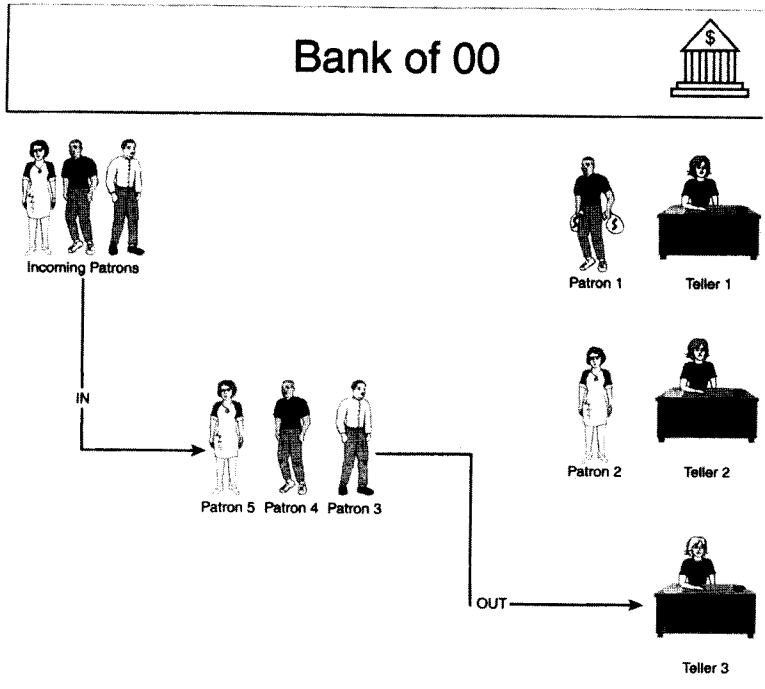
تعمیم دو مزیت مهم دارد: نخست آنکه اجازه می‌دهد یک مسأله را به آسانی حل کنید. از آن مهم‌تر خلاصه‌سازی و تجرید امکان استفاده مجدد را فراهم می‌کند. قطعات نرم‌افزاری اغلب بیش از حد خاص شده‌اند. این تخصصی شدن وقتی با وابستگی و ارتباط غیرضروری اجزاء با یکدیگر همراه شود، استفاده مجدد از یک کد در جای دیگر را مشکل خواهد کرد. تا جایی که ممکن باشد باید خود را مقید کنید تا اشیایی که می‌سازید بتوانند مجموعه‌ای از مسایل را حل کنند، نه فقط یک مسأله خاص را. تعمیم اجازه می‌دهد که یک بار مسأله‌ای را حل کنیم و سپس آن حل را برای مجموعه مسایل مشابه به کار ببریم.

نکته

هرچند نوشتن کد عام و احتراز از تخصصی کردن بیش از حد مطلوب است، اما باید به یاد داشته باشید نوشتن کد عام کار مشکلی است. خصوصاً وقتی به تازگی شروع به تمرین برنامه‌نویسی شیءگرا کرده باشید. بین افراط و تفریط در تجرید کد تفاوت ظریفی وجود دارد که تنها با تجربه قابل تشخیص است. در هر صورت، آگاهی از وجود این مفهوم مهم ضروری است.

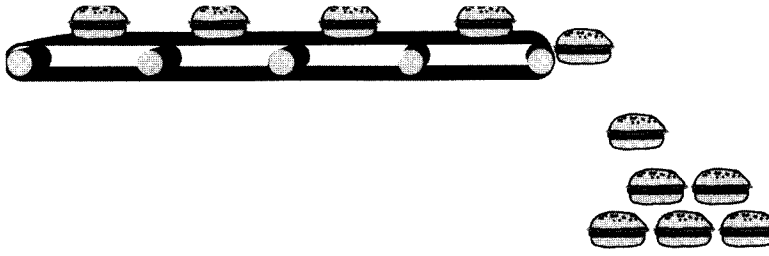
دو نمونهٔ تجرید

به عنوان اولین مثال، مردمی که در صف یک بانک منتظر پاسخگویی هستند را در نظر بگیرید. به محض اینکه یک متصدی آزاد شود، اولین نفر در صف به سوی پنجرهٔ پاسخگویی پیشروی می‌کند. افراد همواره صف را با روال ابتدا ـ وارد، ابتدا ـ خارج (FIFO - Firt In First Out) ترک می‌کنند، روالی که همواره حفظ می‌شود. (شکل ۲-۲ را نگاه کنید)



شکل ۲ـ۲ صف مراجعان به بانک

شکل ۲ـ۳
همبرگرهای خارج شده از اجاق



نمونهٔ دوم یک شرکت تولید غذای آماده است. هر ساندویچ جدیدی که از خط خارج می‌شود، بلافاصله بعد از آخرین ساندویچ در بخارپز قرار می‌گیرد. شکل ۲ ـ ۳ را ببینید. بنابراین اولین ساندویچی که از بخارپز بیرون کشیده می‌شود، قدیمی‌ترین ساندویچ است.

با وجودی که این دو مثال با هم متفاوتند و هریک مسأله‌ای خاص و جداگانه را بررسی می‌کنند، می‌توان به یک مدل کلی واحد برای هر دو حالت رسید. به عبارت دیگر می‌توان به یک مدل عام واحد دست پیدا کرد. هریک از دو مثال فوق، مثالی از صف FIFO است. درواقع اینکه چه چیزی در صف قرار می‌گیرد اصلاً اهمیت ندارد. آنچه مهم است این است که هر عنصر مانند آنچه در شکل ۲ ـ ۴ نشان داده شده است از انتها به صف وارد شده و از جلوی صف خارج می‌شود.

با تعمیم به تجرید این گروه مسایل، می‌توان یک صف ایجاد کرد و آن را در هر جایی که ترتیب عناصر FIFO را مشاهده کردید به کار ببرید.

شکل ۲ ـ ۴
تجرید دو نمونه اخیر

تجرید مؤثر

در این مرحله می‌توانیم چند قانون برای تجرید مؤثر تدوین کنیم.

- سعی کنید به روال کلی دست پیدا کنید، نه روال یک مسأله خاص
- هنگامی که با چندین مسأله مختلف مواجه شدید، به دنبال یافتن مدل کلی باشید. سعی کنید موضوع اصلی را درک کنید، نه یک مورد خاص را.
- فراموش نکنید که هدف، حل مسأله است. تعمیم عمل ارزشمندی است. ولی به امید نوشتن کد مجرد از مسأله مورد بررسی غفلت نکنید.
- شاید تعمیم دادن در ابتدا آسان نباشد. شاید مدل عام، اولین باری که مسأله را بررسی می‌کنید به فکرتان خطور نکند. حتی شاید برای بار دوم و سوم هم به جایی نرسید.
- برای روبرو شدن با اشتباهات آماده شوید. تقریباً غیرممکن است بتوان روالی نوشت که در هر موقعیتی کارایی داشته باشد. دلیل این موضوع را بعداً در درس امروز ذکر خواهیم کرد.

توجه

مراقب باشید که تعمیم شما را از هدف اصلی گمراه نکند. روی حل مسأله‌ای که با آن مواجه هستید، متمرکز شوید. تجرید را به عنوان یک مزیت اضافی تلقی کنید، نه هدف نهایی. در غیر این صورت باید با خطر از دست دادن زمان پروژه و مدل‌سازی اشتباه روبرو شوید. مدل‌سازی گاهی مفید است و گاهی هم مناسب و به جا نیست.

نکته

شاید همیشه نتوانید شانس و موقعیت عمومیت دادن را تشخیص دهید. شاید لازم باشد قبل از اینکه بتوانید مدل عام را تشخیص دهید، مسأله را چند بار حل کنید. گاهی موقعیتها و شرایط به استخراج مدل مجرد مناسب کمک می‌کنند. حتی در چنین شرایطی هم شاید احتیاج باشد که مدل قدری تصحیح شود. برای تکمیل شدن مدل عام به زمان نیاز است.

مدل عام می‌تواند قابلیت استفاده مجدد از شیء را گسترش دهد. زیرا مدلی است که برای مجموعه‌ای از مسایل مرتبط ایجاد شده، نه فقط یک مسأله خاص. اما توجه داشته باشید که مفهوم کپسوله‌سازی گسترده‌تر از تنها قابلیت استفاده مجدد از اشیاء است. اختفای جزییات درونی شیء هم از اهمیت خاصی برخوردار است. برای رسیدن به بهترین کپسوله‌سازی گام بعدی پرداختن به انواع دادهٔ مجرد (Abstract Data Type) یا ADT است.

حفظ اسرار با مخفی کردن پیاده‌سازی

تعمیم دادن تنها یکی از ویژگیهای کپسوله‌سازی کامل است. می‌توان کد تعمیم یافته‌ای نوشت که به هیچ وجه کپسوله شده نباشد. در این صورت ناگزیرید راهی برای مخفی کردن پیاده‌سازی خود پیدا کنید. اختفای پیاده‌سازی دو مزیت عمده دارد:

- شیء را در مقابل کاربران حفاظت می‌کند.
- کاربران را از درگیری زاید با شیء در امان می‌دارد.

بیایید مزیت اول را بیشتر بررسی کنیم.

محافظت از شیء توسط ADTها

انواع داده مجرد مفهوم تازه‌ای نیستند. ADTها، همزمان و در کنار خود مفهوم شیءگرا از زبان Simula که در سال ۱۹۶۶ معرفی شد به وجود آمده و توسعه یافتند. درواقع ADTها به هیچ عنوان شیءگرا نیستند. بلکه زیرمجموعهٔ روش شیءگرا هستند. ADTها دو خاصیت جالب دارند: تجرید و نوع. این شکل و ذهنیت است که اهمیت دارد، چون بدون آن، نمی‌توان کپسوله‌سازی واقعی داشت.

نکته

قابلیت کپسوله‌سازی در سطح زبان برنامه‌نویسی توسط زیرساختهای زبان فراهم می‌شود. یعنی زبان برنامه‌نویسی باید ساختار شیءگرا را پشتیبانی کند. هر نوع کپسوله‌سازی غیر از آن تنها توافقی لفظی است که به راحتی می‌تواند نادیده گرفته شود. برنامه‌نویس‌ها به راحتی این توافق را زیر پا می‌گذارند، چون می‌توانند!

واژه جدید

یک ADT عبارت است از مجموعه‌ای از داده‌ها و عملیاتی که روی داده صورت می‌پذیرد. ADTها امکان تعریف انواع جدیدی در زبان از طریق مخفی کردن داده‌های درونی و ساختار داده

با یک رابط خوب را فراهم می‌آورند. رابط مزبور ADT را به صورت یک نوع تازه و معمولی در زبان درمی‌آورد.

استفاده از ADTها راهی عالی برای معرفی کپسوله‌سازی است. چون اجازه می‌دهد کپسوله‌سازی را جدا از مفاهیم وراثت و چندشکلی بودن در نظر گرفت. لذا خواهید توانست بر کپسوله‌سازی متمرکز شوید. علاوه بر این ADTها مفهوم انواع داده را هم بهتر مشخص می‌کنند. وقتی که مفهوم نوع کاملاً درک شود، به سادگی می‌توان دید که روش شیءگرا طبیعتاً راهی برای توسعهٔ زبان از طریق تعریف انواع داده‌های جدید توسط کاربر فراهم می‌کند.

یک نوع چیست؟

هنگام برنامه‌نویسی، تعدادی متغیر تعریف می‌کنید و به آنها مقادیری نسبت می‌دهید. نوع‌ها اشکال جدیدی از مقادیر را تعریف می‌کنند که برای برنامه قابل دسترسی هستند. از این انواع می‌توان برای ساختن برنامه استفاده کرد. مقادیر صحیح، صحیح بلند و ممیز شناور نمونه هایی از انواع داده‌های عمومی هستند. این تعریف‌های نوع معلوم می‌کنند دقیقاً چه انواعی وجود دارند، این انواع چه می‌کنند و می‌شود با آنها چه کرد. ما از این پس تعریف زیر را برای نوع به کار می‌بریم.

واژه جدید

انواع اشکال مختلف مقادیری را تعریف می‌کنند که می‌توانید در برنامه خود از آنها استفاده کنید. یک نوع دامنهٔ مقادیر معتبر خود را نیز مشخص می‌کند. برای اعداد صحیح مثبت، این دامنه عبارت است از مجموعه اعدادی که جزء اعشاری ندارند و برابر یا بزرگتر از صفر هستند. برای انواع ساختاری این تعریف پیچیده‌تر است. علاوه بر دامنه مقادیر، تعریف نوع شامل عملیاتی که روی نوع مزبور معتبرند و نتایج آن عملیات هم می‌باشد.

نکته

بررسی رفتار و عملکرد نوع داده، بسیار فراتر از سطح کتابی مقدماتی در زمینه برنامه‌نویسی شیءگرا است.

انواع اجزا تشکیل دهنده هر عملیاتی هستند. این به آن معنی است که یک نوع، واحدی خودبسنده و مستقل در ساختار برنامه است. به عنوان مثال یک عدد صحیح را در نظر بگیرید. هنگامی که دو عدد صحیح را با هم جمع می‌کنیم، در مورد جمع کردن بیت‌ها فکر نمی‌کنیم، بلکه تنها جمع کردن دو عدد با هم را در نظر می‌گیریم. با وجودی که این بیت‌ها هستند که عدد صحیح را می‌سازند، زبان برنامه‌نویسی عدد صحیح را به صورت فقط یک عدد در اختیار برنامه‌نویس قرار می‌دهد.

مثال Item از روز اول را به یاد بیاورید. ایجاد کلاس Item یک نوع داده تازه به فرهنگ برنامه‌نویسی شما می‌افزاید. به جای فکر کردن به یک شناسهٔ محصول و یک شرح دربارهٔ محصول و یک قیمت به عنوان موجودیت‌های جدا و مستقل و احتمالاً با حوزه‌های تعریف متفاوت، تنها به موجودیت واحد Item فکر می‌کنید. بنابراین انواع این امکان را می‌دهند که ساختارهای پیچیده را به سطحی ساده‌تر و قابل فهم‌تر ببرید. پس انواع در مقابل جزییات زاید از برنامه‌نویس محافظت می‌کنند. این مزیت شما را آزاد می‌گذارد که بتوانید به جای اندیشیدن در سطح کدنویسی، به حل خود مسأله بیاندیشید.

انواع علاوه بر اینکه برنامه‌نویس را از درگیری با جزییات زاید محافظت می‌کنند، مزیت عمده و مهم دیگری هم دارند. تعریف یک نوع، آن نوع را از دسترسی برنامه‌نویس حفظ می‌کند. یک تعریف نوع ضمانت

می‌کند که هر شیء دیگری که با نوع مزبور در تماس است، ارتباطی صحیح، منطقی و ایمن برقرار می‌کند. محدودیتها و قید و بندهای اعمال شده توسط نوع، شیء را از ارتباطات غیرمنطقی و احتمالاً مخرب باز می‌دارد. تعریف نوع مانع استفاده نابجا و ناصحیح از شیء می‌شود. در واقع یک تعریف نوع، استفاده صحیح و مناسب را ضمانت می‌کند. بدون یک تعریف واضح و روشن از عملیات مجاز برای نوع، یک نوع می‌تواند به هر صورت ممکن با یک نوع دیگر مرتبط شود. اغلب چنین رابطه تعریف نشده‌ای می‌تواند مخرب باشد. دوباره به Item از روز اول فکر کنید. فرض کنید که ما تعریف Item را کمی تغییر داده‌ایم:

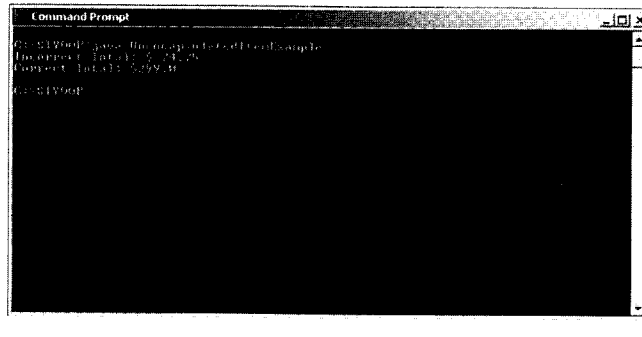
```
public class UnencapsulatedItem {
    //....
    public double unit_price;
    public double discount;    // a percentage discount to apply to the price
    public int    quantity;
    public String description;
    public String id;
}
```

می‌بینید که تمام متغیرهای داخلی کلاس اکنون به صورت عمومی تعریف شده‌اند. اگر کسی برنامه زیر را بنویسد، چه اتفاقی خواهد افتاد.

```
public static void main( String [] args ) {
    UnencapsulatedItem monitor =
            new UnencapsulatedItem( "electronics-012", "17\" SVGA Monitor", 1, 299.00 );
    monitor.discount = 1.25; // invalid, discount must be less than 100%!
    double price = monitor.getAdjustedTotal();
    System.out.println( "Incorrect Total: $" + price );
    monitor.setDiscount( 1.25 ); // invalid, however the setter will catch the error
    price = monitor.getAdjustedTotal();
    System.out.println( "Correct Total: $" + price );
}
```

شکل ۲ ـ ۵ آنچه هنگام اجرای متد ()main رخ می‌دهد را نمایش می‌دهد.

با ایجاد امکان دسترسی نامحدود و حساب نشده به شیء UnencapsulatedItem، دیگران می‌توانند شیء را در وضعیت نامعین و نادرستی قرار دهند. در این مثال تابع ()main یک شیء از نوع UnencapsulatedItem می‌سازد و سپس مستقیماً مقداری نامعتبر در discount قرار می‌دهد. در نتیجه قیمت تعدیل شدهٔ نهایی منفی شده است!



شکل ۲ ـ ۵
خروجی نامعتبر

ADTها ابزارهایی ارزشمند برای کپسوله‌سازی هستند. چون شما را قادر می‌سازند انواع داده‌های جدیدی برای زبان تعریف کنید که استفاده از آنها ایمن و بی‌خطر باشد. همانگونه که هر سال واژه‌های تازه‌ای به زبان افزوده می‌شود، یک ADT شما را قادر می‌سازد که هرگاه نیاز به بیان ایده‌ای تازه‌ای داشتید، واژه‌های برنامه‌نویسی جدیدی ایجاد کنید. به محض اینکه نوع تازه ایجاد شد، می‌توانید همانند هر نوع دیگری آن را به کار ببرید.

واژه جدید یک شیء درجه یک، شیئی است که دقیقاً مانند انواع درونی زبان به کار رود.

واژه جدید شیء درجه دو، نوعی از شیء است که می‌توانید آن را تعریف کنید، اما الزاماً همانند یک نوع درونی مورد استفاده قرار نمی‌گیرد.

یک ADT نمونه

بگذارید به مثال صف تعمیم یافته‌ای که قبلاً مطرح کردیم بازگردیم. هنگامی که یک صف را پیاده‌سازی می‌کنید، انتخاب‌های متعددی از روشهای مختلف پیش رو دارید. می‌توانید صف را به صورت لیست پیوندی، لیست پیوندی دوگانه یا آرایه پیاده کنید. به هر صورت، هر روشی که استفاده شود ساختار و رفتار صف را تغییر نمی‌دهد. صرفنظر از اینکه روش پیاده‌سازی چه باشد، عناصر صف به صورت FIFO به صف وارد و از آن خارج می‌شوند.

صف نمونه‌ای عالی برای ADT است. قبلاً دیدیم که برای استفاده از صف احتیاجی به دانستن روش پیاده‌سازی صف نداریم، درواقع نمی‌خواهیم در مورد روش پیاده‌سازی نگران باشیم. اگر صف را به یک ADT مبدل نکنیم، هر شیء که احتیاج به صف داشته باشد، نیاز به پیاده‌سازی مجدد ساختار داده دارد. هر شیء که بخواهد داده‌های موجود در صف را دستکاری کند، نیازمند خواهد بود بر روش پیاده‌سازی مسلط باشد و دقیقاً بداند چگونه باید با آن پیاده‌سازی خاص ارتباط برقرار کند وگرنه به سرنوشت مثال قبل دچار می‌شود. بنابراین بهتر است که صف را به صورت یک ADT طراحی کنید. یک ADT صف که به خوبی کپسوله شده باشد دسترسی مطمئن و صحیح به داده‌ها را تضمین می‌کند.

هنگامی که قصد دارید یک ADT طراحی کنید، باید از خود بپرسید که ADT چه می‌کند. در این نمونه، شما چه کاری می‌توانید با صف بکنید. شما می‌توانید:

- عناصری را در صف قرار دهید: enqueue
- عناصری را از صف بردارید: dequeue
- از حالت و شرایط صف پرس و جو کنید.
- به اولین عنصر صف بدون برداشتن آن دسترسی پیدا کنید: peek

هریک از عبارات فوق به یک مورد در رابط عمومی شیء صف ترجمه می‌شوند، همچنین ADT به نامی هم احتیاج دارد. در این مورد، نام ADT را Queue قرار می‌دهیم. ADT ما به صورت زیر تعریف می‌شود:

```
public interface Queue {
    public void enqueue( Object obj );
    public Object dequeue();
```

```
    public boolean isEmpty();
    public Object peek();
}
```

توجه کنید که رابط صف چیزی در مورد اینکه صف چگونه داده‌ها را نگاه می‌دارد، نمی‌گوید. همچنین دقت کنید که رابط امکان هیچگونه دسترسی غیرمجازی به هیچ یک از داده‌های درونی‌اش را نمی‌دهد. تمام این جزییات مخفی شده‌اند. اما اکنون، یک نوع داده جدید دارید. یک صف که می‌توانید از آن در هر برنامه‌ای که لازم بدانید استفاده کنید.

از آنجایی که Queue یک شیء درجه یک است، می‌توانید از آن به عنوان یک پارامتر استفاده کنید. با این مدل می‌توان به صورت یک شیء واحد رفتار کرد، چون تمام اجزاء خودبسنده هستند. این رویکرد بسیار قدرتمند و کارآمد است و به برنامه‌نویس اجازه می‌دهد منظور خود را بهتر پیاده کند. زیرا این امکان را برای او فراهم می‌کند که به جای اندیشیدن در سطح لیست‌ها و اشاره‌گرها در سطحی بالاتر یعنی در سطح و در جهت حل مسأله بیاندیشند. هنگامی که برنامه‌نویس می‌گوید صف، این کلمه حاوی تمام جزییات لیست و اشاره‌گرها است. اما علاوه بر آنها به برنامه‌نویس اجازه می‌دهد که تمام این جزییات را نادیده بگیرد و تنها به ساختار داده FIFO سطح بالا توجه کند.

نکته

همچنانکه به زودی خواهید دید یک نوع می‌تواند حاوی انواع دیگری هم باشد. این عمل علاوه بر مخفی کردن جزییات، غنای مفهومی هم ایجاد می‌کند. انواعی که شامل انواع دیگر هم هستند مفاهیم زیادی در خود دارند. مثلاً وقتی در برنامه می‌گویند int، مفهوم بسیار ساده است، شما عددی صحیح تعریف کرده‌اید. اما Queue مفهوم بیشتری دارد.

بیایید کمی بیشتر روی رابط کار کنیم و به یاد داشته باشید که این رابط بسیار کلی است. به جای اینکه بگوییم صفی از اعداد صحیح یا از همبرگر داریم، رابط فقط اشیایی را به صف می‌کند یا از صف خارج می‌کند. از آنجایی که هر زبانی برای خود روشی خاص فراهم کرده است، با تعریف این چنینی پارامترها، می‌توان هر شیء را که بخواهید وارد صف کنید. بنابراین این تعریف، نوع Queue را در موارد مختلف زیادی قابل استفاده کرده است.

توجه

رابط کلی هم دردسرهای خاص خود را دارد. مفهوم صفی از اعداد صحیح کاملاً واضح و دقیق است. واضح است که هر عنصری در Queue عددی صحیح است. اما صفی از اشیاء کمی شرایط را سخت‌تر می‌کند. هنگامی که عنصری از صف بیرون کشیده می‌شود، شاید نتوان به سادگی نوع داده آن را مشخص کرد.

کپسوله‌سازی واقعاً کارا چند خصوصیت دیگر هم دارد، که باید آنها را لحاظ کنید. ما به یک جنبهٔ قضیه یعنی مخفی کردن پیاده‌سازی توجه کرده‌ایم، اما روی دیگر سکه یعنی محافظت از کاربران چه می‌شود؟

محافظت از کاربران از طریق اختفای کد

تا به حال دیدید که رابط می‌تواند پیاده‌سازی و کد یک شیء را مخفی کند. وقتی پیاده‌سازی را در پس رابط مخفی می‌کنید، در حقیقت از شیء خود در مقابل دسترسی غیرمجاز یا مخرب محافظت کرده‌اید. محافظت

از شیء یکی از مزایای مخفی کردن پیاده‌سازی است. اما اکنون به بخش دیگر ماجرا می‌رسیم: کاربران شیء.

اختفای کد امکان طراحی انعطاف‌پذیرتری را فراهم می‌آورد. چون کاربران شما را از اجبار به هماهنگی کامل با روش پیاده‌سازی درونی شیء محافظت می‌کند. بنابراین اختفاء نه تنها شیء را محافظت می‌کند، بلکه کسانی که از شیء استفاده می‌کنند را هم با آزاد کردنشان از قید وابستگی به کد خودتان، حمایت می‌کند.

واژه جدید کد با وابستگی محدود، از روش پیاده‌سازی کد اشیاء دیگر مستقل است.

واژه جدید کد متصل یا همبسته به شدت وابسته به روش پیاده‌سازی کد اشیای دیگر است.

شاید از خود بپرسید فایده کد با وابستگی محدود چیست؟ هرگاه خصوصیتی در رابط عمومی شیء ظاهر شود، هر کسی که از خاصیت استفاده کند به وجود آن خاصیت وابسته می‌شود. اگر آن خصوصیت به طور ناگهانی حذف شود، کاربر ناگزیر است کدی که به آن رفتار یا صفت وابسته شده است را تغییر دهد.

واژه جدید کد وابسته به وجود یک نوع خاص وابسته است. وابستگی کد غیرقابل اجتناب است. اما با این وجود درجاتی برای قابل قبول بودن وابستگی وجود دارد.

وابستگی دارای درجه‌بندی خاص خود است. وابستگی را نمی‌توان کاملاً حذف کرد. اما باید برای حصول حداقل وابستگی درون شیء تلاش کرد. معمولاً چنین وابستگی با نوشتن یک رابط خوب و کامل محدود می‌شود. کاربران فقط می‌توانند به آنچه برنامه‌نویس تصمیم بگیرد در رابط قرار دهد وابسته شوند. اما اگر بخشی از پیاده‌سازی وارد رابط شود، کاربران آن شیء به آن پیاده‌سازی وابسته خواهند شد. چنین کد پیوسته و همبسته‌ای آزادی شما برای تغییر پیاده‌سازی به صورتی که صلاح می‌دانید را محدود می‌کند. چون تغییری کوچک در پیاده‌سازی شیء می‌تواند، زنجیره‌ای از تغییرات را برای تمام کاربران آن شیء ضروری کند.

توجه

کپسوله‌سازی و اختفای کد جادوگری نیستند. اگر لازم است که رابطی را تغییر دهید، مجبور هستید کد وابسته به رابط قبلی را تغییر دهید. از طریق اختفای کد و نوشتن نرم‌افزار مجهز به رابط، شما کدی ایجاد می‌کنید که وابستگی محدودی دارد.

یک مثال اختفاء از دنیای واقعی

یک مثال واقعی از مخفی کردن پیاده‌سازی این درس را به پایان می‌رساند. تعریف کلاس زیر را در نظر بگیرید.

```
public class Customer{
    //...Various customer methods
    public Item [] items; // this array holds any selected item
}
```

یک مشتری (Customer) چند مورد (Item) خرید می‌کند. در این مثال Customer آرایهٔ Itemها را جزئی از رابط بیرونی خود کرده است.

```
public static void main(String [] args){
    Customer customer=new Customer();
```

```
    // ... Select some items
    //price the items
    double total=0.0
    for (int i=0;i<customer.items.length;i++){
        Item item=customer.items[i];
        total=total+item.getAdjustedTotal();
    }
}
```

تابع ()main فوق یک مشتری جدید در نظر می‌گیرد، چند مورد خرید به او نسبت می‌دهد و سرانجام مبلغ سفارش‌ها را جمع می‌زند. همه چیز درست کار می‌کند، اما اگر بخواهید روش نگهداری خریدها توسط Customer را تغییر دهید چه اتفاقی می‌افتد؟ فرض کنید بخواهید یک کلاس Basket به برنامه وارد کنید. اگر پیاده‌سازی را تغییر دهید مجبور خواهید بود تمام کدهایی که به آرایه Itemها دسترسی دارند را تغییر دهید.

بدون مخفی کردن پیاده‌سازی، شما آزاد نخواهید بود که هرگاه لازم شد، عملکرد شیء خود را بهبود دهید. در مثال اخیر در Customer باید دسترسی به آرایهٔ عناصر Item را خصوصی کنید و دسترسی به این آرایه را از طریق توابع مناسب فراهم کنید.

نکته

مخفی کردن پیاده‌سازی معایبی هم دارد. مواردی هست که احتیاج دارید کمی بیشتر از آنچه رابط می‌گوید، بدانید. در دنیای برنامه‌نویسی شما نیازمند یک جعبهٔ سیاه هستید که با تلرانس خاصی کار کند و در عین حال دقت کافی داشته باشد. ممکن است از قبل بدانید که به یک عدد صحیح ۶۴ بیتی نیاز دارید، چون با اعداد خیلی بزرگ سروکار دارید. هنگامی که رابط خود را تعریف می‌کنید علاوه بر فراهم کردن رابط، آنچه اهمیت فراوان دارد مستندسازی این خواص پیاده‌سازی است. اما همانند بقیهٔ اجزاء رابط عمومی، هرگاه رفتاری را تعریف کردید، نمی‌توانید آن را تغییر دهید.

مخفی کردن پیاده‌سازی این امکان را برای شما فراهم می‌کند که کد مستقل و با وابستگی محدود به سایر اجزای برنامه بنویسید. چنین کدی کمتر شکننده است و انعطاف‌پذیری بیشتری برای تغییر دارد. کد انعطاف‌پذیر امکان استفاده مجدد و بهبودهای بعدی را فراهم می‌کند، چون تغییر یک بخش سیستم اجزای غیرمرتبط دیگر را تحت تأثیر قرار نمی‌دهد.

تذکر

چگونه کد با وابستگی محدود و پیاده‌سازی مخفی شده بنویسیم؟ چند نکته مهم عبارتند از:

- دسترسی به شیء را فقط به رابط مبتنی بر متد محدود کنید. چنین رابطی این اطمینان را ایجاد می‌کند که پیاده‌سازی در دسترس رها نشده است.
- نگذارید از طریق بازگرداندن سهوی اشاره‌گرها یا ارجاع‌ها، دسترسی غیرمجاز به ساختار درونی شیء فراهم شود. چنین اشتباهی کنترل شیء را به دست کاربران خواهد داد.
- هرگز در مورد ساختار درونی انواع داده‌ای که استفاده می‌کنید، حدس نزنید. به جز مواردی که رفتاری در رابط یا مستندات همراه آن ذکر شده باشد، به هیچ موردی اتکا نکنید.
- هنگام نوشتن دو نوع داده‌ای که با هم ارتباط تنگاتنگی دارند، مراقب باشید. بر پایه حدس‌ها و وابستگی‌ها برنامه ننویسید.

تقسیم مسئولیت: فقط به کار خودتان برسید!

گفتگو دربارهٔ اختفای پیاده‌سازی خود به خود به سوی بحث تقسیم مسئولیت سوق پیدا می‌کند. در بخش قبلی، دیدید که چگونه می‌توانید وابستگی و قید و بندهای کد را با اختفای کد کاهش دهید. مخفی کردن پیاده‌سازی تنها یک قدم به سوی نوشتن کد مستقل است. برای اینکه بتوانیم کد واقعاً مستقل داشته باشیم، باید محدوده مسئولیت و تقسیم وظایف صحیحی هم در برنامه داشته باشید. تقسیم مسئولیت یعنی هر شیء باید تنها یک کار را انجام دهد (وظیفه‌اش را) و آن را هم به بهترین نحو اجرا کند. علاوه بر این تقسیم مسئولیت صحیح به آن معنی است که شیء قابل استفاده به صورت موجودیتی واحد و تک کاربردی باشد. به عبارت دیگر کپسوله کردن مشتی متغیر و تابع بی‌ارتباط با یکدیگر هیچ فایده‌ای ندارد. توابع و متغیرهایی که یک شیء را تشکیل می‌دهند باید از لحاظ مفهومی، ارتباط محکمی با هم داشته باشند. تمام توابع باید با هم در جهت انجام یک مسئولیت و وظیفه واحد فعالیت کنند.

نکته

مخفی کردن پیاده‌سازی و تقسیم مسئولیت به نحوی لازم و ملزوم یکدیگر هستند. بدون اختفای کد، وظیفه شیء هم زیر سؤال می‌رود. این وظیفهٔ خود شیء است که داند چگونه کار خود را انجام دهد. اگر پیاده‌سازی را آزادانه در معرض دسترسی قرار دهید، یک کاربر ممکن است به جای استفاده از رابط مستقیماً با پیاده‌سازی کار کند. که نتیجه آن مضاعف شدن تعداد عناصر دارای مسئولیت واحد است. به محض اینکه دو شیء شروع به انجام عمل واحدی کنند، تقسیم مسئولیت به باد می‌رود! هر جا متوجه چنین چیزی شدید، لازم است کد خود را اصلاح کنید. اما احساس بدی به شما دست ندهد! ما انتظار دوباره کاری را داشتیم. دوباره کاری بخشی از چرخهٔ برنامه‌نویسی شیءگرا است. همچنانکه طرح شما رشد می‌کند و شکل می‌گیرد، فرصت‌های فراوانی برای بهبود آن خواهید داشت.

بگذارید مثالی از زندگی واقعی در مورد تقسیم مسئولیت مطرح کنیم: رابطه بین مدیر پروژه و برنامه‌نویس. فرض کنید که مدیر پروژه شما به سراغتان می‌آید، پروژه و سهم کار شما از آن را برای شما توضیح می‌دهد و سپس شما را تنها می‌گذارد که به کارتان برسید. او می‌داند که شما کاری برای انجام دادن دارید و می‌دانید چگونه کار خود را به بهترین صورت انجام دهید. حال، تصور کنید رییس شما تا این حد وارد نباشد. او پروژه و قسمتی که شما مسئول آن هستید را برای شما تشریح می‌کند. او به شما اطمینان می‌دهد که برای هر کمکی آماده است. اما هنگامی که شما شروع به کار می‌کنید، او یک صندلی کنار شما می‌گذارد و تا پایان روز کنار شما می‌نشیند و کار را قدم به قدم به شما دیکته می‌کند. هرچند در این مثال کمی غلو شده است، اما برنامه‌نویس‌ها اکثراً به همین صورت کد می‌نویسند. کپسوله‌سازی مانند یک مدیر خوب است. همانند دنیای واقعی دانش و مسئولیت باید به کسی سپرده شوند که می‌داند چگونه کار را به بهترین صورت انجام دهد. خیلی از برنامه‌نویس‌ها کد خود را به صورتی می‌نویسند که یک رئیس مداخله‌جو با زیردستانش رفتار می‌کند. این مثال به سادگی قابل ترجمه به عبارات برنامه‌نویسی است. مثال زیر را در نظر بگیرید.

```
public class BadItem {
    private double unit_price;
    private double adjusted_price;
    private double discount;    // a percentage discount to apply to the price
```

```
private int    quantity;
private String description;
private String id;

public BadItem( String id, String description, int quantity, double price ) {
    this.id = id;
    this.description = description;

    if( quantity >= 0 ) {
        this.quantity = quantity;
    }
    else {
        this.quantity = 0;
    }

    this.unit_price = price;
}

public double getUnitPrice() {
  return unit_price;
}

// applies a percentage discount to the price
public void setDiscount( double discount ) {
    if( discount <= 1.00 ) {
        this.discount = discount;
    }
}

public double getDiscount() {
    return discount;
}

public int getQuantity() {
    return quantity;
}

public void setQuantity( int quantity ) {
    this.quantity = quantity;
}

public String getProductID() {
    return id;
```

```
    }

    public String getDescription() {
        return description;
    }

    public double getAdjustedPrice() {
      return adjusted_price;
    }

    public void setAdjustedPrice( double price) {
      adjusted_price = price;
    }
}
```

واضح است که BadItem دیگر مسئول محاسبه قیمت تعدیل شده نهایی نیست. پس قیمت نهایی چگونه مشخص می‌شود؟ تابع ()main زیر را در نظر بگیرید.

```
    public static void main( String [] args ) {
        // create the items
        BadItem milk   = new BadItem( "dairy-011", "1 Gallon Milk", 2, 2.50 );

        // apply coupons
        milk.setDiscount( 0.15 );

        // get adjusted prices
        double milk_price    = milk.getQuantity() * milk.getUnitPrice();
        double milk_discount = milk.getDiscount() * milk_price;
        milk.setAdjustedPrice( milk_price - milk_discount );

        System.out.println( "Your milk costs:\t $" + milk.getAdjustedPrice() );
    }
```

در این مثال به جای اینکه به سادگی قیمت نهایی را از Item بخواهید، مجبورید مانند یک مدیر ناکارآمد عمل کنید و قدم به قدم به شیء بگویید چه کار کند.

وقتی برای محاسبه‌ی قیمت نهایی لازم باشد چندین تابع را صدا زد، مسئولیت از دست Item خارج می‌شود و در دست کاربر قرار می‌گیرد. چنین تحویل مسئولیتی به همان بدی در دسترس قرار دادن پیاده‌سازی است. باز هم با همان نتیجه مسئولیت‌های موازی در کد خود مواجه خواهید شد. هر شیء که بخواهد قیمت تعدیل شده را محاسبه کند باید همان روش ()main را تکرار کند. هنگامی که رابط شیء را می‌نویسید، باید توجه داشته باشید که اشتباهاً به صورت دیگری، با نام دیگری یا به صورت غیرمستقیم پیاده‌سازی را در دسترس کاربر قرار ندهید. به عنوان مثال برگردیم به صف، نباید متدهایی از قبیل addObjectToList() یا updateEndListPointer() را در رابط صف قرار دهید. این قبیل رفتارها مختص پیاده‌سازی هستند. در عوض باید پیاده‌سازی را از طریق معرفی متدهای سطح بالاتر enqueue() یا dequeue() مخفی کنید. حتی اگر واقعاً در پیاده‌سازی یک اشاره‌گر را به روز کنید یا شیء را به لیست

بیافزایید. وقتی از BadItem استفاده می‌کنید، نباید قبل از استفاده از ()getAdjustedPrice مجبور باشید یک تابع ()calculateAdjustedPrice را صدا بزنید. بلکه، متد ()getAdjustedprice باید خود بداند چگونه محاسبه را انجام دهد.

هنگامی که با شیئی سروکار دارید که در آن به درستی تقسیم مسئولیت نشده، در نهایت با کدی داده محور و روال‌گرا مواجه خواهید شد. تابع ()main بالا برای محاسبهٔ قیمت روال‌گرا است. یک تابع ()main که برای استفاده از صف مجبور به راهنمایی قدم به قدم ()enqueue باشد، روال‌گرا خواهد بود. اگر فقط یک پیام برای شیء بفرستید و بتوانید انجام کار را به خود شیء بسپارید، توانسته‌اید یک کار شیءگرای واقعی انجام دهید. کپسوله‌سازی کاملاً در مورد مخفی کردن جزییات است. مسئولیت، دانش هر بخش از جزییات را در جایی قرار می‌دهد که واقعاً به آن نیاز هست. برای یک شیء مهم است که یک یا تعداد معدودی مسئولیت داشته باشد. اگر مسئولیتهای شیء متعدد باشند، پیاده‌سازی آن خیلی گیج کننده خواهد شد، همچنین مدیریت و توسعهٔ آن مشکل خواهد بود. برای تغییر در یک مسئولیت، باید ریسک تغییر ناخواسته، حساب نشده و غیرقابل پیش‌بینی رفتار دیگری را بپذیرید، به خصوص اگر شیء رفتارها و خواص متعددی داشته باشد. این عمل همچنین داده‌ها و اطلاعات زیادی را یک جا انباشته می‌کند که بهتر بود پخش می‌شدند. اگر یک شیء بیش از حد بزرگ شود، خودبه‌خود تقریباً مبدل به یک برنامه می‌شود و در دام روال‌گرایی می‌افتد. در نتیجه با تمام مسایلی روبرو خواهید بود که اگر از کپسوله‌سازی استفاده نمی‌کردید با آنها روبرو بودید. یعنی کپسوله‌سازی برای برنامه‌نویس هیچ سودی نداشته است.

هرگاه متوجه شدید که شیء بیش از یک مسئولیت به عهده دارد، باید مسئولیت اضافی را به شیء مربوط به آن محول کنید.

توجه

مخفی کردن پیاده‌سازی تنها یک قدم به سوی کپسوله‌سازی مناسب است. بدون تقسیم مسئولیت صحیح، شما به سادگی با مجموعه‌ای از روال‌ها روبرو خواهید شد.

حال می‌توانیم مفهوم کپسوله‌سازی را گسترش دهیم.

واژه جدید

کپسوله‌سازی مؤثر عبارت است از تجرید مسأله به علاوهٔ مخفی کردن پیاده‌سازی به علاوهٔ تقسیم مسئولیت.

اگر تعمیم را از تعریف فوق حذف کنید، کد قابلیت استفاده مجدد نخواهد داشت. به همان صورت اگر روش پیاده‌سازی را خصوصی نگه ندارید، کد بسیار ناپایدار، وابسته و شکننده خواهد بود. تقسیم مسئولیت کد را از آفات داده محوری، روال‌گرایی، وابستگی و پراکندگی حفظ می‌کند.

اگر هریک از سه قسمت تعریف فوق غایب باشند، کپسوله‌سازی مناسبی ندارید. اما عدم وجود تقسیم مسئولیت بیشترین افتضاح را باعث می‌شود: برنامه‌نویسی روال‌گرا در محیطی شیءگرا.

نکته‌ها و دامهای کپسوله‌سازی

وقتی درصدد استفاده از روش کپسوله‌سازی هستید، نکاتی را باید مد نظر قرار دهید و از مواردی باید احتراز کنید.

نکته و دامهای تعمیم

وقتی قصد دارید که یک کلاس را بنویسید، اگر بخواهید بیش از حد عام عمل کنید، دچار دردسر خواهید

شد. نوشتن کلاسی که همهٔ کاربران را راضی کند و در هر موقعیتی قابل استفاده باشد، غیرممکن است. فرض کنید که باید برای یک سیستم حقوق و مزایای پیچیده یک شیء person (شخص) بنویسید. این شیء با یک شیء person که در برنامه شبیه‌سازی ترافیک (که قبلاً به آن اشاره کردیم) قابل استفاده باشد، تفاوت اساسی و چشم‌گیری دارد.

توجه

تعمیم می‌تواند خطرناک باشد. شیئی که خواص آن را تعمیم داده‌اید، شاید تحت تمام شرایط کار نکند. نوشتن کلاسی که تمام کاربران را راضی کند، بسیار مشکل است. پس سعی کنید در دام زیاده‌روی در تعمیم نیافتید. بلکه ابتدا روی حل مسأله‌ای که در دست دارید تمرکز کنید.

حد تعمیم کد و شیء قابلیت حل مسأله جاری توسط شیء است. در نظر گرفتن تمام جزییات و تعمیم شیء person به طوری که در هر دو مسأله مطرح شده قابل استفاده باشد بسیار گران تمام می‌شود. چنین شیئی تمام مشکلات و معضلات مطرح شده در مبحث مسئولیتهای متداخل را پیش رو دارد. هر چند می‌توانید این شیء person را در هر مسأله به کار ببرید، اما تمام مزایای ساده شدن حل را که از تعمیم حاصل می‌آیند را از دست خواهید داد.

توجه

یک کلاس را بیش از حدی که برای حل مسأله لازم است پیچیده نکنید. به دنبال حل یکباره تمام مسایل نباشید، بلکه فقط به حل مسأله جاری بپردازید. تنها پس از آن است که باید به تعمیم آنچه انجام داده‌اید بپردازید.

البته گاهی هم پیش می‌آید که خود مسأله ذاتاً پیچیده است. مثلاً یک محاسبه مشکل یا یک شبیه‌سازی پیچیده. به مسأله از نظر اهمیت مسئولیت بنگرید. هرچه وظایفی که شیء بر عهده می‌گیرد بیشتر باشند، خود شیء پیچیده‌تر شده و اداره آن مشکل‌تر خواهد شد.

نکته

به یاد داشته باشید که افزودن یک کلاس جدید به سیستم دقیقاً مانند ایجاد یک نوع داده جدید است. در این صورت بهتر می‌توانید روی کاری که می‌کنید متمرکز شوید. زیرا دربارهٔ مسأله با استفاده از مفاهیم و عبارتهای شیء و ارتباطات متقابل اشیاء فکر می‌کنید. نه داده و متد که خصوصیات دیدگاه روال‌گرا هستند.

نهایتاً توجه داشته باشید که تعمیم واقعی فقط با گذشت زمان شکل می‌گیرد. درواقع تعمیم عموماً از به کارگیری سیستم در محیط عمل به وجود می‌آید، نه توسط برنامه‌نویسی که در جای خود نشسته و تصمیم می‌گیرد که یک شیء قابل استفادهٔ مجدد بنویسد. به یاد بیاورید که احتیاج مادر اختراع است. روال در مورد اشیاء هم به همان صورت است. نمی‌توانید بدون مقدمه و ابتدا به ساکن پشت کامپیوتر خود بنشینید و یک شیء واقعاً عام و قابل استفاده مجدد بنویسید. بلکه اشیاء قابل استفاده مجدد معمولاً از شکل گرفتن و ساخت یافتن کدی استخراج می‌شوند که کارایی آن محک زده شده و تغییرات زیادی به خود دیده است.

حصول توانایی تعمیم واقعی به تجربه هم نیازمند است. رسیدن به این توانایی هدفی است که در راه استاد شدن در برنامه‌نویسی شیءگرا باید برای آن سخت بکوشید.

نکته‌ها و دامهای ADT

روش تبدیل یک ADT به کلاس، به خواص زبان برنامه‌نویسی بستگی دارد. با این وجود چند نکته مستقل از زبان برنامه‌نویسی هم وجود دارند که باید به آنها توجه کنید.

بیشتر زبان‌های شیءگرا دارای کلماتی کلیدی هستند که برای تعریف کلاسهای کپسوله شده به کار می‌روند. ابتدا خود تعریف کلاس است. کلاس تا حدی شبیه ADT است که برخی خواص ویژه آن را در روزهای آتی خواهید دید. در یک کلاس، می‌توانید متد و متغیر داخلی (داده) داشته باشید. دسترسی به این متغیرها و متدها توسط توابع دسترسی خود کلاس میسر است. تمام موارد موجود در رابط ADT باید به عنوان جزئی از رابط عمومی شیء ظاهر شوند.

توجه

ADTها مستقیماً قابل مقایسه با کلاسهای شیءگرا نیستند. ADTها هیچ کدام از خواص وراثت و چند شکلی بودن کلاسها را ندارند. اهمیت این قابلیتها در روزهای ۴ و ۶ مشخص خواهد شد.

نکاتی در مورد مخفی کردن پیاده‌سازی

تصمیم‌گیری در مورد آنچه باید در رابط کلاس ظاهر شود، همواره ساده نیست. با این وجود می‌توان به چند نکته مستقل از زبان دست یازید. فقط متدهایی باید جزء رابط عمومی محسوب شوند که تمایل دارید دیگران بتوانند از آنها استفاده کنند. متدهایی که فقط خود نوع از آن‌ها استفاده می‌کند باید مخفی باشند. در مثال صف متدهای ()enqueue و ()dequeue باید جزیی از رابط عمومی باشند. در حالی که متدهای کمکی مانند ()updateFrontPoint و ()addToList باید پنهان شوند.

نکته

تنها در صورتی می‌توانید متغیرهای داخلی را در معرض دسترسی محیط خارجی قرار دهید که زبان برنامه‌نویسی شما با این مقادیر مانند متد رفتار کند. Delphi و C++ چنین خاصیتی دارند.
اگر کاربران بتوانند بدون دانستن اینکه با یک مقدار تماس دارند، به متدها و مقادیر دسترسی داشته باشند، در این صورت رها کردن متغیرها در معرض دید موردی ندارد. در چنین زبانی، یک متغیر داخلی رو شده دقیقاً مانند متدی خواهد بود که هیچ پارامتری نمی‌پذیرد. زبان‌های برنامه‌نویسی دارای چنین خاصیتی معدودند.

همواره باید متغیرهای درونی را مخفی کنید، مگر اینکه ثابت باشند. تأکید می‌کنم که مخفی بودن کفایت نمی‌کند بلکه متغیرها باید فقط توسط خود کلاس قابل دسترسی باشند. در این مورد در روز چهارم بیشتر صحبت خواهیم کرد. قرار دادن متغیرهای داخلی در معرض دسترسی دیگران پیاده‌سازی شما را در معرض دید قرار می‌دهد.

در پایان تأکید می‌کنم، رابطی ایجاد نکنید که عملکرد متدها و متغیرهای داخلی را با نام دیگر داشته باشند. رابط شیء باید رفتاری سطح بالاتر از پیاده‌سازی داشته باشد.

چگونه کپسوله‌سازی اهداف برنامه‌نویسی شیءگرا را تأمین می‌کند

در درس روز اول تأکید کردیم که هدف برنامه‌نویسی شیءگرا تولید نرم‌افزاری است که ویژگیهای زیر را داشته باشد:

۱. طبیعی بودن
۲. قابل اعتماد
۳. قابل استفاده مجدد
٤. قابل اداره و مدیریت
۵. قابل توسعه در وقت لزوم
۶. قابل حصول در زمان قابل قبول

کپسوله‌سازی هر هدف را به شیوه‌ای تأمین می‌کند:

- طبیعی بودن: کپسوله‌سازی امکان تقسیم مسئولیت مطابق روش تفکر بشر را فراهم می‌کند. از طریق تجرید شما آزاد خواهید بود که به مسأله با دید کلی بنگرید. نه به عنوان یک مسأله خاص. تعمیم به شما اجازه می‌دهد که به صورت عام بیاندیشید و برنامه بنویسید.
- قابلیت اعتماد: با افراز مسئولیت و اختفای روش پیاده‌سازی، می‌توانید به شیء اعتبار ببخشید. در این صورت می‌توانید آن را با اطمینان به کار ببرید. این کار امکان تست فراواحدی را فراهم می‌کند. با این حال باز هم احتیاج هست که مجموعه را برای اطمینان از عملکرد صحیح تست کلی کرد.
- قابلیت استفاده مجدد: تعمیم و ایجاد مدل کلی کدی انعطاف‌پذیر و قابل استفاده مجدد به صورت مکرر در شرایط مختلف فراهم می‌کند.
- قابل اداره: کد کپسوله شده آسانتر مدیریت می‌شود. به راحتی می‌توانید هر تغییری که بخواهید در کد کلاس خود بدهید، بدون اینکه کدهای وابسته به آن دچار مشکل شوند. این تغییرات می‌توانند شامل بهبود کارایی پیاده‌سازی یا افزودن متدهای جدید به رابط باشند. تنها تغییراتی که معنا و رفتار قبلی رابط را بر هم بزنند، باعث لزوم تغییر در کد وابسته می‌شوند.
- قابل توسعه: می‌توان پیاده‌سازی را بدون ایجاد مشکل برای کدهای دیگر تغییر داد. در نتیجه می‌توانید بدون تغییر کدهای موجود کارایی شیء را بهبود بخشید یا کارکرد آن را تغییر داد. به علاوه، از آنجایی که پیاده‌سازی مخفی است، کدی که از شیء استفاده می‌کند به صورت خودکار از مزایای هر قابلیت تازه‌ای که ایجاد کنید برخوردار می‌شود. اگر چنین تغییری ایجاد کردید حتماً شیء را تست کنید. اثر تخریب یک شیء می‌تواند مانند بازی دومینو در تمام برنامه پخش شود.
- حصول در زمان قابل قبول: با شکستن برنامه به اجزای خودبسنده، می‌توانید کار توسعهٔ نرم‌افزار را بین چند برنامه‌نویس تقسیم کنید. بنابراین کار توسعه نرم‌افزار با سرعت بیشتری پیش می‌رود.

وقتی قطعات نرم‌افزاری ایجاد شدند و از طریق تست معتبر شناخته شدند، دیگر احتیاج به ساخت دوباره ندارند. لذا برنامه‌نویس آزاد است که در هر زمان همان کارکرد را در اختیار داشته باشد، بدون اینکه نیازی به کار مجدد روی آن وجود داشته باشد.

هشدار

شاید پیش خود فکر کنید: «من برای تعمیم، تجرید و کپسوله کردن کد احتیاجی به روش شیءگرا ندارم.» این درست است. شما به روش شیءگرا نیازی ندارید. ADTها خود شیءگرا نیستند. کاملاً امکان دارد که در هر زبانی کپسوله‌سازی داشت، با این حال، مشکلی وجود دارد. در انواع دیگر زبان‌ها، اغلب مجبورید خود،

مکانیزمی برای کپسوله‌سازی ایجاد کنید. از آنجایی که در زبان مزبور هیچ قید و بندی وجود ندارد که شما را وادارد، به استاندارد خود احترام بگذارید، لذا باید خیلی گوش به زنگ باشید. باید خود را مجبور کنید که در مسیر تعیین شده بمانید و از حدودی که تعیین کرده‌اید پا فراتر نگذارید. علاوه بر این مجبورید برای هر برنامه‌ای که می‌نویسید قید و بندها و مکانیزم‌ها را دوباره ایجاد کنید.

برای یک برنامه‌نویس چنین کاری خیلی سخت نیست. اما اگر دو نفر شدند چه؟ یا ۱۰ تا؟ یک گروه کامل چطور؟ همچنانکه تعداد برنامه‌نویس‌ها افزوده می‌شود، قرار دادن همه در همان مسیر و روال کار مشکل‌تری است.

یک زبان برنامه‌نویسی واقعاً شیءگرا مکانیزمی برای کپسوله‌سازی فراهم می‌کند. پس لازم نیست چنین کاری بکنید. زبان، جزییات کپسوله‌سازی را از دید کاربر کپسوله می‌کند. یک زبان شیءگرا چند کلمه کلیدی فراهم می‌کند، که برنامه‌نویس به سادگی فقط از آنها استفاده می‌کند و خود زبان بقیهٔ جزییات را بر عهده می‌گیرد.

هرگاه با خصوصیاتی که خود زبان فراهم کرده کار کنید، زبان برای تمام برنامه‌نویس‌ها همان مکانیزم ثابت را فراهم می‌کند.

خلاصه

حال که کپسوله‌سازی را درک کردید، می‌توانید برنامه‌نویسی با اشیاء را شروع کنید. با استفاده از کپسوله‌سازی می‌توانید از مزایای تعمیم و تجرید، اختفای کد و تقسیم مسئولیت در کار روزانه خود بهره بگیرید.

با تعمیم (یا به اصطلاح بعضی تجرید)، می‌توانید اشیایی ایجاد کنید که در شرایط مختلفی قابل استفاده باشند. اگر به درستی پیاده‌سازی شیء خود را پنهان کنید، آزاد خواهید بود که هرگونه بهبود و به روزآوری که می‌خواهید را هر زمان لازم باشد بر کد خود اعمال کنید. در پایان اگر بین اشیاء به درستی تقسیم مسئولیت کنید از دردسر مضاعف شدن وظایف و کد روال‌گرا پیشگیری کرده‌اید.

اگر اکنون این کتاب را زمین بگذارید و دیگر هرگز به آن رجوع نکنید، به حد کافی مهارتهای شیءگرای تازه‌ای به دست آورده‌اید که بتوانید قطعات نرم‌افزاری خودبسنده بنویسید. اما داستان برنامه‌نویسی شیءگرا به کپسوله‌سازی ختم نمی‌شود. کمی جلوتر بروید تا بیاموزید چگونه از تمام امکاناتی که برنامه‌نویسی شیءگرا در اختیار کاربر قرار می‌دهد بهره بگیرید.

پرسش‌ها و پاسخ‌ها

چطور بدانیم چه متدهایی را در رابط قرار دهیم؟

فهمیدنش بسیار ساده است. فقط متدهایی را اضافه کنید که شیء را قابل استفاده می‌کنند. متدهایی که به آنها احتیاج دارید تا از طریق آنها شیء دیگری کار خود را با شیء انجام دهد. وقتی تصمیم می‌گیرید که یک رابط بنویسید، باید کوچکترین و در عین حال کاراترین حالت را در نظر داشته باشید. تا جایی که ممکن است رابط خود را ساده کنید. متدهایی که امکان استفاده از آنها کم است را وارد رابط نکنید. هرگاه واقعاً به آنها نیاز داشتید می‌توانید آنها را اضافه کنید.

از بعضی انواع متدها هم باید احتراز کنید. از ایجاد متدی که مستقیماً به اشیاء درونی ارجاع می‌شود خودداری کنید.

مثلاً فرض کنید شما یک شیء سبد خرید دارید که موارد خرید را وارد آن می‌کنید. به این شیء نباید متدی افزود که از یکی از موارد شیء (Item) درون آن مثلاً قیمت را بپرسد. در عوض باید متدی ایجاد کنید

که Item را برگرداند و خودتان قیمت را از آن بخواهید.

قبلاً به کلمه‌های کلیدی عمومی، اختصاصی و حفاظت شده اشاره کردید. آیا هیچگونه تعیین کننده سطح دسترسی دیگری هم وجود دارد؟

هر زبانی تعیین کننده‌های خاص خود را تعریف می‌کند. با این حال بیشتر زبان‌های شیءگرا سه سطح مذکور را تعریف می‌کنند. Java هم دارای چنین چیزی برای هر بسته (Package) است. این سطح دسترسی را با تغییر دادن در تعیین کنندهٔ سطح فعال می‌کنید. این سطح، دسترسی را برای کلاسهای همان بسته محفوظ و منحصر می‌دارد.

آیا تعیین کننده‌های سطح همانند یک مکانیزم ایمنی عمل می‌کنند؟

خیر. تعیین کننده سطح دسترسی تنها دسترسی اشیاء دیگر به شیء مورد بررسی را محدود و تنظیم می‌کنند. این مکانیزم‌ها هیچ ارتباطی به امنیت کامپیوتر ندارند.

کارگاه

پرسش‌های آزمون و پاسخ‌های آنها برای درک بیشتر فراهم شده‌اند.

پرسشها

۱. **کپسوله‌سازی چگونه اهداف برنامه‌نویسی شیءگرا را تأمین می‌کند؟**
۲. **تجرید را تعریف کنید و مثالی بزنید که آن را نشان دهد.**
۳. **پیاده‌سازی را تعریف کنید.**
٤. **رابط را تعریف کنید.**
۵. **تفاوت بین پیاده‌سازی و رابط را شرح دهید.**
۶. **چرا تقسیم مسئولیت واضح برای کپسوله‌سازی صحیح لازم است؟**
۷. **نوع را تعریف کنید.**
۸. **ADT را تعریف کنید.**
۹. **چگونه به مخفی کردن پیاده‌سازی و کد غیر وابسته دست پیدا می‌کنید؟**
۱۰. **خطرات ذاتی تجرید کدامند؟**

تمرین‌ها

۱. ساختمان داده کلاسیک پشته (Stack) را در نظر بگیرید. پشته ساختار آخر ـ وارد، ابتدا ـ خارج (LIFO) است. برخلاف صف FIFO، عناصر را تنها می‌توان از یک سمت پشته به آن وارد کرد یا از آن برداشت. همانند صف، پشته اجازه می‌دهد خالی بودن آن را بررسی کنید و در غیراین‌صورت اولین عنصر رابردارید یا مقدار آن را بررسی کنید.
یک ADT برای کلاس پشته تعریف کنید.

۲. ADT تمرین نخست را در نظر گرفته و آن را پیاده‌سازی کنید وقتی طراحی تمام شد، پیاده‌سازی دیگری تعریف کنید.

۳. به تمرین‌های ۱ و ۲ نگاهی بیاندازید. آیا رابط تمرین ۱ برای هر دو پیاده‌سازی تمرین ۲ مناسب بود؟ اگر چنین بود، رابط چه مزایایی فراهم می‌کند؟ اگر نه، کمبود و نقص رابط در چه بود؟

روز ۳

Object Oriented Programming

کپسوله‌سازی: زمان نوشتن کد

دیروز، مطالبی را در مورد کپسوله‌سازی فراگرفتید. با شروع درس امروز، باید ایده مناسبی در مورد اینکه کپسوله‌سازی چیست و چگونه باید آن را بطور مؤثر به کار برد، داشته باشید. آنچه که در این مورد تاکنون انجام نداده‌اید داشتن تجربیاتی عملی در این زمینه است. حال که به تئوری مطالب مسلط شده‌اید، نیاز است که به صورت عملی نیز با این تکنیک آشنا شوید. در طی درس امروز، تعدادی کارگاه را که مطالب آن را در روز دوم فراگرفتید، پشت سر خواهید گذاشت.

آنچه امروز خواهید آموخت به شرح زیر است

- چگونه محیط Java را نصب کنید
- اصول و مبانی کلاس
- چگونه کپسوله‌سازی را پیاده کنید
- در مورد کلاسهای اولیه Java

کارگاه ۱: برپایی محیط جاوا

برای گذراندن تمامی کارگاههای این هفته و همچنین پروژه نهایی از زبان برنامه‌نویسی Java استفاده خواهید کرد. برای آنکه بتوانید با Java برنامه بنویسید، باید نسخه‌ای از Java را نصب کرده باشید.

برای این منظور نسخه ۱/۲ از کیت توسعه نرم‌افزار Java (SDK) کفایت می‌کند.

در صورتی که نسخه فوق را ندارید، بهتر است آن را از سایت http://www.javasoft.com/j2se/ دریافت کرده و نسبت به نصب آن اقدام نمایید. شرکت Sun Microsystems از سه سیستم عامل مهم یعنی Solaris، لینوکس و ویندوز پشتیبانی می‌کند. همچنین شرکت IBM اقدام به عرضه نسخه‌هایی از Java کرده است. برای این منظور به آدرس http://www.ibm.com/java/jdk/index.html مراجعه نمایید. در کنار سیستمهای عاملی که شرکت Sun Microsystems از آنها پشتیبانی می‌کند، شرکت IBM پشتیبانی از دیگر سیستمهای عامل نظیر OS/2، AS/400 و AIX را نیز انجام می‌دهد.

هر کیت توسعه همراه با دستورات نصب آن ارایه می‌شود. برای این منظور از این دستورات پیروی نمایید تا بتوانید کیت مورد نظر را نصب کنید.

همچنین می‌توانید یکی از محیطهای توسعه مجتمع (IDE) معروف نظیر Forte، JBuilder و یا Visual Age را برای Java برگزینید. مثالها و کارگاههای ارایه شده همگی در محیطهای فوق قابل اجرا هستند.

این کتاب فرض می‌کند که آشنایی مختصری با برنامه‌نویسی دارید، با این حال نیاز به درک عمیق از Java برای تکمیل کارگاههای ارایه شده، ندارید.

تعریف مسأله

برای آنکه بتوانید یک گاری را هدایت کنید، نیاز به یک اسب دارید! برای آنکه بتوانید برنامه‌ای بنویسید نیاز است که محیط توسعه Java را نصب کنید. پس از نصب مسیر کلاسها را تنظیم کرده و اولین برنامه Java را کامپایل و اجرا کنید. این نکته را در نظر داشته باشید که باید به نحوه اجرا و کامپایل برنامه‌های Java مسلط باشید.

کارگاه ۲: اصول و مبانی کلاسها

بسیار مهم است که مطالب ارایه شده در روزهای اول و دوم را برای نوشتن اولین کلاس به خاطر داشته باشید.

در روز اول مطالبی را در مورد کلاسها و اشیاء فراگرفتید. روز دوم به شما نشان داد چگونه از کپسوله‌سازی می‌توانید جهت ایجاد اشیاء خوش‌تعریف استفاده نمایید.

کتابخانه کلاسهای Java مجموعه‌ای غنی از ساختمانهای داده نظیر لیستها (Lists) و Hash Table ها می‌باشد. به عنوان مثال کلاس DoubleKey در لیست ۳ـ۱ را در نظر بگیرید.

لیست ۳-۱ کلاس DoubleKey.java

```
public class DoubleKey {

   private String key1, key2;

   // a no args constructor
   public DoubleKey() {
      key1 = "key1";
      key2 = "key2";
   }
   // a constructor with arguments
   public DoubleKey( String key1, String key2 ) {
      this.key1 = key1;
```

لیست ۳-۱ (ادامه)

```
        this.key2 = key2;
    }

    // accessor
    public String getKey1() {
        return key1;
    }

    // mutator
    public void setKey1( String key1 ) {
        this.key1 = key1;
    }

    // accessor
    public String getKey2() {
        return key2;
    }

    // mutator
    public void setKey2( String key2 ) {
        this.key2 = key2;
    }

    //equals and hashcode omitted for brevity
}
```

زمانی که شیء را در هر نوع پیاده‌سازی java.util.Map قرار می‌دهید می‌توانید هر نوع شیء دیگری را به عنوان یک کلید برای شیء اصلی تعیین کنید. بنابراین زمانی که نیاز به شیء اصلی دارد کافی است از کلید آن برای فراخوانی شیء استفاده نمایید. DoubleKey اجازه می‌دهد که برای فراخوانی شیء از دو کلید رشته‌ای به جای یک کلید استفاده نمایید.

توجه نمایید که DoubleKey دارای دو تابع سازنده است:

```
public DoubleKey() {
    key1 = "key1";
    key2 = "key2";
}

public DoubleKey( String key1, String key2 ) {
    this.key1 = key1;
    this.key2 = key2;
}
```

توابع سازنده به دو شکل آورده می‌شوند: بدون آرگومان (سازنده‌های بدون آرگومان یا noargs costructor) و آنهایی که همراه با آرگومان هستند.

واژه جدید سازنده‌های بدون آرگومان، سازنده‌هایی هستند که هیچگونه آرگومان ورودی ندارند.

توجه

عبارت Noarg Constructor عبارتی تعریف شده در Java است. معادل آن در C++ سازنده‌های پیش فرض (Default Constructor) می‌باشد.

سازنده‌های بدون آرگومان، شیء را با مقادیر پیش فرض مقداردهی می‌کنند، در حالی که سازنده‌های با آرگومان، شیء را با مقادیر داده شده، مقداردهی می‌نمایند.

تابع public doubleKey() مثالی از سازنده‌های بدون آرگومان است، در حالی که public DoubleKey(String Key1, String Key2) آرگومان می‌پذیرد.

از روز اول به خاطر دارید متدهایی چون public String getKey1() و public String getKey2() به عنوان دست یابنده شناخته می‌شوند چرا که اجازه دسترسی به متغیرهای درونی شیء را فراهم می‌نمایند.

نکته

دنیای Java دو نوع دست یابنده می‌شناسد: تنظیم کننده‌ها (Setters) و دریافت کننده‌ها (Getters). تنظیم کننده‌ها، مقدار متغیری را تغییر داده و تنظیم می‌نمایند، حال آنکه دریافت کننده‌ها اجازه می‌دهند مقدار متغیری را بخوانید.

شرکت Sun Microsystems برای انواع نامگذاریهای تنظیم کننده‌ها و دریافت کننده‌ها روشی را توسعه داده است که الگوی طراحی JavaBean نامیده می‌شود. JavaBean درواقع روشی استاندارد برای نوشتن انواع تکه‌های نرم‌افزاری یا Component است. در صورتی که Component نوشته شده توسط شما از این استاندارد پیروی کند به راحتی می‌توانید آن را به محیطهای سازگار با JavaBean اتصال دهید. اینگونه محیطها می‌توانند به صورت بصری و با استفاده از beanها، برنامه‌تان را ایجاد نمایند.

نحوه نامگذاری در Java بسیار ساده است. درواقع باید از روش زیر پیروی کنید:

```
public voi                    d set<variable Name> (<type> value)
public <type> get<VariableName> ()
```

که <type> نوع داده‌ای متغیر و <variableName> نام متغیر است.

برای مثال، شیء Person را در نظر بگیرید. یک فرد (person) دارای نام است. نام تنظیم کننده و دریافت کننده باید به صورت زیر باشد:

```
public void SetName(String name)
public String getName()
```

در آخر آنکه از متدهای public void setKey1(String Key1) و public void setKey2 (String Key2) استفاده نمایید تا بتوانید وضعیت و حالت درونی شیء را تغییر دهید.

DoubleKey روش صحیح کپسوله‌سازی را نمایش می‌دهد. با به خدمت گرفتن یک رابط با تعریف خوب و مناسب، DoubleKey پیاده‌سازی خود را از دنیای خارج پنهان کرده است. همچنین DoubleKey کلاسی کاملاً مجرد است. بدین معنا که می‌توانید از DoubleKey هر جا که نیاز به دسترسی به شیء با دو کلید رشته‌ای داشته باشید، استفاده نمایید. در ضمن DoubleKey توانسته است با فراهم کردن متدهای ضروری، به خوبی وظایف را تقسیم نماید.

تعریف مسأله

در روز دوم با بانک OO آشنا شدید. در این بانک مشتریان وارد یک صف شده و منتظر می‌شوند تا آنها را

صدا زنند. نگران نباشید. نیازی به نوشتن کلاس Queue ندارید. Java بسیاری از کدهای مربوط به ساختمانهای داده را در خود دارد. در عوض نیاز دارید برای برنامه‌تان کلاسی برای حساب بانکی بنویسید، به هر حال Java چیزهایی را هم برای برنامه‌نویس کنار گذاشته است! جدای از نوع حساب بانکی، همه آنها خواص و ویژگیهای مشترکی هم دارند. مثلاً همه حسابها شامل موجودی هستند. یا اینکه در همه حسابها می‌توان پولی را واریز کرد. پولی را برداشت کرد و یا درخواست موجودی نمود.

امروز کلاسی برای حساب بانکی خواهید نوشت. کارگاه ۲ به همراه کلاس Teller ارایه خواهد شد. کلاس Teller شامل تابع main() خواهد بود تا از آن طریق بتوانید نحوه پیاده‌سازی کلاس حساب بانکی را تست کنید.

برای این منظور قواعد زیر باید رعایت گردد:

- نام کلاس باید Account باشد.
- کلاس شامل دو تابع سازنده است:

```
public Account()
public Account (double initial_deposti)
```

سازنده بدون آرگومان موجودی اولیه را برابر 0.00 و سازنده دومی مقدار اولیه موجودی حساب را برابر initial_deposit قرار می‌دهد.

- کلاس باید شامل متدهای زیر باشد. اولین متد، جهت واریز پول به حساب مورد استفاده قرار می‌گیرد:

```
public void depostiFunds(double funds)
```

متد بعدی جهت دریافت پول از حساب به اندازه funds به کار می‌رود:

```
public double withdrawFunds(double funds)
```

در اینحالت، withdrawFunds() نباید بیش از موجودی از حساب پول بردارد. در این صورت اگر funds مقداری بیش از موجودی حساب باشد، تنها به اندازه موجودی از حساب برداشت می‌شود. خروجی تابع withdrawFunds() مقدار پولی است، که از حساب برداشت شده است.

- از سومین متد جهت گرفتن موجودی از حساب استفاده می‌کنیم:

```
public double getBalance()
```

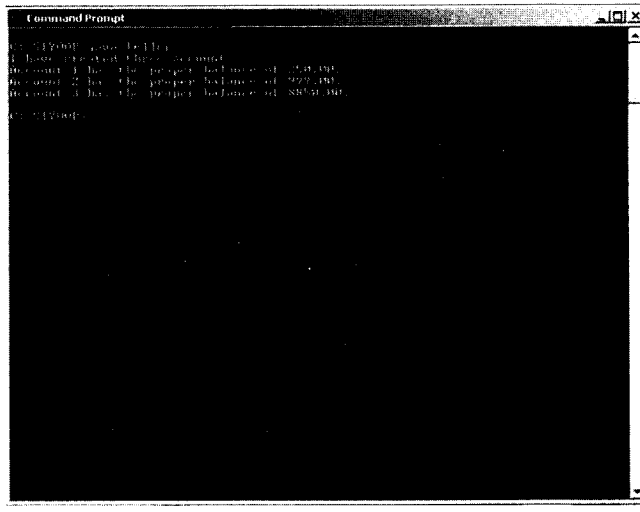
علاوه بر متدهای ذکر شده، می‌توانید دیگر متدهایی راکه فکر می‌کنید مفید فایده خواهند بود، اضافه نمایید. توجه نمایید متدهای خود را همانند آنچه در بالا گفته شده به دقت پیاده‌سازی کنید.

- زمانی که نوشتن کلاس Account به پایان رسید مطمئن شوید هر دو کلاس Account و Teller را کامپایل کرده‌اید. پس از کامپایل، با نوشتن java Teller تابع main() از کلاس Teller را صدا زنید.
- اگر همه کارها را به درستی انجام داده باشید، باید خروجی نظیر شکل ۳-۱ را مشاهده کنید.

توجه بخش بعدی جواب کارگاه ۲ است. تا زمانی که کارگاه را به پایان نرسانده‌اید، به این قسمت مراجعه نمایید!

حل و بحث

لیست ۳-۲ یکی از روشهای پیاده‌سازی کلاس Account را نشان می‌دهد.



شکل ۳ـ۱
خروجی صحیح کلاس Teller

لیست ۳-۲ Account.java

```
public class Account {

   // private data
   private double balance;

   // constructor
   public Account( double init_deposit ) {
       balance = init_deposit;
   }

   public Account() {
      // no need to do anything, _balance will default to 0
   }

   // deposit monies into account
   public void depositFunds( double amount ) {
       balance = balance + amount ;
   }
   // query the balance
   public double getBalance() {
       return balance;
   }

   // withdraw funds from the account
   public double withdrawFunds( double amount ) {

      if( amount > balance ) {  // adjust the amount
          amount = balance;
      }

      balance = balance - amount;
```

لیست ۳-۲ (ادامه)

```
        return amount;
    }

}
```

کلاس Account نمایانگر مفاهیم مهمی از کپسوله‌سازی است. درواقع Account کلاسی کاملاً مجرد است و می‌تواند به عنوان کلاسی پایه برای دیگر انواع حسابهای بانکی باشد. کلاس Account پیاده‌سازی خود را پشت رابطی خوب پنهان کرده است. در آخر آنکه این کلاس یک نگرش صحیح از مسئولیت‌پذیری را نشان داده است، چراکه همه اطلاعاتی که برای واریز یا برداشت پول نیاز است را در خود دارد. این که چگونه این اطلاعات وظایف کلاس را اداره می‌کنند، باعث نشده است که نقصی در آن به وجود آید.

با این حال کلاس Account کامل نیست. همچنان جا برای بهبودی وجود دارد. برای مثال برای همه متدها باید روتینهایی جهت چک کردن آرگومان ورودی وجود داشته باشد.

کارگاه ۳ ـ افزایش کپسوله‌سازی

روز دوم سه مشخصه کپسوله‌سازی مؤثر را بیان کرد:

- تجرید (Abstraction)
- پنهان ساختن پیاده‌سازی
- مسئولیت‌پذیری

هر مشخصه در زمان طراحی و نوشتن کلاس از اهمیت به سزایی برخوردار است. به همین جهت برای کپسوله کردن اشیاء نیاز است تا همه این مشخصه‌ها به خوبی اعمال شوند.

خوب، اجازه دهید این مشخصه‌ها را در یک بازی ورق به کار ببریم.

در ابتدا، به تجرید می‌پردازیم. به خاطر داشته باشید که نیازی به تجرید بیش از اندازه نیست. شما مسأله‌ای دارید که باید آن را حل کنید و در ضمن نمی‌توان همه مسایل را حل کرد! ابتدا باید مسأله‌ای را که می‌دانید حل نمایید!

به طور کلی در مورد بازیهای ورق که همراه با یک میز استاندارد بازی می‌شود چه می‌توان گفت؟

بهترین جای ممکن برای شروع، همان میز است. یک میز استاندارد شامل ۵۲کارت است. پس از بُر زدن می‌توان ورق را از هر جای میز برداشت. همچنین می‌توان ورق را در هر موقعیتی از میز قرار داد. باقی برداشتها می‌تواند با برداشت ورق از هر نقطه از میز به صورت ویژه صورت گیرد.

در مورد خود کارتها چه چیزی می‌توان گفت؟

همه کارتها ساختار مشترکی دارند. همه کارتها در یکی از چهار مجموعه خشت، گشنیز(خاج)، اسپیک و دل قرار می‌گیرند. هر کارت در ضمن مقداری دارد: ۲ تا ۱۰، شاه، بی بی، سرباز و آس. تفاوت هر کارت با کارت دیگر تنها در این دو خاصیت است.

چگونه می‌شود جزییات پیاده‌سازی را پنهان کرد؟ و در آخر آنکه چگونه می‌توان مسئولیت‌پذیری را نیز اضافه کرد؟

در دنیای واقعی، کارتها کار زیادی انجام نمی‌دهند. یک کارت تنها مجموعه و مقدار خود را نشان

می‌دهد. همچنین هر کارت دارای یک وضعیت است: رو به بالا یا رو به پایین. به همین صورت، میزهای بازی نیز کار خاصی انجام نمی‌دهند. درواقع این بازیگر است که همه کارها از بر زدن تا بازی کردن را انجام می‌دهد. میز تنها شامل کارتهای بازی است.

در دنیای کامپیوتر، یک کارت تنها نگهدارنده مجموعه، مقدار و وضعیت خود است. در یک برنامه ساده، کارت باید نحوه نمایش خود را نیز بداند. میز بازی باید کارتها را ساخته و در خود نگه دارد و در آخر اینکه بازیکن باید بداند چگونه کارتها را بکشد و از آنها در بازی استفاده کند.

نکته

در فصول و بخشهای بعدی اهمیت جداسازی نحوه نمایش از مدل/داده خود را فرا خواهید گرفت. برای این مسأله می‌توانید این دو را با هم ترکیب کنید.

تعریف مسأله

با استفاده از توضیحات گفته شده در بخش قبل، کلاسهایی که نمایانگر کارتها، میز کارتها و بازیکن هستند را طراحی نمایید. همچنین با نوشتن یک تابع main() کوچک، کلاسهای تعریف شده را به خدمت گرفته و کارتها را پخش کنید و وضعیت میز را بر روی صفحه نمایش چاپ کنید.

زمانی که بر روی کلاسهای طراحی شده خود فکر می‌کنید، مطمئن شوید که پنهان‌سازی پیاده‌سازی و مسئولیت‌پذیری را در نظر گرفته‌اید.

نکته

برای تولید اعداد تصادفی نگاهی به java.lang.Math.random() داشته باشید. از random() برای پخش کارتها می‌توانید استفاده کنید. متن کامل راهنمای توابع Java را می‌توانید از سایت دریافت کنید.
برای مثال (int) (Math.ramdom() * 52) عددی تصادفی بین صفر و ۵۱ تولید می‌کند.

توجه

بخش بعدی راه حل کارگاه ۳ است. بنابراین تا زمانی که مسأله را حل نکرده‌اید، به بخش بعد نروید!

حل و بحث

لیست ۳ـ۳ یکی از روشهای پیاده‌سازی کلاس Card را نشان می‌دهد.

لیست ۳-۳ Card.java

```
public class Card {
    private int rank;
    private int suit;
    private boolean face_up;

    // constants used to instantiate
    // suits
    public static final int DIAMONDS = 4;
    public static final int HEARTS   = 3;
    public static final int SPADES   = 6;
    public static final int CLUBS    = 5;
    // values
```

لیست ۳-۳ (ادامه)

```
    public static final int TWO   = 2;
    public static final int THREE = 3;
    public static final int FOUR  = 4;
    public static final int FIVE  = 5;
    public static final int SIX   = 6;
    public static final int SEVEN = 7;
    public static final int EIGHT = 8;
    public static final int NINE  = 9;
    public static final int TEN   = 10;
    public static final int JACK  = 74;
    public static final int QUEEN = 81;
    public static final int KING  = 75;
    public static final int ACE   = 65;

    // creates a new card - only use the constants to initialize
    public Card( int suit, int rank ) {
        // In a real program you would need to do validation on the arguments.

        this.suit = suit;
        this.rank  = rank;
    }

    public int getSuit() {
        return suit;
    }

    public int getRank() {
        return rank;
    }

    public void faceUp() {
        face_up = true;
    }

    public void faceDown() {
        face_up = false;
    }
    public boolean isFaceUp() {
        return face_up;
    }

    public String display() {
        String display;

        if( rank > 10 ) {
            display = String.valueOf( (char) rank );
        } else {
            display = String.valueOf( rank );
        }

        switch ( suit ) {
            case DIAMONDS:
```

لیست ۳-۳ (ادامه)

```
            return display + String.valueOf( (char) DIAMONDS );
        case HEARTS:
            return display + String.valueOf( (char) HEARTS );
        case SPADES:
            return display + String.valueOf( (char) SPADES );
        default:
            return display + String.valueOf( (char) CLUBS );
    }
  }
}
```

تعریف کلاس Card با تعریف تعدادی ثابت شروع می‌شود. این ثابتها برای شمارش مقادیر کارتها و مجموعه آنها به کار برده می‌شوند. باید به این نکته توجه کنید که زمانی که نمونه‌ای از کلاس Card را ایجاد می‌کنید دیگر نمی‌توانید، مقدار آن کارت را تغییر دهید. اشیاء از نوع کلاس Card غیرقابل تغییر هستند. با ایجاد کارتهای غیرقابل تغییر، کسی نمی‌تواند اشتباهاً مقدار ورق را تغییر دهد.

واژه جدید یک شیء غیرقابل تغییر (immutable) شیئی است که پس از ایجاد، وضعیت و حالت آن تغییر نمی‌یابد.

کلاس Card موظف است مقدار و نوع مجموعه خود را در خود نگه دارد. همچنین می‌داند چگونه رشته‌ای را که بیانگر وضعیتش است، برگرداند.

لیست ۳ـ۴ یکی از روشهای ممکن برای پیاده‌سازی Deck (میز کارتها) را نشان می‌دهد.

لیست ۳-٤ Deck.java

```
public class Deck {

    private java.util.LinkedList deck;

    public Deck() {
        buildCards();
    }

    public Card get( int index ) {
        if( index < deck.size() ) {
            return (Card) deck.get( index );
        }
        return null;
    }

    public void replace( int index, Card card ) {
        deck.set( index, card );
    }

    public int size() {
        return deck.size();
    }
```

لیست ۳-٤ (ادامه)

```
public Card removeFromFront() {
    if( deck.size() > 0 ) {
        Card card = (Card) deck.removeFirst();
        return card;
    }
    return null;
}

public void returnToBack( Card card ) {
    deck.add( card );
}

private void buildCards() {

    deck = new java.util.LinkedList();

    deck.add( new Card( Card.CLUBS, Card.TWO   ) );
    deck.add( new Card( Card.CLUBS, Card.THREE ) );
    deck.add( new Card( Card.CLUBS, Card.FOUR  ) );
    deck.add( new Card( Card.CLUBS, Card.FIVE  ) );
    deck.add( new Card( Card.CLUBS, Card.SIX   ) );
    deck.add( new Card( Card.CLUBS, Card.SEVEN ) );
    deck.add( new Card( Card.CLUBS, Card.EIGHT ) );
    deck.add( new Card( Card.CLUBS, Card.NINE  ) );
    deck.add( new Card( Card.CLUBS, Card.TEN   ) );
    deck.add( new Card( Card.CLUBS, Card.JACK  ) );
    deck.add( new Card( Card.CLUBS, Card.QUEEN ) );
    deck.add( new Card( Card.CLUBS, Card.KING  ) );
    deck.add( new Card( Card.CLUBS, Card.ACE   ) );
    deck.add( new Card( Card.SPADES, Card.TWO   ) );
    deck.add( new Card( Card.SPADES, Card.THREE ) );
    deck.add( new Card( Card.SPADES, Card.FOUR  ) );
    deck.add( new Card( Card.SPADES, Card.FIVE  ) );
    deck.add( new Card( Card.SPADES, Card.SIX   ) );
    deck.add( new Card( Card.SPADES, Card.SEVEN ) );
    deck.add( new Card( Card.SPADES, Card.EIGHT ) );
    deck.add( new Card( Card.SPADES, Card.NINE  ) );
    deck.add( new Card( Card.SPADES, Card.TEN   ) );
    deck.add( new Card( Card.SPADES, Card.JACK  ) );
    deck.add( new Card( Card.SPADES, Card.QUEEN ) );
    deck.add( new Card( Card.SPADES, Card.KING  ) );
    deck.add( new Card( Card.SPADES, Card.ACE   ) );

    deck.add( new Card( Card.HEARTS, Card.TWO   ) );
    deck.add( new Card( Card.HEARTS, Card.THREE ) );
    deck.add( new Card( Card.HEARTS, Card.FOUR  ) );
    deck.add( new Card( Card.HEARTS, Card.FIVE  ) );
    deck.add( new Card( Card.HEARTS, Card.SIX   ) );
    deck.add( new Card( Card.HEARTS, Card.SEVEN ) );
    deck.add( new Card( Card.HEARTS, Card.EIGHT ) );
```

لیست ۳-٤ (ادامه)

```
        deck.add( new Card( Card.HEARTS, Card.NINE  ) );
        deck.add( new Card( Card.HEARTS, Card.TEN   ) );
        deck.add( new Card( Card.HEARTS, Card.JACK  ) );
        deck.add( new Card( Card.HEARTS, Card.QUEEN ) );
        deck.add( new Card( Card.HEARTS, Card.KING  ) );
        deck.add( new Card( Card.HEARTS, Card.ACE   ) );

        deck.add( new Card( Card.DIAMONDS, Card.TWO   ) );
        deck.add( new Card( Card.DIAMONDS, Card.THREE ) );
        deck.add( new Card( Card.DIAMONDS, Card.FOUR  ) );
        deck.add( new Card( Card.DIAMONDS, Card.FIVE  ) );
        deck.add( new Card( Card.DIAMONDS, Card.SIX   ) );
        deck.add( new Card( Card.DIAMONDS, Card.SEVEN ) );
        deck.add( new Card( Card.DIAMONDS, Card.EIGHT ) );
        deck.add( new Card( Card.DIAMONDS, Card.NINE  ) );
        deck.add( new Card( Card.DIAMONDS, Card.TEN   ) );
        deck.add( new Card( Card.DIAMONDS, Card.JACK  ) );
        deck.add( new Card( Card.DIAMONDS, Card.QUEEN ) );
        deck.add( new Card( Card.DIAMONDS, Card.KING  ) );
        deck.add( new Card( Card.DIAMONDS, Card.ACE   ) );

    }

}
```

کلاس Deck مسئول ایجاد کارتها (از طریق کلاس Card) و فراهم آوردن امکانات دسترسی به آنها است. کلاس Deck این امر را به وسیله متدهای تعریف شده‌اش انجام می‌دهد.

لیست ۳ـ۵ پیاده‌سازی کلاس Dealer را نشان می‌دهد.

لیست ۳-۵ Dealer.java

```
public class Dealer {

    private Deck deck;

    public Dealer( Deck d ) {
        deck = d;
    }

    public void shuffle() {
        // randomize the card array
        int num_cards = deck.size();
        for( int i = 0; i < num_cards; i ++ ) {
            int index = (int) ( Math.random() * num_cards );
            Card card_i = ( Card ) deck.get( i );
            Card card_index = ( Card ) deck.get( index );
            deck.replace( i, card_index );
            deck.replace( index, card_i );
        }
```

لیست ۳-۵ (ادامه)

```
    }

    public Card dealCard() {
        if( deck.size() > 0 ) {
            return deck.removeFromFront();
        }
        return null;
    }

}
```

Dealer مسئولیت پخش کردن کارتهای بازیکن و میز را بر عهده دارد. هر سه کلاس به نحوی مسئولیت‌پذیری را بر عهده می‌گیرند. همچنین هر سه کلاس نحوه پیاده‌سازی خود را مخفی نگاه می‌دارند. هیچ چیزی بیانگر این نیست که کلاس Deck از Linked list برای پیاده‌سازی استفاده کرده است.

کلاس Card همچنین تعدادی ثابت را تعریف کرده است. کلاس Card آزاد است به هر نحو ممکن از این ثابتها استفاده نماید. همچنین آزاد است در صورت نیاز مقدار این ثابتها را تغییر دهد.

متد ()buildCards از کلاس Deck نحوه مخفی کردن جزییات پیاده‌سازی را به خوبی نشان می‌دهد. می‌توانید مجموعه کارتهای خود را با استفاده از یک حلقه for مقداردهی اولیه نمایید. اگر به ثابتها خوب دقت کرده باشید، متوجه خواهید شد از TWO تا TEN اعداد ۲ تا ۱۰ شمارش شده‌اند. استفاده از یک حلقه به مراتب ساده‌تر از مقداردهی به صورت تکی و دستی است.

به خاطر داشته باشید که برنامه نباید به مقداری که به عنوان ثابت تعریف شده است، وابسته باشد. برای فراخوانی ثوابت کافی است از نام آنها پس از نام کلاس و به همراه عملگر نقطه استفاده کنید. مثل Card.TWO و یا Card.THREE نگران مقدار هریک از ثوابت نباشید. در زمان لازم کلاس Card مقدار واقعی را جایگزین نام ثابت خواهد کرد.

در کدهای ارایه شده، تعامل بین کلاس Card و کار بر ثابتهای تعریف شده در Card از طریق نام ثابت صورت می‌گیرد. در فصل ۱۲، «الگوهای طراحی پیشرفته» روش جدیدی جهت استفاده از ثابتها ارایه می‌کنیم.

کارگاه ٤: مطالعه موردی ـ کلاسهای ابتدایی Java (اختیاری)

توجه کارگاه ۴ یک کارگاه اختیاری است. با این حال اتمام این کارگاه باعث درک بهتری از برنامه‌نویسی شیءگرا خواهد شد. اتمام این فصل برای موفقیت در روزهای بعدی الزامی نیست.

هر زبان شیءگرا دارای قوانینی است که از آن طریق می‌توان تشخیص داد چه چیزی شیء است و چه چیزی شیء نیست. تعدادی از زبان‌های شیءگرا از باقی زبان‌ها خالص‌تر هستند. زبان برنامه‌نویسی شیءگرای خالصی نظیر Smalltalk، هر چیز را به مثابه شیء (Object) در نظر می‌گیرند، حتی اپراتورها و انواع داده‌ای ساده.

نکته یک زبان برنامه‌نویسی شیءگرای خالص، همه چیز را نوعی شیء در نظر می‌گیرد.

در یک زبان شیءگرای خالص همه چیز ـ کلاسها، اپراتورها و حتی بلوک‌های کد ـ یک شیء فرض می‌شوند. Java برای خود دارای قوانینی است که از طریق آن مشخص می‌کند چه چیزی شیء است و چه چیزی شیء نیست. در Java همه چیز شیء نیست. برای مثال، زبان Java تعدادی از انواع داده‌ای ساده تعریف می‌کند. این انواع داده‌ای ابتدایی ساده در Java شیء در نظر گرفته نمی‌شوند. این انواع داده‌ای شامل float، boolean، char، byte، short، int، long و double هستند.

نکته یک زبان با قابلیت شیءگرایی همه چیز را به مثابه شیء در نظر نمی‌گیرد.

انواع داده‌ای ساده مزایایی نسبت به اشیاء در اختیار می‌گذارند. برای استفاده از انواع داده‌ای ساده نیازی به عملگر new نیست. به عنوان یک نتیجه، استفاده از انواع داده‌ای ساده در مقایسه با اشیاء روشیء بهینه‌تر است چراکه باری راکه اشیاء بر سیستم تحمیل می‌کنند، ایجاد نمی‌نمایند.

به عبارت دیگر گاهی اوقات در استفاده از انواع داده‌ای ساده با محدودیت روبرو خواهید شد. مثلاً جایی که نیازی به شیء است نمی‌توانید از این انواع داده‌ای استفاده کنید. کلاس java.util.vector را از مجموعه کلاسهای عمومی Java در نظر بگیرید. برای قرار دادن مقداری در بردار (vector) از متد add() باید به صورت زیر استفاده کنید:

```
public boolean add(object);
```

برای ذخیره مقداری در بردار، مقدار باید یک شیء باشد. درواقع اگر از انواع داده‌ای ساده بخواهید استفاده نمایید، این کار مقدور نمی‌باشد.

برای رفع این مشکل Java کلاسهایی را برای این انواع داده‌ای ساده در نظر گرفته است. Character، Byte، Double، Float، Integer، Long، Short، Boolean. این کلاسها را پوشاننده (شامل شونده ـ wrapper) می‌نامند. چراکه شامل یک مقدار و نوع داده‌ای ساده هستند.

واژه جدید یک پوشاننده، شیئی است که هدف آن تنها نگهداری شیء یا مقدار دیگری است. هر پوشاننده می‌تواند شامل چندین متد برای دریافت و انجام محاسبات بر روی مقداری که پوشیده شده است (wrapped) باشد.

برای مثال، رابط عمومی Boolean راکه در لیست ۳ـ۶ آمده است، در نظر بگیرید:

لیست ۳-۶ java.lang.Boolean

```
public final class boolean implements Serializable{
    public Boolean (boolean value);
    public Boolean (String s);

    public static final Boolean FALSE;
    public static final Boolean TRUE;
    public static final Boolean CLASS_TYPE;

    public static boolean getBoolean(String name);
    public static Boolean valueOf( String s);
```

لیست ۳-۶ (ادامه)

```
    public booleanValue();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}
```

تذکر

کلمه final به این مفهوم اشاره دارد که عنصر this هنگام مشتق شدن کلاسها از یکدیگر تغییر نکند. با وراثت در روز چهارم بیشتر آشنا خواهید شد.

پوشاننده Boolean شامل نوع داده‌ای boolean است. برای این منظور که بخواهیم مقداری boolean را وارد کلاس vector کنیم ابتدا کلاسی از نوع Boolean ایجاد کرده و سپس مقدار boolean را در آن ذخیره می‌کنیم. سپس از کلاس نهایی استفاده کرده و آن را به vector ارسال می‌کنیم.

رابط Boolean ویژگی دیگری از زبان‌های شیءگرا را معرفی می‌نماید: متغیرها و متدهای کلاس.

تاکنون همه متغیرها و متدهایی که دیده‌اید، متغیرها و متدهای نمونه ایجاد شده از آن کلاس است. این بدین معنی است که هر متد و یا هر متغیر به همان نمونه از کلاس برمی‌گردد. به عبارت دیگر برای دسترسی به متد یا متغیری نیاز به نمونه‌ای از آن کلاس است.

این واقعیت که به نمونه‌ای از کلاس یا شیء نیاز دارید، عبارتی منطقی است. متد booleanValue() از کلاس Boolean را در نظر بگیرید. متد booleanValue() درواقع متد نمونه ایجاد شده است. مقداری که این متد برمی‌گرداند بستگی به حالت درونی هرنمونه مجزا از کلاس Boolean دارد. یک نمونه می‌تواند شامل مقدار true و نمونه دیگر شامل مقدار false است. مقدار بازگشتی بستگی به مقدار درونی هر نمونه از کلاس دارد.

حال متد getBoolean() از کلاس Boolean را در نظر بگیرید. این متد کلاس است. اگر نگاهی به تعریف متد getBoolean() بیاندازید متوجه کلمه کلیدی static خواهید شد. در Java کلمه کلیدی static جهت تعریف متدها و متغیرهای کلاسی به کار می‌روند.

برخلاف متدها و متغیرهای هر نمونه از کلاس، متدها و متغیرهای کلاس وابستگی به نمونه‌های ایجاد شده ندارند. می‌توان جهت دسترسی به متدها و متغیرهای کلاسی از خود کلاس استفاده کرد. بنابراین برای فراخوانی getBoolean() نیازی به نمونه‌ای از کلاس Boolean نیست و می‌توانید به سادگی دستور Boolean.getBoolean() را به کار ببندید. جواب بازگشتی از getBoolean() وابستگی به حالت و وضعیت هیچ نمونه‌ای از این کلاس ندارد. به همین دلیل این متد را، متد کلاس می‌نامیم.

واژه جدید

متغیرهای کلاس متغیرهایی هستند که تنها به کلاس مرتبط می‌شوند و هیچ وابستگی به نمونه‌های ایجاد شده از آن کلاس ندارند. متغیرهای کلاس بین همه نمونه‌های آن کلاس به اشتراک در می‌آیند.

واژه جدید متدهای کلاس متدهایی هستند که تنها به کلاس مرتبط می‌شوند و هیچ وابستگی به نمونه‌های ایجاد شده از آن کلاس ندارند.

متغیرهای کلاس نیز به همین شیوه به کار می‌روند. درواقع نیازی به نمونه‌ای از آن کلاس نیست. با این حال کاربرد آنها در جای دیگری است. از آنجاکه متغیرها در سطح کلاس نگه داشته می‌شوند، تمام نمونه‌های آن کلاس از آن متغیر به صورت اشتراکی استفاده می‌نمایند (و اگر به صورت عمومی تعریف شده باشند، همه اشیاء می‌توانند آنها را به اشتراک بگذارند). متغیرهای کلاس نیاز به حافظه بیشتر را کم می‌کنند. عبارت public static final Boolean FALSE را در نظر بگیرید. این عبارت ثابتی را تعریف می‌کند که مقدار false را در خود دارد. از آن‌جا که به صورت ایستا (static) تعریف شده است، همه نمونه‌های ایجاد شده از کلاس Boolean این ثابت را به طور یکسان به اشتراک می‌گذارند. درواقع هر نمونه نیاز به یک کپی از این ثابت در حافظه ندارد. کلاس CountedObject را در نظر بگیرید:

```
public class CountedObject {

   private static int instances;

   /* Creates new CountedObject */
   public CountedObject() {
      instances++;
   }

   public static int getNumberInstances() {
      return instances;
   }

   public static void main( String [] args ) {
      CountedObject obj = null;
      for( int i = 0; i < 10; i++ ) {
         obj = new CountedObject();
      }
      System.out.println( "Instances created: " + obj.getNumberInstances() );
      // note that this will work too
      System.out.println( "Instances created: " + CountedObject.getNumberInstances() );
   }
}
```

CountedObject یک متغیر کلاسی به نام instances ایجاد می‌کند. همچنین از طریق متد getNumberInstances می‌توان مقدار این متغیر را دریافت کرد. درون تابع سازنده، مقدار متغیر یک واحد افزایش پیدا می‌کند. از آنجاکه همه نمونه‌های کلاس، به طور اشتراکی از این متغیر استفاده می‌نمایند، متغیر instances همچون یک شمارنده (Counter) عمل می‌کند. هر زمان که شیء جدیدی از این کلاس ایجاد می‌شود، یک واحد به این متغیر اضافه می‌شود.

تابع main()، ۱۰ نمونه از این کلاس ایجاد می‌کند. همانگونه که ملاحظه می‌نمایید برای دسترسی به مقدار متغیر کلاس می‌توان از متد getNumberInstances() و از خود کلاس استفاده نمود.

اینکه تعریف متغیر و یا متدی از نوع ایستا باشد، بسته به شرایط طراحی دارد. اگر وضعیت متغیر

و یا متدی مستقل از دیگر اجزای کلاس و یا نمونه‌های آن کلاس باشد، تعریف آن به صورت ایستا، ایده خوبی است. همچنین اگر وضعیت متغیر و یا متدی وابسته به نمونه‌های آن کلاس باشد، نمی‌توان آن را به صورت ایستا تعریف نمود. متد ()boolean Value از کلاس Boolean اینگونه است.

با مطالعه کلاس Boolean نکات مختلفی را دریافتید. برای مثال اگر کلاس Boolean با یک مقدار boolean مقداردهی شود، دیگر نمی‌توان وضعیت آن را تغییر داد! از آنجا که نمی‌توان مقدار آن را تغییر داد، نمونه‌های کلاس Boolean، غیرقابل تغییر (immutable) نامیده می‌شوند.

گاهی پیش می‌آید که اشیاء غیرقابل تغییر، مفید واقع می‌شوند. اگر با برنامه‌نویسی چند رشته‌ای (Multi Threading) آشنا باشید خواهید دید که اشیاء غیرقابل تغییر به دلیل آنکه وضعیتشان تغییر نمی‌یابد می‌توانند به راحتی در اینگونه برنامه‌ها به خدمت گرفته شوند. اصطلاحاً اینگونه اشیاء را thread safe می‌گویند.

زمانهایی نیز پیش می‌آید که اشیاء غیرقابل تغییر بیش از آنکه مفید باشند، مضر هستند. برای مثال اگر قرار باشد، برای هر نوع داده‌ای ساده یک شیء تعریف گردد، فشار ایجاد شده، بیش از اندازه سنگینی می‌نماید.

تعریف مسأله

برای کارگاه ۴ نیاز دارید که یک کلاس Boolean غیرقابل تغییر ایجاد نمایید. حداقل این کلاس نیاز به توابع set, get دارد تا مقدار را دریافت کرده و یا بتوان مقدار را از کلاس گرفت. همچنین کلاس ایجاد شده باید شامل دو تابع سازنده باشد، یک سازنده بدون آرگومان و دیگری با آرگومانی که مقدار اولیه را دریافت می‌کند.

می‌توانید به هر تعداد که نیاز باشد، متدهای دیگری نیز تعریف کنید. به هر حال فراموش نکنید که از کپسوله‌سازی به طور مؤثر استفاده نمایید.

توجه قسمت بعد، راه حل کارگاه ۴ را ارایه می‌کند. تا زمانی که کارگاه ۴ را انجام نداده‌اید، به این قسمت مراجعه نکنید!

حل و بحث

لیست ۳-۷ یک راه حل ممکن برای کارگاه ۴ را نشان می‌دهد.

لیست ۳-۷ MyBoolean.java

```
public class MyBoolean {

    // some constants for convenience
    public static final Class TYPE = Boolean.TYPE;

    private boolean value;

    // no arg constructor - default to false
    public MyBoolean() {
        value = false;
    }
```

لیست ۳-۷ (ادامه)

```
// set the initial wrapped value to value
public MyBoolean( boolean value ) {
    this.value = value;
}

public boolean booleanValue() {
    return value;
}

public void setBooleanValue( boolean value ) {
    this.value = value;
}

// for getBoolean and valueOf we can simply delegate to Boolean
// you'll learn more about delegation in chapter 4
public static boolean getBoolean( String name ) {
    return Boolean.getBoolean( name );
}

public static MyBoolean valueOf( String s ) {
    return new MyBoolean( Boolean.getBoolean( s ) );
}

// it is good practice to override equals, toString, and hashCode
// you'll learn more about overriding in chapter 4
public boolean equals( Object obj ) {
    if( obj == this ) {
        return true;
    }
    if( obj.getClass() == this.getClass() ) {
        MyBoolean bool = ( MyBoolean ) obj;
        if( bool.booleanValue() == this.booleanValue() ) {
            return true;
        }
    }

    return false;
}

public int hashCode() {
    if( value ) {
        return "true".hashCode();
    }

    return "false".hashCode();
}
public String toString() {
    if( value ) {
        return "true";
    }
```

لیست ۳-۷ (ادامه)

```
        return "false";
    }
}
```

کلاس MyBoolean همان رابط عمومی Boolean را تعریف کرده است با سه تفاوت:

- کلاس MyBoolean تابع public void setBooleanValue (boolean value) را به تعریف کلاس اضافه کرده است. از طریق این تابع می‌توان مقدار ذخیره شده در کلاس را تغییر داد.
- کلاس MyBoolean تابع valueOf() را به نحو دیگری تعریف کرده است. از این طریق می‌توان نمونه‌ای از MyBoolean را به عنوان خروجی تابع دریافت کرد.
- کلاس MyBoolean ثابتهای TRUE و FALSE را حذف کرده است. از آنجاکه کلاس فوق غیرقابل تغییر است و دو ثابت گفته شده در تعریف کلاس گنجانده نشده‌اند، مقدار آنها را می‌توان در هر لحظه‌ای از زمان تغییر داد.

پرسشها و پاسخها

آیا کپسوله‌سازی می‌تواند مضر باشد؟

این امکان وجود دارد. در نظر بگیرید که ماژولی دارید که قرار است تعدادی محاسبات ریاضی را انجام دهد. در ضمن مایلید که دقت محاسبات تا انتها یکسان بماند. متأسفانه این ماژول جهت انجام محاسبات همه عملیات را کپسوله کرده است. در صورتی که پیاده‌سازی از دقت دیگری استفاده نماید، مقدار نهایی به دست آمده اشتباه خواهد بود.

در این صورت کپسوله‌سازی می‌تواند در صورتی که شما نیاز به کنترل دقیق روند اجرای عملیات داشته باشید، مضر باشد. تنها راه مقابله در اینگونه مواقع نوشتن مستندات کافی و جامع است. به نحوی که همه قسمتهای ماژول به خوبی مستند شده و کاربر به همه اجزای آن تسلط داشته باشد.

کارگاه

سؤالات زیر برای درک و فهم بیشتر شما طرح شده است.

پرسشها

۱. **نگاهی دوباره به کلاس Account در کارگاه ۲ بیاندازید. کدام متد (یا متدها) از نوع mutator هستند؟ کدام متدها از نوع دست یابنده (Accessor) هستند؟**

۲. **دو نوع از توابع سازنده کدامند؟ از حل هریک از کارگاهها، این دو نوع را پیدا کنید.**

۳. **کلاس Boolean که در کارگاه ۴ تعریف شده است، ۳ متغیر عمومی تعریف کرده است. به نظر شما چرا این متغیرها از نوع عمومی (public) تعریف شده‌اند؟**

٤. **چگونه می‌توان راه حل بهینه‌تری برای کارگاه ۳ نوشت؟**

۵. **به نظر شما چرا راه حل کارگاه ۳ کلاس Card جداگانه‌ای برای هر مجموعه ایجاد نمی‌کند؟**

۶. در کارگاه ۳، با نحوه ایجاد مسئولیت‌پذیری آشنا شدید. چه مزایایی از ایجاد مسئولیت‌پذیری برای کلاسهای Deck, Card و Dealer نصیب ما می‌شود؟

تمرین‌ها

۱. کارگاه ۲ را در نظر بگیرید. کلاس DoubleKey را به نحو بیشتری مجرد کنید. به نحوی در DoubleKey تغییر ایجاد کنید تا هر نوع شیء را به مثابه کلید بپذیرد. درواقع تنها یک رشته را به عنوان کلید نپذیرد. برای انجام این تغییرات، باید تغییراتی در متدهای equals() و hashCode() ایجاد نمایید. کدهای مربوط به این دو متد جهت رعایت اختصار در کارگاه ۲ آورده نشده‌اند.

۲. در کارگاه ۲، کلاسهای Card و نمونه‌های آنها می‌دانند چگونه خود را نمایش دهند. این امر برای Deck صادق نیست. چرا که آنها نمی‌دانند چگونه خود را نمایش دهند. Deck را به نحوی تغییر دهید تابتواند خود را نمایش دهد.

روز ۴

Object Orinted Programming

وراثت: ساختن از هیچ

در سه روز گذشته به اولین رکن برنامه‌نویسی شیءگرا یعنی کپسوله‌سازی پرداختیم. هر چند کپسوله‌سازی مفهومی اساسی در برنامه‌نویسی شیءگراست، اما این روش برنامه‌نویسی به تعریف ADT و ماژول منحصر نیست. زیرا در این صورت مزایای چندانی در این روش نسبت به روشهای قدیمی‌تر موجود نیست.

برنامه‌نویسی شیءگرا دو رکن دیگر یعنی وراثت و چند شکلی بودن هم دارد. در دو روز آتی به وراثت خواهیم پرداخت.

آنچه امروز خواهیم آموخت به شرح زیر است:

- وراثت چیست
- انواع وراثت
- برخی مشکلات وراثت
- چند نکته دربارهٔ وراثت و اعمال آن
- چگونه وراثت اهداف برنامه‌نویسی شیءگرا را تأمین می‌کند.

وراثت چیست؟

دیروز دیدید که چگونه کپسوله‌سازی به شما امکان نوشتن اشیاء با تعریف مناسب و خودبسنده را می‌دهد. کپسوله‌سازی به شیء اجازه می‌دهد که از

شیء دیگر از طریق پیام (Message) استفاده کند. استفاده (use) تنها یکی از روشهای ارتباط اشیاء با یکدیگر در برنامه‌نویسی شیءگرا است. روش دیگر وراثت است.

وراثت در واقع امکان بنیاد نهادن یک کلاس جدید بر ساختار یک کلاس موجود است. هنگامی که یک کلاس را از دیگری مشتق می‌کنید، کلاس جدید به طور خودکار تمام صفات، خواص، رفتارها و پیاده‌سازی کلاس موجود یا والد را به ارث می‌برد.

واژه جدید وراثت مکانیزم مشتق کردن یک کلاس تازه از یک کلاس موجود است. با استفاده از وراثت کلاس جدید تمام خواص و رفتارهای کلاس قبلی را به ارث می‌برد.

کلاس زیر را در نظر بگیرید:

```
public class Employee {
    private String first_name;
    private String last_name;
    private double wage;

    public Employee( String first_name, String last_name, double wage ) {
        this.first_name = first_name;
        this.last_name  = last_name;
        this.wage = wage;
    }

    public double getWage() {
        return wage;
    }

    public String getFirstName() {
        return first_name;
    }

    public String getLastName() {
        return last_name;
    }
}
```

این کلاس می‌تواند در یک سیستم مدیریت حقوق و مزایا مورد استفاده قرار گیرد. حال فرض کنید در همین برنامه نیاز به یک کارمند شریک در سود یا کمیسیون بگیر (Commissioned Employee) پیدا کنید. یک کارمند شریک در سود علاوه بر حقوق پایه، سهمی از فروش یا سود هم دارد. به جز این خاصیت ساده CommissionedEmployee دقیقاً مانند Employee خواهد بود، چون یک کارمند شریک در سود هم در نهایت یک کارمند است.

برای نوشتن CommissionedEmployee با استفاده از کپسوله‌سازی دو راه وجود دارد. می‌توان به سادگی کد کلاس Employee را تکرار کرد و کد لازم برای افزودن حق کمیسیون به حقوق را به آن اضافه کرد. در این

صورت دو کد مشابه خواهیم داشت و در صورت لزوم کوچکترین تغییر، باید تغییر در هر دو مورد صورت پذیرد. بنابراین باید به دنبال روش مناسب‌تری باشیم. می‌توان متغیری از نوع Employee در کلاس جدید تعریف کرد و پاسخگویی به تمام پیغامهای مربوطه مانند getWage() و getFirstName() ارسال شده به CommissionedEmployee را به آن واگذار کرد.

واژه جدید واگذاری یا توکیل (Delegation) عبارت است از فرایند ارسال پیغام از یک شی به دیگری جهت درخواست انجام خواسته‌ای.

باید توجه کرد که واگذاری هم برنامه‌نویس را وادار می‌کند که تمام متدهای موجود در رابط Employee را برای پاسخگویی به پیغام‌ها، دوباره تعریف کند. پس هیچ یک از این دو راه کاملاً رضایتبخش نیستند.

بگذارید ببینیم وراثت چگونه می‌تواند این معضل را از میان بردارد:

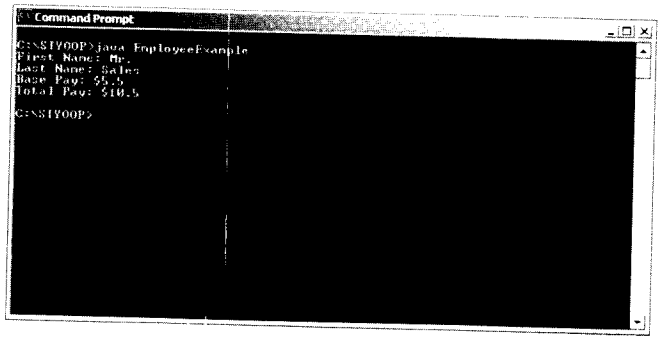
```
public class CommissionedEmployee extends Employee {

    private double commission; // the $ per unit
    private int    units;      // keep track of the # of units sold

public CommissionedEmployee( String first_name, String last_name, double wage, double
commission ) {
super( first_name, last_name, wage ); // call the original constructor in order to properly
initialize
        this.commission  = commission;
    }

    public double calculatePay() {
        return getWage() + ( commission * units );
    }

    public void addSales( int units ) {
        this.units = this.units + units;
    }

    public void resetSales() {
        units = 0;
    }
}
```

در مثال اخیر CommissiondEmployee از کلاس Employee مشتق می‌شود. لذا first_name ،last_name ،getFirstName()، getLastName() و wage همگی به آن ارث می‌رسند و جزیی از تعریف کلاس جدید خواهند بود.

بر همین اساس هر پیغامی که برای Employee ارسال می‌شود، را می‌توان مستقیماً به CommissionedEmployee ارسال کرد و پاسخ مقتضی دریافت کرد. برای مثال برنامه زیر را در نظر بگیرید که دقیقاً به همین صورت عمل می‌کند.

```
public static void main( String [] args ) {
    CommissionedEmployee c = new CommissionedEmployee("Mr.","Sales",5.50,1.00);
    c.addSales(5);
    System.out.println( "First Name: " + c.getFirstName() );
    System.out.println( "Last Name: "  + c.getLastName()  );
    System.out.println( "Base Pay: $" + c.getWage() );
    System.out.println( "Total Pay: $" + c.calculatePay() );
}
```

شکل ۴ـ۱ اجرای برنامه را نشان می‌دهد.



شکل ۴ـ۱
اجرای برنامه

چرا وراثت؟

همانطوری که در مثال اخیر دیدیم، رابطهٔ ساده استفاده که توسط کپسوله‌سازی به کار گرفته می‌شود همیشه کارایی ندارد. علاوه بر این وراثت مفهومی فراتر از به ارث بردن رابط و پیاده‌سازی دارد. امروز متوجه خواهید شد که وراثت به کلاس وارث اجازه می‌دهد هر رفتاری از کلاس والد را که مناسب نمی‌بیند دوباره برای خود تعریف کند. چنین ویژگی مفیدی به برنامه‌نویس این امکان را می‌دهد که هر جا نیازمندیها تغییر کردند، برنامهٔ خود را با آنها تطبیق دهد. یعنی کلاسی بسازد که از کلاس قبلی‌اش ارث ببرد و آنچه باید تغییر کند یا افزوده شود را جایگزین کند. ارزش جایگزینی (Override) در آن است که اجازه می‌دهد که عملکرد شیء را بدون نیاز به تغییر کلاس اصلی، تغییر دهید! لذا می‌توان کد آزموده شده و مطمئن را بدون تغییر حفظ کرد. جایگزینی حتی در مواردی که کد کلاس اصلی در دسترس نباشد کارایی دارد.

وراثت کاربرد بسیار مهم دیگری هم دارد. در درس روز اول، دیدید که چگونه یک کلاس (Class) اشیاء (Object) مرتبط با یکدیگر را گروه‌بندی می‌کند. به همان صورت وراثت هم کلاسهای مرتبط را گروه‌بندی می‌کند. روند و گرایش شیوهٔ شیءگرا رو به سوی ایجاد نرم‌افزار طبیعی است. همانند دنیای واقعی، برنامه‌نویسی شیءگرا امکان گروه‌بندی و طبقه‌بندی کلاسها را در اختیار قرار می‌دهد.

رابطهٔ همانی در مقایسه با رابطه مالکیت: چه وقتی از وراثت کمک بگیریم؟

توجه به یک نکته در این جا ضروری است و آن این است که وجود امکان ارث بردن یک کلاس از دیگری، الزامی برای استفاده از وراثت ایجاد نمی‌کند.

اما کی و کجا باید از وراثت استفاده کرد؟ خوشبختانه برای جلوگیری از وراثت نادرست قاعدهٔ ساده‌ای وجود دارد. هر بار که می‌خواهید برای استفادهٔ مجدد از قابلیتهای کد موجود یا به هر دلیل دیگری از وراثت استفاده کنید، باید از آزمون همانی استفاده کنید. یعنی از خود بپرسید: «آیا کلاس وارث از همان نوع کلاس مورث است؟»

واژه جدید همانی (Is-a) عبارت است از رابطه‌ای که در آن دو کلاس از یک نوع واحد شمرده می‌شوند.

برای استفاده از قاعدهٔ همانی برای مثال Employee می‌گوییم: CommissionedEmployee همان Employee است» عبارت اخیر صحیح است، لذا می‌توان نتیجه گرفت که وراثت در این مورد قابل استفاده است. حال این مثال را همین‌جا بگذارید و Iterator جاوا را در نظر بگیرید.

```
public interface Iterator{
    public boolean hasNext();
    public Object next();
    public void remove();
}
```

فرضاً می‌خواهید کلاسی بنویسید که رابط فوق را پیاده‌سازی کند. اگر به مطالب روز دوم رجوع کنید، شاید متوجه شوید که پیاده‌سازی Queue ممکن است به کار Iterator بیاید. یعنی می‌توانید عناصر را در Iterator نگهدارد مثلاً برای توابع hasNext(), remove() می‌توانید به سادگی متد مناسب از Queue را فراخوانی کنید و نتیجهٔ آن را برگردانید.

در مثال اخیر وراثت راه حلی مناسب برای پیاده‌سازی Iterator ارایه می‌دهد. پس قبل از شروع به کدنویسی استفاده از آزمون همانی را فراموش نکنید.

عبارت «Iterator همان Queue است» را در نظر بگیرید. واضح است که این عبارت نادرست است. بنابراین مشتق کردن Iterator از Queue را فراموش کنید.

نکته یک صف می‌تواند مالک یا حاوی یک تکرارکننده (Iterator) باشد که روی عناصر جابجا شود.

موارد بسیاری پیش می‌آید که آزمون همانی برای تشخیص روش استفاده مجدد از کد موجود پاسخگو نیست. خوشبختانه راههای دیگری هم وجود دارد. ترکیب (Composition) و ارجاع یا واگذاری (Delegation) راههای دیگری هستند که همواره می‌توان از آنها استفاده کرد. آزمون مالکیت در این موارد کارساز است.

واژه جدید مالکیت رابطه‌ای را توضیح می‌دهد که یک کلاس حاوی یک نمونه (Instance) از کلاس دیگر است.

واژه جدید ترکیب یعنی یک کلاس با استفاده از متغیرهای درونی‌ای پیاده‌سازی شود که از نوع کلاسهایی دیگر هستند.

ترکیب نوعی از استفاده مجدد است که قبلاً هم با آن روبرو شده‌ایم. اگر نمی‌توانید از وراثت استفاده

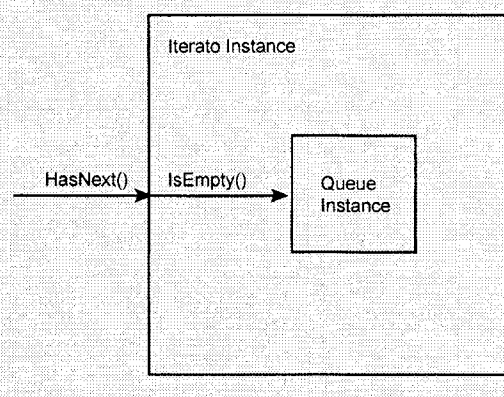
کنید لزوماً هیچ چیزی شما را از استفاده از موجودیتهایی از نوع کلاس موجود در کلاس جدید باز نمی‌دارد. البته باید محدودیتهایی که قبلاً در مورد آنها صحبت شد را نیز مدنظر داشت. بار دیگر مثال Queue/Iterator را درنظر بگیرید. در عوض مشتق شدن از Queue، کلاس Iterator می‌تواند متغیری از نوع Queue داشته باشد. هرگاه Iterator نیاز به بازیابی عنصری داشته باشد یا بخواهد خالی بودن خود را بررسی کند، می‌تواند به سادگی کار را به موجودیت Queue خود واگذار کند. (شکل ۴ـ۲ را ببینید.)

وقتی از ترکیب استفاده می‌کنید، آنچه را نیاز دارید خود برمی‌گزینید. از طریق واگذاری، ممکن است یکی یا تمام خواص شیء مورد استفاده رو شود. شکل ۴ـ۲ نشان می‌دهد که چگونه Iterator متد hasNext() خود را به متد isEmpty() از Queue ربط می‌دهد. توجه کنید که واگذاری دو تفاوت عمده با وراثت دارد:

۱. با وراثت می‌توان تنها یک نمونه شیء داشت. تنها یک شیء منفرد داریم چون آنچه ارث برده می‌شود، جزء جدا نشدنی کلاس جدید خواهد شد.

۲. واگذاری تنها رابط عمومی شیء اولیه را در اختیار کاربر قرار می‌دهد. وراثت معمولی دسترسی بیشتری به ساختارهای درونی کلاس مورث را در اختیار کاربر قرار می‌دهد. در این باره باز هم سخن خواهم گفت.

شکل ۴-۲



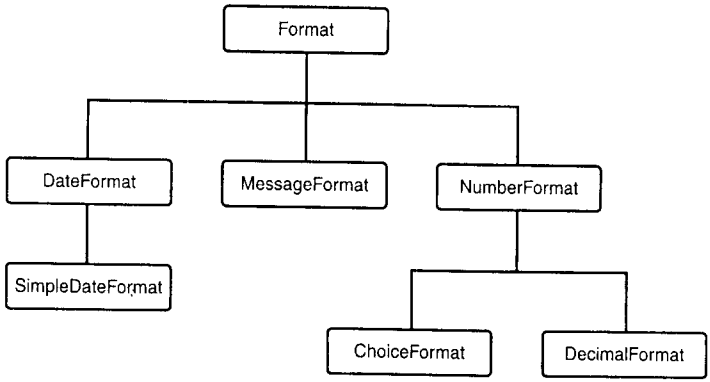
گشت و گذار در دنیای پیچیده وراثت

مفهوم همانی و نیز مفهوم ترکیب طبیعت مبحث وراثت را از کاربری مجدد کد صرف به یکی از روابط درون کلاسی تغییر می‌دهد.

کلاسی که از دیگری مشتق می‌شود باید با کلاس رابط ارتباط مفهومی داشته باشد تا وراثت معنی پیدا کند.

تعریف یک سلسله مراتب وراثتی عبارت است از ساختار درختی شکل روابطی که بین کلاسهایی که در وراثت سهیم هستند به وجود می‌آید.

شکل ۴ـ۳ یک سلسله مراتب واقعی برگرفته از Java را نشان می‌دهد.



شکل ۴ـ۳
یک سلسله مراتب نمونه برگرفته
از java.text

وراثت کلاسی جدید (فرزند یا Child) را بر اساس کلاسی موجود تعریف می‌کند، کلاس والد (Parent) این رابطه ساده‌ترین نوع وراثت است. درواقع تمام سلسله مراتب وراثت با یک والد و یک فرزند آغاز می‌شود.

واژه جدید کلاس فرزند، کلاسی است که ارث می‌برد. گاهی به نام زیرکلاس (Subclass) هم خوانده می‌شود.

واژه جدید والد، کلاسی است که کلاس فرزند مستقیماً از آن ارث می‌برد یا مشتق می‌شود. کلاس والد در بعضی موارد، فوق کلاس (Superclass) هم نامیده می‌شود.

شکل ۴ ـ ۴ یک رابطهٔ والد-فرزند نمونه را نشان می‌دهد. NumberFormat والد دو کلاس فرزند ChoiceFormat و DecimalFormat است.

اکنون که با چند تعریف تازه برخورد کرده‌اید، می‌توانید تعریف وراثت را تکمیل کنید.

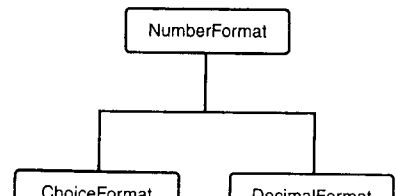
واژه جدید وراثت فرایندی است که امکان برقراری رابطه همانی بین کلاسها را ایجاد می‌کند.

چنین رابطه‌ای همچنین این امکان را ایجاد می‌کند که یک زیر کلاس بتواند صفات و رفتارهای یک کلاس والد را به ارث برد.

نکته

وقتی یک کلاس از یک کلاس والد مشتق می‌شود، کلاس فرزند تمام صفات و رفتارهایی که ممکن است کلاس والد از کلاسهای دیگر به ارث برده باشد را خودبه‌خود دریافت می‌کند.

چنانکه دیدید برای اینکه سلسله مراتب وراثت معنی‌دار باشد، باید بتوان هر عملی که با والد ممکن است را به همان صورت با کلاس فرزند انجام داد. این چیزی است که آزمون همانی درواقع تست می‌کند. کلاس مشتق شده فقط می‌تواند طرز کار را تغییر دهد و یا کارایی‌های جدید ایجاد کند، اما هیچگاه قادر نخواهد بود کارایی را کم کند.



شکل ۴ـ۴
یک والد با چند فرزند

توجه

اگر به موردی برخوردید که کلاس مشتق شده نیاز به حذف یکی از رفتارها یا کارایی‌های والد داشت، باید برداشت کنید که این کلاس باید قبل از کلاس والد در سلسله مراتب وراثت ظاهر می‌شد! همانند والدین و فرزندان واقعی، فرزندان کلاس و والدهای کلاس با هم مشابهند. کلاسها به جای اشتراک در ژن‌ها در انواع (Type) مشترک هستند.

نکته

برخلاف زندگی واقعی کلاسها می‌توانند تنها از یک والد مشتق شوند. بعضی زبان‌ها اجازه می‌دهند کلاسی از چند کلاس مشتق شود. این عمل به نام وراثت چندگانه (Mutiple-Inheriterence) خوانده می‌شود.

برخی زبان‌ها مانند Java تنها یک والد برای پیاده‌سازی می‌پذیرند، اما مکانیزمی هم برای ارث بردن از رابط (Interface) چند کلاس دارند. چنین وراثتی به بدنهٔ روال‌ها نمی‌رسد، تنها شکل تعریف آنها به ارث برده می‌شود.

مانند فرزندان واقعی، کلاسهای فرزند گاهی رفتارها و صفات جدیدی را به خود می‌افزایند. مثلاً یک فرزند انسان می‌تواند نواختن پیانو را در شرایطی بیاموزد که والدینش چنین توانایی نداشته باشند. به همان صورت یک فرزند می‌تواند در رفتار خاصی تغییر ایجاد کند.

مکانیزمهای وراثت

وقتی کلاسی از دیگری ارث می‌برد، تمام روالها، صفات و رفتارهای کلاس والد در رابط کلاس فرزند ظاهر خواهند شد. کلاسی که از راه وراثت ایجاد شود سه نوع روال یا خصوصیت می‌تواند داشته باشد:

- جایگزین شده: کلاس جدید روال‌ها یا خواص را ارث می‌برد، اما در آنها تغییر ایجاد می‌کند.
- جدید: کلاس جدید روال یا صفت جدیدی به خود می‌افزاید.
- بازگشتی (Recursive): کلاس جدید، روال یا صفت را به همان صورتی که ارث می‌برد حفظ می‌کند.

توجه

اکثر زبان‌های برنامه‌نویسی اجازهٔ جایگزینی صفات یا خواص را نمی‌دهند. با این وجود، برای کامل بودن مبحث از صفات جایگزین هم ذکری به میان آمده است.

نخست بگذارید یک مثال بزنیم. سپس هریک از انواع روال یا خاصه را بررسی خواهیم کرد.

```
public class TwoDimensionalPoint {

    private double x_coord;
    private double y_coord;

    public TwoDimensionalPoint( double x, double y ) {
        setXCoordinate( x );
        setYCoordinate( y );
    }

    public double getXCoordinate() {
        return x_coord;
```

```
    }
    public void setXCoordinate( double x ) {
        x_coord = x;
    }

    public double getYCoordinate() {
        return y_coord;
    }

    public void setYCoordinate( double y ) {
        y_coord = y;
    }

    public String toString() {
        return "I am a 2 dimensional point.\n" +
            "My x coordinate is: " + getXCoordinate() + "\n" +
            "My y coordinate is: " + getYCoordinate();
    }
}


public class ThreeDimensionalPoint extends TwoDimensionalPoint {

    private double z_coord;

    public ThreeDimensionalPoint( double x, double y, double z ) {
        super( x, y ); // initialize the inherited attributes by calling the parent constructor
        setZCoordinate( z );
    }

    public double getZCoordinate() {
        return z_coord;
    }

    public void setZCoordinate( double z ) {
        z_coord = z;
    }

    public String toString() {
        return "I am a 3 dimensional point.\n" +
            "My x coordinate is: " + getXCoordinate() + "\n" +
            "My y coordinate is: " + getYCoordinate() + "\n" +
            "My z coordinate is: " + getZCoordinate();
    }

}
```

در اینجا دو کلاس نقطه داریم که برای حفظ و نمایش نقاط دو و سه بعدی هندسی به کار می‌آیند. برای مثال

در یک برنامهٔ گرافیکی، یا مدلسازی تصویری یا برنامه‌ریز پرواز. نقاط کاربردهای عملی بسیاری دارند. در اینجا TwoDimentionalPoint مختصات x و y را نگه می‌دارد. کلاس روالهایی برای تنظیم (set) یا برگرداندن (get) مختصات نقطه، همچنین نمایش رشته‌ای آن فراهم آورده است.

ThreeDimentionalPoint از TwoDimentionalPoint مشتق می‌شود و مختص z و روالهای تنظیم و بازگرداندن مقدار آن را به کلاس می‌افزاید. این کلاس هم روالی برای نمایش رشته‌ای مقادیر مختصات دارد. این مثال انواع مختلف روال را در کلاس فرزند نشان می‌دهد.

صفات و روالهای جایگزین شده

وراثت امکان تغییر در یک روال یا صفت که از قبل موجود بوده را در اختیار برنامه‌نویس قرار می‌دهد. این تغییر که با تعریف دوبارهٔ روال یا صفت در کلاس فرزند صورت می‌گیرد رفتار شیء را تغییر می‌دهد. روال یا متد جایگزین شده هم در والد و هم در فرزند وجود دارد. مثلاً ThreeDimentionalPoint روال toString() والد خود را تغییر می‌دهد تا بتواند به جای دو، سه مختصه موقعیت نقطه را به صورت رشتهٔ متنی بازگرداند.

تابع main() زیر را در نظر بگیرید.

```
public static void main(String [] args){
    TwoDimentionalPoint two=new TwoDimentionalPoint(1,2);
    ThreeDimentionalPoint three=new ThreeDimentionalPoint(1,2,3);
    System.out.println(two.toString());
    System.out.println(three.tostring());
}
```

همچنانکه از شکل ۴ ـ ۵ قابل مشاهده است، ThreeDimentionalPoint رشته مربوط به تابع نمایش رشتهٔ جایگزین شده خود را نمایش می‌دهد.

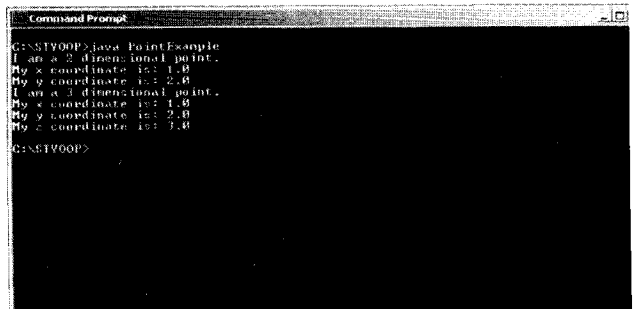
شکل ۴ ـ ۵ اجرا main() فوق را نشان می‌دهد.

جایگزینی روال تحت عنوان تعریف مجدد روال هم شناخته می‌شود. از این طریق کلاس فرزند می‌تواند، روال مورد نیاز خود را در اختیار داشته باشد.

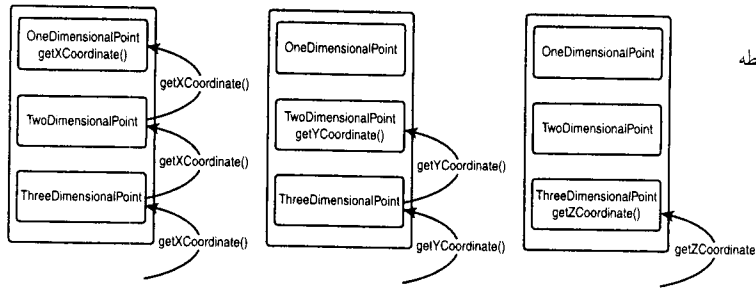
واژه جدید

جایگزینی عبارت است از اینکه کلاس فرزند، روالی از کلاس والد را بر اساس نیازهای خود بازنویسی کند، تا رفتاری مطلوب در کلاس فرزند فراهم کند.

اما شیء چگونه می‌فهمد از کدام تعریف روال استفاده کند؟ تعریف والد یا تعریف فرزند؟ پاسخ این پرسش به ساختار زبان بستگی دارد. سیستم‌های شیءگرا ابتدا تعریف روال را در شیئی که پیغام داده جستجو می‌کنند. اگر تعریفی یافت نشد، سیستم در سلسله مراتب بالا می‌رود تا زمانی که به تعریف روال دست پیدا



شکل ۴ـ۵
آزمون روال toString()
جایگزین شده



شکل ۴ـ۶
انتشار پیغام در اشیاء نقطه

کند. فهمیدن این شیوه‌ی پاسخدهی و مدیریت پیغام‌ها از این رو اهمیت دارد، که اساس جایگزینی بر آن نهاده شده است. تعریف روال کلاس فرزند به این دلیل فراخوانده می‌شود، که در دسترس‌ترین تعریف است. مکانیزمی مشابه در مورد روالها و صفات بازگشتی مؤثر است.

شکل ۴ـ۶ شیوهٔ گسترش تعریف روال در اشیاء نقطه (Point) برای یک فراخوانی getXCoordinate() را نشان می‌دهد. فراخوانی این روال در سلسله مراتب تا جایی بالا می‌رود که تعریف یا پیاده‌سازی روال را بیابد.

نکته

بازگشت به کد و تغییر سطوح دسترسی شاید نادرست به نظر برسد، اما سلسله مراتب وراثت هیچگاه نباید بر اساس تصادف بنا شود. در عوض این سلسله مراتب باید همچنانکه شما برنامه می‌نویسید به صورت طبیعی شکل بگیرد. تغییر در سلسله مراتب هیچ ایرادی ندارد. برنامه‌نویسی شیءگرای عملی، فرایندی مبتنی بر آزمون و خطا و تکرار است. به یاد داشته باشید، اختصاصی نگاه داشتن همه چیز یک قانون فراگیر است. هرچند شرایطی هم وجود دارد که چنین قانونی صادق نیست. تمام این موارد به برنامه‌ای که می‌نویسید بستگی دارد. مثلاً اگر شما تهیه کننده کتابخانه‌های کلاس **(Class Library)** عام باشید و قصد ارایه کد منبع **(Source Code)** به مشتری را نداشته باشید، بهتر است روالهای خود را به صورت پیشفرض محافظت شده قرار دهید، تا مشتری بتواند از کد شما کلاسهای جدید مشتق کند. وقتی در نظر دارید یک زیرکلاس بنویسید، بد نیست یک پروتکل وراثت ایجاد کنید. پروتکل وراثت ساختار مجردی است که تنها برای عناصر محافظت شده کلاس قابل رؤیت است. کلاس والد این روال‌ها را فراخوانی می‌کند و کلاس فرزند با جایگزینی، کارایی را افزایش می‌دهد.

هنگامی که جایگزینی یک روال یا صفت را مدنظر دارید، درک این مطلب ضروری است که تمام روالها یا صفات قابل جایگزینی نیستند. بیشتر زبان‌های شیءگرا دارای کنترل دسترسی هستند. یعنی تعیین اینکه چه کسی می‌تواند روالها را مشاهده کند و به آنها دسترسی داشته باشد. این سطوح دسترسی ذاتاً در یکی از سه دسته‌ای قرار می‌گیرند که در روز دوم مختصراً به آنها اشاره شد:

اختصاصی: سطحی که دسترسی به آن فقط مختص خود کلاس است.

محافظت شده: سطحی از دسترسی که شامل خود کلاس و فرزندان آن می‌شود.

عمومی: دسترسی به روال یا صفت مزبور همگانی است.

روالها و صفات محافظت شده همانهایی هستند که دسترسی به آنها مختص زیرکلاس‌های کلاس اصلی است. هرگز چنین روالهایی را عمومی قرار ندهید. تنها کسانی باید از این روالها یا صفات استفاده کنند که دانش کافی از ساختار و عملکرد کلاس داشته باشند.

تمام صفات غیرثابت و روالهایی که استفاده از آنها فقط برای کلاس معنی‌دار است را اختصاصی نگه دارید. روالهای اختصاصی را برای کاربردهای احتمالی و غیرمحتمل، در سطح محافظت شده قرار ندهید. سطح محافظت شده را فقط مخصوص روالهایی قرار دهید که مطمئن هستید، یک زیرکلاس واقعاً به تغییر آنها نیاز خواهد داشت، چنین نظام‌دهی دقیقی بدان معنی است که شاید بعدها لازم باشد به تعریف کلاس خود رجوع کنید و سطوح دسترسی را تغییر دهید، اما در عوض، طراحی استوارتری خواهید داشت.

با در نظر گرفتن تعریف‌ها و قوانین فوق، می‌توان به راحتی نتیجه گرفت که روالها و خواص محافظت شده و عمومی برای وراثت اهمیت زیادی دارند.

روالها و خواص جدید

روال یا خاصیت جدید، روال یا خاصیتی است که در کلاس فرزند ظاهر می‌شود، اما در کلاس والد موجود نیست. در مثال قبل افزودن روالهای جدید به کلاس فرزند را مشاهده کردید. ThreeDimentionalPoint روالهای جدید ()getZCoordinate و ()getZCoordinate را به خود می‌افزاید. با افزودن روال یا صفت جدید به رابط کلاس فرزند می‌توان کارایی آن را گسترش داد.

روالها و خاصیت‌های بازگشتی

یک روال یا صفت بازگشتی در کلاس والد یا یکی از والدهای قبلی تعریف شده است، اما تعریف آن در کلاس فرزند مشاهده نمی‌شود. هنگامی که از چنین صفت یا روالی استفاده می‌شود، پیغام در طول سلسله مراتب وراثت بالا می‌رود تا به تعریف برسد. این مکانیزم عملاً همان، مکانیزمی است که در بخش روالها و صفات جایگزین شده معرفی شد. در مثال نقطه‌ها ()getXCoordinate مثال بارزی از روالهای بازگشتی است. زیرا در TwoDimentionalPoint تعریف شده، نه در ThreeDimentionalPoint.

روالهای جایگزین شده گاهی ممکن است، همانند روالهای بازگشتی عمل کنند. با اینکه تعریف روال جایگزین شده در فرزند ظاهر می‌شود، اما اکثر زبان‌های شیءگرا مکانیزمی را فراهم می‌کنند، تا بتوان نسخه مربوط به والد (یا والدهای دیگر) روال مزبور را فراخوانی کرد. این توانایی، به کاربر این امکان را می‌دهد که بتواند از قابلیتهای والد و فرزند همزمان استفاده کند. در Java کلمه کلیدی super دسترسی به پیاده‌سازی والد را ممکن می‌کند.

نکته

تمام زبانها دارای کلیدواژه **super** نیستند. در چنین زبان‌هایی برای ایجاد هر کد وراثتی، باید دقت لازمه را لحاظ کنید وگرنه با یک منبع تولید خطا در برنامه مواجه خواهید شد.

انواع وراثت

در کل، سه انگیزهٔ مهم برای استفاده از وراثت وجود دارد:

۱. برای استفاده مجدد از پیاده‌سازی کد
۲. برای ایجاد تفاوت
۳. برای تغییر نوع داده

بهتر است از قبل مطلع باشید که، برخی انواع استفاده مجدد مطلوب‌تر از بقیه هستند. بیایید هرکدام را جداگانه بررسی کنیم.

وراثت در پیاده‌سازی

دیدیم که وراثت اجازه می‌دهد تا از پیاده‌سازی کد یک کلاس، در یک کلاس جدید استفاده مجدد کرد. در این صورت یک کلاس جدید، با کارایی کامل ایجاد می‌شود. مثل شعبده‌بازی!

هم سلسله مراتب Employee و هم مثال Queue/Iterator استفاده مجدد از کد را به نمایش می‌گذارند. در هر دو مورد، کلاس فرزند برخی از رفتارهای موجود در والد را دوباره مورد استفاده قرار داده است.

تذکر

به یاد داشته باشید که در این نوع وراثت، برنامه‌نویس به آنچه ارث برده می‌شود محدود خواهد شد. بنابراین کلاسهایی که می‌خواهید از آنها کلاس مشتق کنید را با دقت انتخاب کنید و بین مزایای این روش و معایب آن تعادل برقرار کنید.

با همهٔ این حرفها، کلاسی که برای وراثت طراحی شده باشد، از روالهای حفاظت شده مناسب استفاده فراوانی خواهد کرد. کلاس فرزند، می‌تواند این روالها را جایگزین کند و نقص احتمالی پیاده‌سازی را رفع کند. جایگزینی می‌تواند اثرات منفی مشتق کردن کلاس از کلاسی با پیاده‌سازی ضعیف یا نامناسب را کاهش دهد.

معایب وراثت پیاده‌سازی

تا به حال وراثت پیاده‌سازی خیلی خوب به نظر می‌رسید. اما باید مراقب بود، آنچه روی کاغذ روشی خوب و مؤثر به نظر می‌رسد، ممکن است در عمل کاری خطرناک باشد. درواقع این نوع وراثت ضعیف‌ترین نوع وراثت است و باید از آن احتراز کرد. استفاده مجدد شاید آسان باشد، اما همچنان که خواهید دید، شاید بهایی سنگین داشته باشد.

برای درک کمبودها، لازم است انواع داده‌ها را در نظر بگیرید. وقتی یک کلاس از دیگری ارث می‌برد، به طور خودکار نوع داده کلاس والد را به خود می‌گیرد. ارث‌بری نوع داده صحیح باید در اولویت قرار گیرد. دلیل این موضوع را بعداً خواهید دانست، اما بهتر است فعلاً آن را به عنوان حقیقت بپذیرید.

دوباره نگاهی به مثال Queue/Iterator بیاندازید. وقتی Iterator از Queue ارث می‌برد، تبدیل به یک Queue می‌شود. این به آن معنی است که می‌توان با Iterator طوری رفتار کرد که انگار از نوع Queue است. از آنجایی که Iterator هم یک Queue است، تمام کارایی‌های Queue را در خود دارد. یعنی روالهایی مانند ()enqueue و ()dequeue هم جزو رابط عمومی Iterator هستند.

روی کاغذ، این موضوع به نظر مشکل‌زا نمی‌رسد، اما بهتر است نگاهی دقیق‌تر به تعریف Iterator بیاندازیم. یک Iterator به سادگی دو روال تعریف می‌کند، یکی برای استخراج عنصر و دیگری برای بررسی اینکه آیا عنصر دیگری هم موجود است یا نه. بر اساس تعریف، نمی‌توان به Iterator عنصری افزود. اما روال ()enqueue دقیقاً همین کار را انجام می‌دهد. در عوض، فقط می‌توان عناصر را از Iterator حذف کرد. درواقع نمی‌توان عنصری را استخراج کرد و آن را درون Iterator باقی گذاشت. اما باز هم Queue عملکرد دیگری دارد. روال ()peek عناصر را استخراج می‌کند، اما آنها را از Queue حذف نمی‌کند. به سادگی می‌توان دید که استفاده از Queue به عنوان والد یک Iterator انتخاب مناسبی نیست، زیرا رفتاری را به Iterator تحمیل می‌کند که درواقع به آن تعلق ندارند.

نکته

برخی زبان‌ها، اجازهٔ وراثت از پیاده‌سازی، بدون نیاز به ارث بردن اطلاعات نوع داده، را می‌دهند. اگر زبان مورد استفاده شما چنین امکانی دارد، در مثال اخیر مشکل چندانی وجود نخواهد داشت. اما، بیشتر زبان‌ها چنین امکانی فراهم نمی‌کنند. از میان زبان‌هایی که چنین امکانی را در اختیار برنامه‌نویس قرار می‌دهند، برخی آن را به صورت خودکار انجام می‌دهند. برخی دیگر مانند C++، اجازه جداسازی را می‌دهند، اما برنامه‌نویس باید صریحاً چنین چیزی را درخواست کند.

نکته

در این کتاب تعریف ساده وراثت ملاک قرار می‌گیرد. با این وصف بحث دربارهٔ وراثت شامل هم رابط و هم پیاده‌سازی خواهد بود.

وراثت ضعیف، برای برنامه‌نویسی مانند هیولای فرانکشتین است. وقتی از وراثت بدون توجه به مسایل دیگر فقط برای استفاده مجدد از کد قبلی بهره می‌گیرید، هیولایی خواهید ساخت، که از بخشهای غیرمرتبط به هم ایجاد شده است.

وراثت برای ایجاد تغییر

این نوع وراثت را قبلاً در مثال ThreeDimetionalPoint مشاهده کردیم. همچنین در مثال Employee. برنامه‌نویس با معیار قرار دادن ایجاد تغییر، کار را تا حد تعیین چگونگی تغییر کلاس فرزند ساده می‌کند.

واژه جدید

برنامه‌نویسی تغییر (Programming by difference) یعنی مشتق کردن کلاسی جدید و افزودن کدی که عملکرد کلاس را طوری تغییر دهد که با عملکرد کلاس والد متفاوت باشد.

در مورد ThreeDimentionalPoint، دیدید که تفاوت با افزودن متغیر z ایجاد شد. برای پشتیبانی مختصهٔ z، کلاس فرزند دو روال جدید به خود افزود. همچنین دیدید که ThreeDimentionalPoint عملکرد روال toString() را مجدداً تعریف کرد.

برنامه‌نویسی تغییر، مفهوم قدرتمندی است، زیرا به برنامه‌نویس این امکان را می‌دهد که فقط کد لازم برای ایجاد تفاوت مورد نظرش را به کلاس فرزند بیافزاید.

چنین کد کوچکتر و ساده‌تری طراحی را ساده‌تر می‌کند. همچنین از آنجایی که کد کمتری نوشته می‌شود، به نظر می‌رسد خطای کمتری هم ایجاد می‌شود. یعنی کد صحیح بیشتری در زمان کمتری نوشته می‌شود. مانند وراثت پیاده‌سازی، در اینجا هم مزیت بلاتغییر ماندن کد جاری، موجود است.

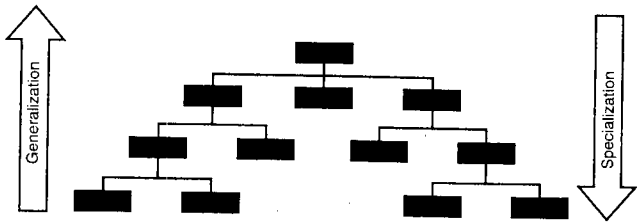
دو راه برای تغییر برنامه از طریق وراثت موجود است: افزودن رفتارها و صفات جدید و تعریف مجدد روالها و صفات قدیمی. هر دو راه تحت عنوان تصریح (Specialization) شناخته می‌شوند.

تصریح

واژه جدید

تصریح عبارت است از فرایندی که طی آن کلاس فرزند، خود را از طریق معرفی تفاوتهایش با کلاس والد، می‌شناساند. بعد از اتمام این کار، کلاس فرزند تنها شامل کدی خواهد بود که تفاوت ایجاد می‌کند.

کلاس فرزند این کار را با تغییر صفات یا روالهای موجود یا تعریف صفات یا روالهای جدید انجام می‌دهد. برخلاف آنچه ممکن است تصور شود، این روش امکان حذف رفتار یا صفات فرزند را ایجاد نمی‌کند. یک کلاس نمی‌تواند به صورت گزینشی مشتق شود.



شکل ۴ـ۷
وقتی در سلسله مراتب بالا می‌روید تعمیم می‌دهید و وقتی پایین می‌روید تصریح می‌کنید.

آنچه تصریح انجام می‌دهد، این است که آنچه کلاس می‌تواند باشد، یا نمی‌تواند باشد را معین کند. یک ThreeDimentiionalPoint همواره می‌تواند یک TwoDimentionalPoint باشد (به جای آن استفاده شود)، اما عکس آن صحیح نیست. در عوض می‌توان گفت که یک ThreeDimentionalPoint تصریحی از TwoDimentionalPoint است و یک TwoDimentionalPoint تعمیمی (Generalization) از ThreeDimentionalPoint است.

شکل ۴ ـ ۷ تفاوت بین تصریح و تعمیم را نشان می‌دهد. همچنانکه در سلسله مراتب پایین می‌روید، تصریح می‌کنید و برعکس همچنانکه بالا می‌روید، کلاسهای بیشتری در گروه جای می‌گیرند، یعنی تعمیم می‌دهید. وقتی تصریح صورت می‌گیرد، هر بار، کلاسهای کمتری تمام شرایط قرار گرفتن در یک گروه را خواهند داشت.

پس دیدید که تصریح به معنی محدود کردن کارایی نیست، بلکه تقسیم‌بندی انواع را محدود می‌کند. لزومی ندارد تصریح به ThreeDimentionalPoint ختم شود. درواقع حتی لازم نیست که از TwoDimentionalPoint آغاز کنید. وراثت هر قدر بخواهید می‌تواند عمیق شود. می‌توانید از وراثت برای ایجاد ساختارهای سلسله مراتب کلاسهای پیچیده استفاده کنید. بسط ایده وراثت که قبلاً به آن پرداختیم، دو مفهوم جدید را ایجاد می‌کند: نیا (Ancestor) و خلف (Descendant).

توجه

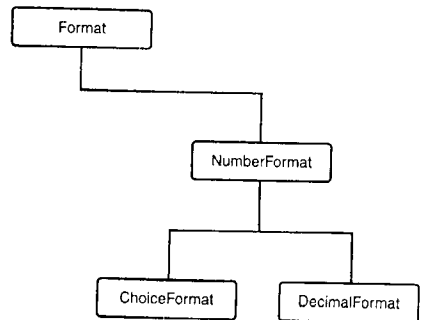
تا جای ممکن از پیچیدگی در سلسله مراتب احتراز کنید. همچنین سعی کنید آن را خیلی طولانی نکنید، سلسله مراتب کوتاه به دلیل قابلیت اداره بهتر، بر سلسله مراتب گسترده و طولانی ارجحیت دارد.

واژه جدید

با در نظر گرفتن یک کلاس فرزند، هر کلاسی که در سلسله مراتب قبل از والد قرار گیرد، یک بنا برای فرزند است. همچنانکه در شکل ۴ ـ ۸ دیده می‌شود Format نیای DecimalFormat است.

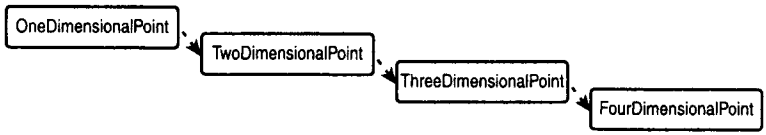
واژه جدید

یک کلاس در نظر بگیرید، هر کلاسی که در سلسله مراتب بعد از آن ظاهر شود یک خلف آن کلاس است. مجدداً با رجوع به شکل ۴ ـ ۸ در زیر DecimalFormat یک خلف Format است.



شکل ۴ـ۸
DecimalFormat یکی از اخلاف Format است.

شکل ۴ـ۹
سلسله مراتب نقطه



فرض کنید، سلسله مراتب وراثت نشان داده شده در شکل ۴ ـ ۹ را داشته باشیم. می‌توان گفت OneDimentionalPoint والد TwoDimentionalPoint و نیای ThreeDimentionalPoint و FourDimentionalPoint است.

همچنین TwoDimentionalPoint, ThreeDimentionalPoint و ForuDimentionalPoint همگی اخلاف OneDimentionalPoint هستند. تمام اخلاف در روالها و صفات نیایشان مشترک هستند.
می‌توانیم چند جملهٔ جالب دیگر درباره این سلسله مراتب کلاس بگوییم. مثلاً OneDimentionalPoint ریشه است و FourDimentionalPoint یک برگ است.

واژه جدید کلاس ریشه (root) یا کلاس پایه (base)، کلاسی است که در سلسله مراتب وراثت بالاترین جایگاه را به خود اختصاص داده است. مثلاً در شکل ۴ ـ ۹ می‌توان دید که OneDimentionalPoint ریشه یا سرسلسله ساختار سلسله مراتبی نقطه است.

واژه جدید یک کلاس برگ (leaf)، کلاسی است که هیچ کلاسی از آن مشتق نمی‌شود. در شکل ۴ ـ ۸ کلاس DecimalFormat یک برگ است.

لازم به ذکر است که کلاسهای خلف (فرزندان) تغییرات ایجاد شده در نیاکان را در خود منعکس می‌کنند. مثلاً، وقتی در TwoDimentionalPoint خطایی یافت می‌شود و به رفع آن می‌پردازید، خطا خودبه‌خود در کلاسهای ThreeDimentionalPoint و FourDimentionalPoint نیز اصلاح می‌شود. بنابراین با اصلاح یک خطا یا بهبود عملکرد کد در یک کلاس، تمام کلاسهای زیر آن در سلسله مراتب منتفع می‌شوند.

وراثت چندگانه (Multiple Inheritance)

تا به اینجا وراثت ساده را طی چند مثال بررسی کردیم. برخی پیاده‌سازیهای وراثت، اجازهٔ مشتق کردن یک کلاس از چند کلاس را می‌دهند. چنین قابلیتی که وراثت چندگانه نامیده می‌شود، یکی از بحث‌برانگیزترین مفاهیم برنامه‌نویسی شیءگرا است. برخی مدعی می‌شوند که چنین کاری تنها برنامه‌نویسی را پیچیده‌تر و مشکل‌تر می‌کند و برخی دیگر روی آن قسم می‌خورند و ادعا می‌کنند، زبان برنامه‌نویسی فاقد این قابلیت، زبان نیست.

به هر صورت وراثت چندگانه در صورتی که بادقت و به طرزی صحیح مورد استفاده قرار گیرد، می‌تواند ارزشمند باشد. اما مشکلاتی هم در راه وجود دارند، که بحث در مورد آنها خارج از حوصلهٔ درس امروز است.

وراثت برای جانشینی نوع داده

آخرین نوع وراثت، برای ایجاد امکان جانشین کردن کلاس به جای والد صورت می‌پذیرد. جانشینی انواع داده (Type Substitution) راهی برای توضیح روابط جانشین‌پذیر است. اما این روابط چه هستند؟ کلاس Line را در نظر بگیرید:

```
public class Line {

    private TwoDimensionalPoint p1;
    private TwoDimensionalPoint p2;

    public Line( TwoDimensionalPoint p1, TwoDimensionalPoint p2 ) {
        this.p1 = p1;
        this.p2 = p2;
    }

    public TwoDimensionalPoint getEndpoint1() {
        return p1;
    }

    public TwoDimensionalPoint getEndpoint2() {
        return p2;
    }

    public double getDistance() {
        double x =
            Math.pow( (p2.getXCoordinate() - p1.getXCoordinate()), 2 );
        double y =
            Math.pow( (p2.getYCoordinate() - p1.getYCoordinate()), 2 );
        double distance = Math.sqrt( x + y );

        return distance;
    }

    public TwoDimensionalPoint getMidpoint() {
        double new_x = ( p1.getXCoordinate() + p2.getXCoordinate() ) / 2;
        double new_y = ( p1.getYCoordinate() + p2.getYCoordinate() ) / 2;
        return new TwoDimensionalPoint( new_x, new_y );
    }

}
```

کلاس Line دو TwoDimentionalPoint را به عنوان آرگومان می‌گیرد و روالهایی برای محاسبه فاصلهٔ دو نقطه و نقطه میانه خط حاصل از آنها در اختیار برنامه‌نویس قرار می‌دهد.

یک رابطه جانشین‌پذیر یعنی اینکه می‌توان هر شیء را که از TwoDimentionalPoint ارث می‌برد را به روال سازنده (Constructor) کلاس Line پاس کرد.

به یاد بیاورید که وقتی یک کلاس فرزند از یک والد مشتق می‌شود، بین والد و فرزند رابطه همانی (Is-a) برقرار است. بنابراین چون ThreeDimentionalPoint همان TwoDimentionalPoint است، می‌توان به راحتی ThreeDimentionalPoint را به سازندهٔ Line پاس کرد.

تابع main() زیر را در نظر بگیرید:

```
public static void main( String [] args ) {
    ThreeDimensionalPoint p1 = new ThreeDimensionalPoint( 12, 12, 2 );
```

```
    TwoDimensionalPoint p2  = new TwoDimensionalPoint( 16, 16 );
    Line l = new Line( p1, p2 );

    TwoDimensionalPoint mid = l.getMidpoint();

    System.out.println( "Midpoint: (" + mid.getXCoordinate() + "," + mid.getYCoordinate() + ")" );
    System.out.println( "Distance: " + l.getDistance() );
}
```

نکته

امکاناتی که روابط جانشین‌پذیری در اختیار برنامه‌نویس قرار می‌دهند، را تصور کنید! مثال خط هندسی فوق می‌تواند راهکاری برای تغییر سریع محیط گرافیک سه بعدی به دوبعدی در یک رابط گرافیکی کاربر **(GUI)** باشد.

شکل ۴ـ۱۰ خروجی تابع فوق را نشان می‌دهد. ملاحظه می‌کنید که برنامه بدون خطا اجرا شده است.

اتصال‌پذیری (Pluggability) مفهوم بسیار عمیقی است. از آنجایی که می‌توان تمام پیغامهایی که برای والد قابل ارسالند، را برای فرزند هم ارسال کرد، می‌توان با آن به صورت جانشینی برای کلاس والد رفتار کرد. این قابلیت دلیل کافی برای حذف نکردن قابلیتهای والد در فرزند است. زیرا در غیر این صورت اتصال‌پذیری امکان‌پذیر نخواهد بود.

با کمک گرفتن از اتصال‌پذیری می‌توانید انواع فرعی (Subtype) جدیدی به برنامه خود اضافه کنید. اگر برنامه شما برای کار کردن با یک کلاس نیا نوشته شده باشد، خودبه‌خود خواهد دانست چگونه از شیء جدید استفاده کند و نیازی به نگرانی و دقت دربارهٔ انواع داده نخواهد بود. زیرا رابطه‌ای از نوع جانشین‌پذیر بین شیء جدید و شیء اصلی برقرار است.

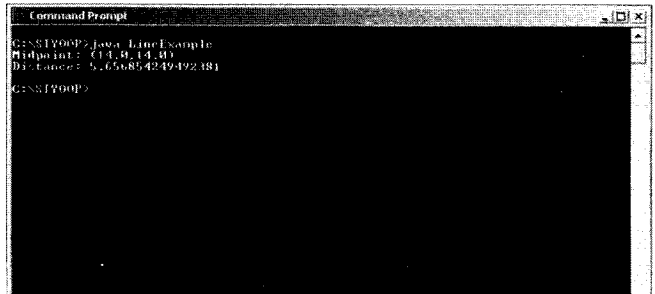
توجه

آگاه باشید که این قبیل روابط در زنجیرهٔ وراثت فقط به پایین منتقل می‌شود. لذا اگر برنامه‌ای برای کار با یک کلاس فرزند نوشته باشد، لزوماً با والد آن کار نخواهد کرد. در عوض هر کلاس خلفی را می‌توان بدون نگرانی به آن پاس کرد.

سازندهٔ Line را به عنوان مثال در نظر بگیرید:

```
public Line (TwoDimentionalPoint P1,TwoDimentionalPoint P2)
```

به این تابع می‌توان هر متغیری از نوع TwoDimentionalPoint یا هریک از فرزندان آن پاس کرد. اما نمی‌توان متغیر OneDimentionalPoint را به آن پاس کرد، زیرا این کلاس در سلسله مراتب قبل از نوع اصلی ظاهر شده است.



شکل ۴ـ۱۰
بررسی رابطه‌ی جانشین‌پذیری

واژه جدید یک نوع فرعی عبارت است از نوع داده‌ای که از طریق وراثت، یک نوع دادهٔ موجود را بسط دهد.

اتصال‌پذیری امکان استفاده مجدد را بیشتر می‌کند. فرض کنید یک نگهدارنده (Container) برای نگه داشتن مقادیری از نوع TwoDimentionalPoint نوشته‌اید. به دلیل وجود قابلیت اتصال‌پذیری، از این نگهدارنده می‌توان برای هریک از اخلاف TwoDimentionalPoint هم به همان صورت استفاده کرد.

اهمیت اتصال‌پذیری در این است، که با استفاده از آن می‌توان کد کلی (Generic) نوشت. به جای اینکه برای تشخیص نوع کلاس مجموعه از عبارتهای case یا if/else داشته باشیم، به راحتی شیء را فقط برای استفاده از TwoDimentionalPoint برنامه‌ریزی می‌کنید.

نکاتی در مورد وراثت صحیح

وراثت نکات طراحی خاص خود را به همراه می‌آورد. با اینکه وراثت ابزار قدرتمندی است، اما می‌تواند مانند طناب داری گلوی برنامه‌نویس بی‌دقت را بفشارد. پس به نکات زیر توجه کنید:

- به طور کلی، سعی کنید از وراثت برای استفاده مجدد از رابطهای موجود یا تعریف روابط جانشین‌پذیری استفاده کنید. علاوه بر این وراثت را می‌توانید برای بسط دادن پیاده‌سازی هم به کار ببرید. البته فقط درصورتی که کلاس حاصله از آزمون همانی سربلند بیرون آید.
- در مبحث استفاده مجدد از کد، ترکیب بر وراثت ارجحیت دارد. وراثت را فقط در صورتی به کار ببرید که بتوانید آزمون همانی را بر سلسله مراتب حاصله اعمال کنید. اما در استفاده از آن زیاده‌روی نکنید.
- همواره به قانون همانی توجه کنید.

سلسله مراتب وراثت صحیح تصادفی پدید نمی‌آید. گاهی ممکن است به یک سلسله مراتب وراثت برخورد کنید. در این صورت دوباره روی کد خود کار کنید. در موارد غیر آن باید سلسله مراتب خود را به صورتی سنجیده طراحی کنید. در هر صورت، اصولی وجود دارند که باید در طراحی به آنها توجه کرد:

- یک قاعدهٔ عملی: سلسله مراتب کلاس خود را حتی‌الامکان ساده نگاه دارید.
- سلسله مراتب وراثت را با دقت طراحی کنید و موارد اشتراک را به کلاس پایه مجرد (Abstract Base Class) منتقل کنید. کلاس مجرد پایه، تعریف روال بدون آوردن کد اجرایی (پیاده‌سازی) را مجاز می‌دارد. از آنجایی که کلاس مجرد پایه، پیاده‌سازی ندارد، نمی‌توان متغیری از نوع آن تعریف کرد. اما، مکانیزم تجرید، کلاس مشتق شده را مجبور می‌کند که برای روال، پیاده‌سازی تعریف کند. کلاسهای مجرد، برای وراثت برنامه‌ریزی شده مفید هستند. زیرا به برنامه‌نویس کمک می‌کند آنچه نیاز به پیاده‌سازی دارد را پیش‌بینی کند.

نکته اگر زبان برنامه‌نویسی مورد استفاده شما، مکانیزم تجرید را پشتیبانی نمی‌کند، می‌توانید روالهای خالی ایجاد کنید. در این صورت باید این موضوع که برنامه‌نویس استفاده کننده باید خود کلاسها را پیاده‌سازی کند، را در مستندات کلاس ذکر کند.

- اغلب کلاسها در تکه‌ای از کد اشتراک دارند. داشتن چندین کپی از یک کد، معنی ندارد. برنامه‌نویس باید کد مشترک را حذف کرده و آن را در یک کلاس والد قرار دهد. البته نباید آن را زیاد در سلسله مراتب

بالا ببرید. تنها به اندازه یک سطح بالاتر از زمان نیاز.

- نمی‌توان همواره سلسله مراتب را از قبل به درستی برنامه‌ریزی کرد. کد تکراری وقتی خود را نشان خواهد داد که چند بار با آن روبرو شوید. از کار مجدد و تغییر کلاسهای خود نترسید. چنین عملی را تعیین مجدد عامل‌ها (Refactoring) می‌نامند.

کپسوله‌سازی بین والد و فرزند به همان اندازه کپسوله‌سازی بین کلاسهای غیرمرتبط اهمیت دارد. سعی کنید به هیچ عنوان به آن بی‌توجه نباشید و در آن سستی به خرج ندهید. چند نکته‌ای وجود دارند که با استفاده از آنها می‌توان از کپسوله‌سازی خطا یا شکسته شدن کپسوله‌سازی آن پیشگیری کرد:

- دقیقاً مانند آنچه بین کلاسهای دیگر اتفاق می‌افتد، بین والد و فرزند از رابطهای خوب تعریف شده استفاده کنید.
- اگر روالهایی را فقط برای استفاده زیرکلاس تعریف می‌کنید، اطمینان حاصل کنید که آنها را حفاظت شده تعریف کرده‌اید، تا فقط برای زیرکلاس قابل مشاهده باشند، بدون اینکه کلاسهای دیگر به روال دسترسی داشته باشند.
- در کل، تعریف و پیاده‌سازی داخلی را برای زیرکلاس باز نگذارید. زیرا در این صورت زیرکلاس به آن پیاده‌سازی وابسته می‌شود. چنین وابستگی تمام مشکلات مطرح شده در درس روز دوم را به همراه دارد.

و سرانجام چند راه حل کلیدی برای وراثت صحیح عبارتند از:

- همواره به خاطر داشته باشید که جانشین‌پذیری هدف اول است. حتی اگر شیء به طور شهودی، بدون دلیل به نظر برسد که باید در سلسله مراتب بیاید، نباید چنین کنید. حتی در صورتی که می‌توانید یا ادراک شهودی شما اصرار می‌ورزد که چنین کنید، نباید به چنین کاری دست بزنید.
- با ترتیب مناسبی برنامه بنویسید تا کد قابل مدیریت باشد.
- برای استفاده مجدد از کد، ترجیحاً از ترکیب استفاده کنید تا وراثت. تغییر در کلاسهای ترکیب شده ساده‌تر است.

خلاصه

در برنامه‌نویسی شیءگرا، دو نوع رابطه وجود دارد: رابطه استفاده (use) و رابطه وراثت. هر کدام از روابط نوع خاصی از استفاده مجدد را فراهم می‌کند. هریک مزایا و معایب خاص خود را دارند.

تعریف متغیر از کلاس، انعطاف‌پذیری را کاهش می‌دهد. از این راه نمی‌توان از پیاده‌سازی استفاده کرد یا کلاسی را توسعه داد. در عوض فقط باید به سادگی از قطع (cut) و چسباندن (paste) استفاده کنید. وراثت بر این کمبودها غلبه می‌کند. این کار از طریق ایجاد مکانیزمی برای استفاده مجدد مطمئن و مؤثر صورت می‌گیرد.

استفاده مجدد از پیاده‌سازی، راهی سریع اما ضعیف است. برخلاف روش ساده کپی کردن و چسباندن کد، تنها یک کپی از کد وجود دارد. اما این روش، طراحی شما را محدود می‌کند.

وراثت برای تغییر، اجازه می‌دهد با تعیین و تبیین تفاوتهای فرزند با والد، کلاس جدید را تعریف کرد و سرانجام وراثت برای جانشینی، به برنامه‌نویس اجازه می‌دهد که کد کلی بنویسد. با استفاده از جانشین‌پذیری می‌توانید هر وقت لازم باشد، بدون کوچکترین تغییری در کد، فرزند را جانشین والد کنید.

این خاصیت برنامه را قادر می‌سازد تا در برابر تغییر نیازهای آینده، انعطاف‌پذیر باشد.

چگونه وراثت اهداف برنامه‌نویسی شیءگرا را تأمین می‌کند.

وراثت برنامه‌ای با خواص زیر ایجاد می‌کند:

۱. طبیعی بودن
۲. قابل اعتماد بودن
۳. قابل استفاده مجدد
٤. قابل اداره
۵. توسعه‌پذیر
۶. کاهش زمان کدنویسی

اهداف زیر به صورتی که خواهد آمد، حاصل می‌شوند:

- طبیعی بودن: وراثت اجازه می‌دهد که جهان بیرون را به صورت طبیعی‌تری مدل کنید. از طریق وراثت می‌توان سلسله مراتبهای پیچیدهٔ وراثتی بین کلاسها ایجاد کرد. به عنوان انسان، این خاصیت و میل طبیعی ماست که بخواهیم آنچه در پیرامون ما قرار دارد را گروه‌بندی یا طبقه‌بندی کنیم. وراثت این قابلیت را در دنیای برنامه‌نویسی تحقق می‌دهد. علاوه بر این، وراثت آرزوی برنامه‌نویس‌ها برای جلوگیری از دوباره کاری را برآورده می‌کند.
- قابل اعتماد بودن: کد مبتنی بر وراثت قابل اعتماد است. وراثت کد را ساده می‌کند. وقتی بر اساس تغییر برنامه می‌نویسید، تنها کدی را عوض می‌کنید که لازم است تغییر یابد. در نتیجه هر کلاس می‌تواند رد پای کوچکتری برجای بگذارد. هر کلاس می‌تواند فقط بر کاری که قرار است انجام دهد تمرکز کند. کدنویسی کمتر به معنی خطاهای کمتر هم خواهد بود. وراثت امکان استفاده مجدد از کد آزمایش شده و مطمئن که قبلاً نوشته شده را فراهم می‌کند. استفاده مجدد از کد آزمایش شده، همواره بر نوشتن کد جدید برتری دارد. سرانجام اینکه خود مکانیزم وراثت مطمئن است. سازوکار وراثت ذاتی خود زبان است و شما نیازی نخواهید داشت که سازوکاری جدید برای آن تعریف و پیاده‌سازی کنید و دغدغهٔ این را داشته باشید که کاربر قوانین شما را رعایت کند.

با تمام این حرفها وراثت چندان هم دلخواه نیست. وقتی کلاسی مشتق می‌کنید باید گوش به زنگ باشید که مبادا از طریق خراب کردن وابستگی‌های پنهان با بی‌توجهی خطاهای نامحسوسی ایجاد کنید. در هنگام مشتق کردن با احتیاط قدم بردارید.

- قابلیت استفاده مجدد: وراثت به استفاده مجدد کمک می‌کند. طبیعت وراثت آن است که بتوان با آن از کلاسهای موجود، کلاسهای جدید ایجاد کرد.

علاوه بر این وراثت شما را مجاز می‌دارد که از کلاس حتی درجایی که مدنظر طراحی اصلی آن نبوده استفاده کنید. از طریق جایگزینی و تغییر در کلاس موجود می‌توان کلاس جدیدی مشتق کرد که با رفتار جدید بتواند برای حل مسأله‌ای متفاوت مورد استفاده قرار گیرد.

- مدیریت‌پذیر بودن: وراثت به امکان مدیریت کلاس کمک می‌کند. استفاده مجدد از کد قبلی، یعنی

خطاهای کمتر در کد جدید. علاوه بر این وقتی خطایی را در یک کلاس رفع می‌کنید، تمام زیرکلاسهای آن هم منتفع خواهند شد.

به جای شیرجه زدن در کد و افزودن مستقیم ویژگیها، وراثت برنامه‌نویس را آزاد می‌گذارد که کدهای موجود را به منزله زیرساختی برای ایجاد کلاس جدید در نظر بگیرد. تمام روالها، ویژگیها و اطلاعات انواع داده، جزیی از کلاس شما می‌شوند. برخلاف بریدن و چسباندن (cut & paste)، تنها یک کپی از کد اصلی موجود است. بنابراین حجم کدی که باید مدیریت شود کاهش می‌یابد.

اگر قرار بود در کد موجود مستقیماً تغییر ایجاد کنید، شاید به کلاس پایه آسیب می‌رساندید و نسبتهای سیستمی که کلاس را به کار می‌گیرد، را بر هم می‌زدید.

- **توسعه‌پذیر بودن: وراثت توسعه کلاس یا تصریح را ممکن می‌سازد. می‌توانید یک کلاس موجود را در نظر گرفته و به آن کارایی‌های جدید بیافزایید. دو نوع آخر وراثت، برنامه‌نویس را ترغیب به توسعه کلاس می‌کنند.**
- **صرفهٔ زمانی: قبلاً دیدیم که استفاده مجدد ساده چگونه توانست زمان صرف شده برای توسعه را کاهش دهد. برنامه‌نویسی مبتنی بر تفاوت، یعنی اینکه کد کمتری باید نوشته شود، در نتیجه کار زودتر تمام می‌شود. اتصال‌پذیری یعنی اینکه می‌توان بدون اینکه لازم باشد تغییر زیادی در کد موجود داده شود، خواص جدیدی به کلاس افزود.**

افزون بر اینها وراثت می‌تواند آزمون کد را آسانتر کند. چون تنها باید کارایی‌ها جدید و روابط آنها با کارکرد قبلی را مورد بررسی قرار داد.

پرسش و پاسخ

در درس امروز سه دلیل برای استفاده از وراثت ذکر شد. آیا این سه دلیل باید جداگانه لحاظ شوند یا می‌توان آنها را ترکیب کرد؟ مثلاً، در صورتی که بخواهیم برای ایجاد تغییر کلاسی مشتق کنیم، آیا می‌توان همزمان از وراثت پیاده‌سازی هم بهره گرفت؟

خیر. دلیل وراثت لزومی ندارد جداگانه در نظر گرفته شود. می‌توان کلاس جدید را مشتق کرد و در نهایت هریک از سه انگیزه را اقناع کرد.

به نظر می‌رسد وراثت برای استفاده مجدد از کد نادرست باشد. آیا استفاده مجدد یکی از اهداف برنامه‌نویسی شیءگرا نیست؟

استفاده مجدد تنها یکی از اهداف برنامه‌نویسی شیءگرا است. برنامه‌نویسی شیءگرا راهبردی است که به شما اجازه می‌دهد راه حل مشکل خود را به صورت طبیعی‌تری مدل کنید: از طریق اشیاء. با اینکه استفاده مجدد مهم است، نباید صرفاً بر آن متمرکز شد و اهداف دیگر را نادیده گرفت. مثال Queue/Iterator را در نظر بگیرید. آیا مدل Iterator، مدلی طبیعی بود؟ البته که نه!

علاوه بر این، وراثت برای استفاده مجدد از پیاده‌سازی، تنها یک راه استفاده مجدد است. اغلب واگذاری بهترین راه برای استفاده مجدد از کد است. اگر هدف شما تنها استفاده مجدد باشد، وراثت راه حل مناسبی

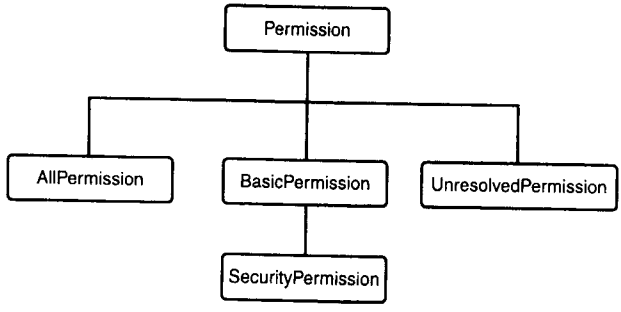
نیست. وراثت وقتی راهکاری مناسب و عالی است که بخواهید تغییر ایجاد کنید یا از جانشینی انواع استفاده کنید.

کارگاه

پرسشهای خودآزمایی و پاسخ‌های آنها برای حصول به درک بهتر تهیه شده‌اند.

پرسشها

۱. **برخی محدودیتهای استفاده مجدد را بیان کنید.**
۲. **وراثت چیست؟**
۳. **سه نوع مختلف وراثت کدامند؟**
٤. **چرا مشتق کردن پیاده‌سازی خطرناک است؟**
۵. **برنامه‌نویسی با ایجاد تغییر چیست؟**
۶. **وقتی کلاس مشتق می‌کنید، کلاس می‌تواند سه نوع روال یا خاصیت داشته باشد. آن سه کدامند؟**
۷. **مزایای برنامه‌نویسی با تغییر کدامند؟**
۸. **سلسله مراتب شکل ۴ ـ ۱۱ را در نظر بگیرید. و توجه خود را به کلاس Permission معطوف کنید.**
فرزندان آن کدامند؟ اخلاف آن کدامند؟
کدام کلاس ریشه است؟ کدام کلاسها برگ هستند؟
در نهایت، آیا Permission نیای SecurityPermission است؟



شکل ۴ ـ ۱۱ سلسله مراتب Permission

۹. **وراثت برای جانشینی انواع چیست؟**
۱۰. **وراثت چگونه می‌تواند کپسوله‌سازی را تخریب کند؟ چگونه می‌توان از آن جلوگیری کرد؟**

تمرین‌ها

۱. اگر بخواهیم کلاسی از کلاس زیر مشتق کنیم، چه مشکلی ممکن است رخ دهد؟

```
public class Point {
    public Point( int x, int y) {
        this.x = x;
        this.y = y;
    }
```

```
public Point getLocation() {
    return new Point( x, y );
}
public void move( int x, int y ) {
    this.x = x;
    this.y = y;
}
public void setLocation( int x, int y ) {
    this.x = x;
    this.y = y;
}
public void setLocation( Point p ) {
    this.x = p.x;
    this.y = p.y;
}
public int x;
public int y;
}
```

۲. چگونه از وقوع این مشکل جلوگیری کنیم؟

روز ۵

Object Oriented Programming

وراثت: زمان نوشتن کد

وراثت ابزاری قدرتمند است. امروز استفاده از این ابزار جدید را با انجام چند تمرین مرور خواهید کرد. در انتهای درس امروز، باید بر مطالب ارایه شده در درس روز چهارم به خوبی مسلط شوید.

آنچه امروز خواهید آموخت

- چگونه در برنامه‌نویسی از وراثت استفاده کنید.
- اهمیت همانی و مالکیت
- چگونه Java رابطه همانی و مالکیت را از بین می‌برد.

کارگاه ۱: وراثت ساده

لیست ۵ـ۱ کلاس پایه MoodyObject را نشان می‌دهد.

لیست ۵-۱ MoodyObject.java

```
public class MoodyObject {

    // return the mood
    protected String getMood() {
    return "moody";
    }
```

لیست ۵-۱ MoodyObject.java

```
    // ask the object how it feels
    public void queryMood() {
    System.out.println("I feel " + getMood() + " today!");
    }

}
```

کلاس MoodyObject یک متد عمومی را تعریف می‌کند: ()queryMood. این متد وضعیت شیء را در خط فرمان (Command Prompt) چاپ می‌کند. کلاس MoodyObject همچنین متدی از نوع حفاظت‌شده تعریف می‌کند. متد ()queryMood از این متد حفاظت شده برای پاسخ دادن استفاده می‌نماید. کلاسهای مشتق شده می‌توانند به راحتی متد ()getMood را جایگزین (override) نمایند تا حالتشان را به نحو بهتری نمایش دهند.

بنابراین اگر زیرکلاس بخواهد متن نمایش داده شده در خط فرمان را عوض کند کافی است متد ()queryMood را جایگزین نماید.

تعریف مسأله

در این کارگاه دو زیرکلاس را ایجاد خواهید کرد: SadObject و HappyObject. هر دو زیرکلاس باید متد ()getMood را جایگزین نمایند تا پیغامهای مناسب وضعیتشان چاپ گردد.

همچنین این دو کلاس باید یکسری از متدهای دیگر را برای مقاصدشان تعریف نمایند. کلاس SadObject باید متدی به صورت Public void cry() تعریف نماید و به همین ترتیب HappyObject باید متدی به صورت Public void laugh() تعریف نماید. متد ()laugh باید پیغام "hahaha" و متد ()cry باید پیغام "boo hoo" را در خط فرمان چاپ نماید.

لیست ۵ـ۲ برنامه‌ای است که می‌توانید از آن برای آزمایش کلاسهای ایجاد شده استفاده نمایید.

لیست ۵-۲ MoodyDriver.java

```
public class MoodyDriver {
    public final static void main( String [] args ) {
        MoodyObject mo = new MoodyObject();
        SadObject   so = new SadObject();
        HappyObject ho = new HappyObject();

        System.out.println( "How does the moody object feel today?" );
        mo.queryMood();
        System.out.println( "" );
        System.out.println( "How does the sad object feel today?" );
        so.queryMood(); // notice that overriding changes the mood
        so.cry();
        System.out.println( "" );
        System.out.println( "How does the happy object feel today?" );
```

لیست ۵-۲ (ادامه)

```
        ho.queryMood(); // notice that overriding changes the mood
        ho.laugh();
        System.out.println( "" );
    }
}
```

توجه بخش بعدی راه حل کارگاه ۱ را توضیح می‌دهد. تا زمانی که کارگاه ۱ را انجام نداده‌اید، به سراغ این بخش نروید.

حل و بحث

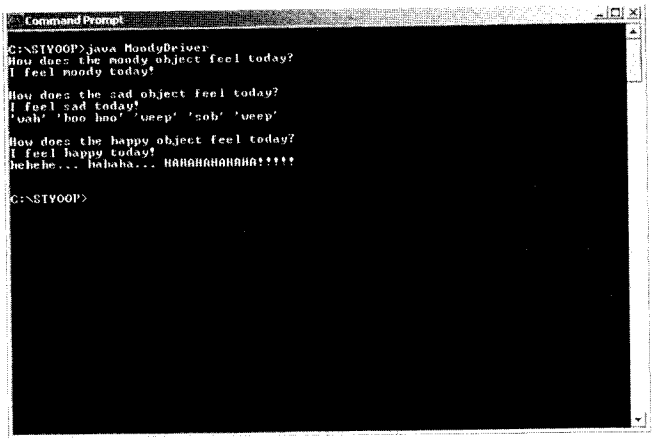
لیستهای ۵-۳ و ۵-۴ یک راه ممکن برای حل کارگاه ۱ را نشان می‌دهند.

لیست ۵-۳ HappyObject.java

```
public class HappyObject extends MoodyObject {

    // redefine class's mood
    protected String getMood() {
    return "happy";
    }

    // specialization
    public void laugh() {
    System.out.println("hehehe... hahaha... HAHAHAHAHAHA!!!!!");
    }
}
```

لیست ۵-۴ SadObject.java

```
public class SadObject extends MoodyObject {

    // redefine class's mood
    protected String getMood() {
    return "sad";
    }

    // specialization
    public void cry() {
    System.out.println("'wah' 'boo hoo' 'weep' 'sob' 'weep'");
    }
}
```



شکل ۵ـ۱
خروجی برنامه MoodyDriver

اگر برنامه را اجرا نمایید، خروجی برنامه چیزی شبیه شکل ۵ ـ ۱، خواهد بود.
نکته جالب توجه فراخوانی متد queryMood() است. زمانی که این تابع را در کلاس SadObject فراخوانی می‌کنید، پیغام "I feel sad today!" بر روی نمایشگر چاپ می‌شود. به همین ترتیب اگر این متد را در کلاس HappyObject فراخوانی کنید پیغام "I feel happy today!" چاپ خواهد شد.

بهتر است نگاه دقیقتری به queryMood() بیاندازیم. متد queryMood() تابع getMood() را در خود صدا می‌زند تا وضعیت و حالت شیء را گزارش کند. از آنجا که زیرکلاسها تعریف getMood() را تغییر داده‌اند، queryMood() متد getMood() را در زیرکلاسها فراخوانی می‌کند. این رفتار مثالی از پروسه‌ای است که در شکل ۴ ـ ۶ از روز چهارم نشان داده شده است.

کارگاه ۲: استفاده از کلاسهای مجرد برای وراثت طرح‌ریزی شده

زمانهایی پیش می‌آید که نیاز است کلاسهایی را طراحی کنید تا دیگران بتوانند کلاسهای دیگری از آن مشتق کنند. با طراحی چند کلاس مرتبط با هم ممکن است به این نکته پی ببرید که یکسری از کدها در همه کلاسها مشترک است. تجربه نشان می‌دهد که در چنین مواقعی بهتر است همه کدهای مشترک را در کلاسی پایه قرار داد. با نوشتن کلاس پایه، می‌توان دیگر کلاسها را از آن مشتق کرد.

با انجام این عمل متوجه خواهید شد که نیازی به فراخوانی کلاس پایه به صورت مستقیم نیست. به عبارت دیگر، اگرچه کلاس پایه کدهای مشترک را در خود دارد هیچ نیازی به تعریف شیء از آن کلاس نیست. درواقع زیرکلاسها از کلاس پایه استفاده کرده و در صورت نیاز رفتار آن را تغییر خواهند داد.

کلاس Employee را به صورت زیر در نظر بگیرید:

```
public class Employee {

    private String first_name;
    private String last_name;
    private double wage;

    public Employee( String first_name, String last_name, double wage ) {
        this.first_name = first_name;
        this.last_name  = last_name;
```

```
        this.wage = wage;
    }

    public double getWage() {
        return wage;
    }

    public String getFirstName() {
        return first_name;
    }

    public String getLastName() {
        return last_name;
    }
    public String printPaycheck() {
        String full_name = last_name + ", " + first_name;
        return ( "Pay: " + full_name + " $" + calculatePay() );
    }
}
```

از کلاس Employee می‌توان به عنوان کلاس پایه برای کلاسهای HourlyEmployee, CommissionedEmployee و SalariedEmployees استفاده کرد. هر زیرکلاس نحوه محاسبه حقوقی که باید پرداخت شود را می‌داند. این در صورتی است که الگوریتم پرداخت برای هریک از زیرکلاسها متفاوت است. زمانی که سلسله مراتب فوق را طراحی کردم، به این نکته توجه کردم که هر زیرکلاس نیاز دارد تا متد ()calculatePay مخصوص خودش تعریف کند. مشکل کوچکی وجود دارد: کلاس Employee هیچ قانونی جهت محاسبه پرداختی ندارد. درواقع فراخوانی تابع ()calculatePay برای این کلاس بی‌معنی است، چراکه هیچ الگوریتمی برای محاسبه مقدار پرداختی به یک کارمند کلی وجود ندارد.

یک راه حل آن است که متد ()calculatePay را برای این کلاس (Employee) تعریف نکنیم. ولی عدم تعریف این تابع تصمیمی نادرست است. چراکه رفتار یک کارمند را به صورت کلی مدل نمی‌کند. هر کارمند می‌داند چگونه باید حقوق پرداختی خود را محاسبه کند. درواقع همه تفاوتها به نحوه پیاده‌سازی ()calculatePay باز می‌گردد. به همین خاطر این متد را در کلاس پایه باقی گذاشته‌ایم.

اگر متد ()calculatePay را در کلاس پایه تعریف نکنیم. مدل یک کارمند را به صورت کلی نمی‌توان ایجاد کرد. یک راه ساده نوشتن متدی است که مقدار دستمزد را برگرداند.

این کار راه حل شفاف و روشنی به دست نمی‌دهد. هیچ تضمینی وجود ندارد که برنامه‌نویس دیگری برای ایجاد یک زیرکلاس جدید این متد را جایگزین کند. خوشبختانه، OOP نوع ویژه‌ای از کلاس را برای وراثت طرح‌ریزی شده، در اختیار گذاشته است: کلاس مجرد.

یک کلاس مجرد بسیار شبیه دیگر کلاسها است. تعریف کلاس رفتارها و خواص را همچنان کلاس‌های عادی تعریف می‌نماید. با این حال نمی‌توانید از این کلاس برای تعریف مستقیم اشیاء و نمونه‌هایی از آن کلاس استفاده نمایید. چراکه یک کلاس مجرد تعدادی از متدها را تعریف نکرده باقی می‌گذارد.

واژه جدید یک متد تعریف شده ولی پیاده‌سازی نشده، متد مجرد (Abstract Method) نامیده می‌شود. تنها کلاسهای مجرد می‌توانند متدهای مجرد تعریف کنند.

این وظیفه کلاسهای فرزند (یا به عبارت دیگر کلاسهای مشتق شده) از کلاس مجرد است که متدهای مجرد را پیاده‌سازی کنند. نگاهی به کلاس مجرد Employee بیاندازید:

```
public abstract class Employee{
...
    public abstract double calculatePay();
    //rest of the definition remains the same
}
```

کلاس مجرد Employee متدی به نام calculatePay() تعریف کرده است و پیاده‌سازی را انجام نداده است. درواقع این وظیفه هریک از کلاسهای مشتق شده از این کلاس است که این روال را پیاده نمایند. برای مثال زیرکلاس HourlyEmployee این متد را به صورت زیر پیاده کرده است:

```
public class HourlyEmployee extends Employee {

    private int hours; // keep track of the # of hours worked

    public HourlyEmployee( String first_name, String last_name, double wage ) {
super( first_name, last_name, wage ); // call the original constructor in order to properly
initialize
    }

    public double calculatePay() {
        return getWage() * hours;
    }

    public void addHours( int hours ) {
        this.hours = this.hours + hours;
    }

    public void resetHours() {
        hours = 0;
    }

}
```

با تعریف متدهای مجرد، زیرکلاسها مجبورند خود به پیاده‌سازی متدها اقدام کنند. درواقع با ایجاد کلاسها و متدهای مجرد، زیرکلاسها مجبور به تعریف دوباره آنها (از لحاظ کارکرد) هستند.

تعریف مسأله

در کارگاه ۱ کلاسی به نام MoodyObject ایجاد کردید. تمام زیرکلاسها متد getMood() را دوباره تعریف کردند. در کارگاه ۲ سلسله مراتب کلاسها اندکی تغییر کرد. حال متد getMood() را به صورت مجرد ایجاد

کنید. همچنین باید تغییراتی در MoodyDriver ایجاد کنید تا مستقیماً از MoodyObject استفاده نکند. هیچ نیازی به تغییر در کلاسهای SadObject و HappyObject نیست. چرا که آنها هر کدام getMood() را پیاده کرده‌اند.

توجه بخش بعدی راه حل کارگاه ۲ را ارایه داده است. بنابراین تا زمانی که کارگاه ۲ را تمام نکرده‌اید، به این بخش مراجعه نکنید.

حل و بحث

لیستهای ۵-۵ و ۵-۶ تغییراتی در تعریف کلاسهای MoodyObject و MoodyDriver ایجاد کردهاند.

```
/**
 * MoodyObject is a base class for writing objects that have a mood.
 * @author Tony Sintes STYOOP
 * @version 1.0
 */
```

لیست ۵-۵ MoodyObject.java

```
public abstract class MoodyObject {

    // return the mood
    protected abstract String getMood();

    // ask the object how it feels
    public void queryMood() {
    System.out.println("I feel " + getMood() + " today!");
    }

}
```

لیست ۵-۶ MoodyDriver.java

```
public class MoodyDriver {
    public final static void main( String [] args ) {
        //MoodyObject mo = new MoodyObject(); // can no longer instantiate MoodyObject
        SadObject   so = new SadObject();
        HappyObject ho = new HappyObject();

        //System.out.println( "How does the moody object feel today?" );
        //mo.queryMood();
        //System.out.println( "" );
        System.out.println( "How does the sad object feel today?" );
        so.queryMood(); // notice that overriding changes the mood
        so.cry();
        System.out.println( "" );
        System.out.println( "How does the happy object feel today?" );
        ho.queryMood(); // notice that overriding changes the mood
```

لیست ۵-۶ (ادامه)

```
        ho.laugh();
        System.out.println( "" );
    }
}
```

تغییرات انجام شده فوق‌العاده ساده هستند. کلاس MoodyObject اقدام به تعریف متد getMood() به صورت مجرد کرده است و پیاده‌سازی آن را رها کرده است. از این رو زیرکلاسها موظفند خود اقدام به پیاده‌سازی این تابع نمایند. زمانی که که queryMood() نیاز به این تابع داشته باشده، به صورت خودکار از تابع مجرد getMood() در ریزکلاسها تعریف شده‌اند، استفاده می‌نماید.

استفاده از کلاسهای مجرد باعث می‌شود زیرکلاسها جهت استفاده از کلاس پایه اقدام به پیاده‌سازی توابع و متدهای مجرد (بسته به الگوریتم و نحوه پیاده‌سازی آن کلاس) کنند. به عنوان یک برنامه‌نویس زمانی که به یک کلاس پایه و مجرد نگاه می‌اندازید، چشم‌انداز نحوه وراثت زیرکلاسها و پیاده‌سازی آنها برای شما ترسیم می‌شود. با تعریف متدهای مجرد اطمینان دارید که زیرکلاسها در سلسله مراتب کلاسها در جای درستی قرار گرفته‌اند.

زمانی که کلاس پایه متدهای زیادی داشته باشد، تشخیص اینکه کدام متدها باید جایگزین شوند، اندکی مشکل می‌نماید. کلاسهای مجرد در این زمینه راهنماییهای لازم را به شما می‌نمایند.

کارگاه ۳: حساب بانکی ـ تمرین وراثت ساده

حال زمان این است که آنچه در مورد وراثت آموخته‌اید را آزمایش کنید. خوب به بانک OO راکه در بخشهای قبل با آن آشنا شدید برگردیم و ببینیم که وراثت چه کمکی می‌تواند به ما بکند.

بانک OO اجازه می‌دهد که مشتریان دارای چندین نوع حساب بانکی باشند: حساب پس انداز، حساب جاری، حساب سپرده گذاری زمانی و حساب اعتباری.

حساب کلی (عمومی)

هریک از انواع حسابها، اجازه می‌دهند مشتری وجهی را به حساب ریخته و یا از آن برداشت کند و همواره میزان وجه را چک کند. حساب عمومی اجازه برداشت بیش از اعتبار موجود (Overdraft) را نمی‌دهد.

حساب پس‌انداز

این نوع حساب پس از محاسبه سود مقداری بیش از آنچه مشتری در حساب واریز کرده است، پرداخت می‌کند. برای مثال اگر نرخ سود بانکی ۲٪ باشد و مشتری مقدار ۱۰۰۰$ به حساب واریز کرده باشد، پس از اعمال نرخ سود، موجودی حساب برابر ۱۰۲۰$ می‌شود:

(نرخ سود * موجودی) + موجودی = موجودی

از حساب پس‌انداز نمی‌توان بیش از موجودی پول برداشت کرد.

حساب سپرده گذاری زمانی

این حساب نیز، سود به موجودی پرداخت می‌کند. در این نوع حساب اگر مشتری قبل از موعد نسبت به

برداشت موجودی اقدام کند، بانک درصدی را به عنوان جریمه از موجودی کسر می‌کند. برای مثال اگر مشتری ۱۰۰۰$ واریز کرده و درصد جریمه ۵٪ باشد، در این صورت موجودی حساب به میزان ۱۰۰۰$ کاهش پیدا کرده و در ضمن مشتری می‌تواند تنها مقدار ۹۵۰$ را برداشت نماید. اگر موعد به پایان برسد، بانک جریمه‌ای را در نظر نمی‌گیرد:

میزان برداشتی ـ موجودی = موجودی

ولی

(مقدار جریمه * پول واریز شده) ـ پول واریز شده = پول پرداختی به مشتری

از این نوع حساب نمی‌توان بیش از مقدار موجودی، برداشت کرد.

حساب جاری

برخلاف حسابهای پس‌انداز و سپرده‌گذاری زمانی، حساب جاری سودی به موجودی اضافه نمی‌کند. این حساب به مشتری اجازه صدور چک و انتقال اسناد مالی و پولی از طریق ماشینهای خودکار (ATM) را می‌دهد و بانک تعداد انتقالها را در ماه محدود کرده است.

در صورتی که مشتری بخواهد به تعداد بیشتری نسبت به نقل و انتقالات مالی اقدام نماید، بانک به ازای هر نقل و انتقال کارمزد دریافت می‌کند. برای مثال اگر مشتری مجاز به ۵ نقل و انتقال مالی در ماه باشد و ۸بار نقل و انتقال انجام داده باشد و کارمزد هر نقل و انتقال ۱$باشد، در این صورت به میزان ۳$ از مشتری کارمزد دریافت می‌کند:

هزینه هر نقل و انتقال * (تعداد مجاز در ماه ـ تعداد نقل و انتقالات) =کارمزد

از حساب جاری نیز نمی‌توان بیش از موجودی برداشت کرد.

حساب اعتباری

در آخر آن که حساب اعتباری اجازه می‌دهد مشتری بیش از اندازه موجودی از حساب برداشت کند. البته بانک برای این کار کارمزد دریافت می‌کند. برای مثال اگر موجودی مشتری برابر ۱۰۰۰-$ شود و میزان کارمزد ۲۰٪ باشد، مشتری باید مبلغ ۲۰۰$ به بانک پرداخت نماید. در این صورت پس از محاسبه کارمزد، موجودی برابر ۱۲۰۰-$ می‌شود:

(میزان کارمزد * موجودی) + موجودی = موجودی

به خاطر داشته باشید که بانک تنها برای موجودیهای منفی کارمزد می‌گیرد! در غیر این صورت تنها به پرداخت پول اقدام می‌نماید. برخلاف حساب جاری تعداد نقل و انتقالات در ماه محدود نمی‌باشد. درواقع بانک مشتریان را تشویق به استفاده از این نوع حساب می‌نماید (به دلیل دریافت کارمزد!)

تعریف مسأله

وظیفه شما استفاده از وراثت جهت پیاده‌سازی انواع حساب معرفی شده در صفحات قبل است. درواقع باید انواع حسابهای زیر را ایجاد کنید:

- Base Account
- SavingsAccount
- TimeMaturityAccount

- CheckingAccount
- OverdraftAccount

BaseAccount کلاس پایه است و شامل تمامی اعمال مشترک برای همه حسابهاست. این تنها راهنمایی در زمینه وراثت به شماست. بقیه اعمال مربوط به کارگاه بر عهده خودتان است.

توجه

خود را درگیر جزئیات غیرضروری نکنید. به خاطر داشته باشید که شما کارگاه را برای آشنایی بیشتر با وراثت و به دست آوردن تجربیات عملی انجام می‌دهید نه نوشتن یک سیستم حسابداری جامع. به همین منظور نیازی به آزمایش پارامترهای ورودی نیست مگر آنجایی که نیاز باشد. می‌توانید فرض کنید همه آرگومانهای ورودی صحیح هستند.

برای سادگی امر ملاحظات چندی را در نظر بگیرید. برای کارمزدها، حسابهای سپرده‌گذاری و نرخهای سود بهره، فرض کنید شخص ثالثی به تقویم نگاه خواهد کرد. درواقع این نوع از اعمال را در کلاسهای خود پیاده‌سازی نکنید. در عوض متدی را برای شیء دیگری در نظر بگیرید تا این کار را بکند. برای مثال، کلاس SavingsAccount باید شامل متد addInterest() جهت اعمال نرخ بهره باشد. یک شیء خارجی در زمان مناسب برای محاسبه بهره این متد را فراخوانی خواهد کرد.

به همین ترتیب CheckingAccount باید نسبت به تعریف accessFee() جهت اعمال کارمزد اقدام نماید. زمانی که این متد فراخوانی می‌شود، کارمزدها محاسبه شده و از موجودی کسر می‌گردند.

روز چهارم شما را با مفهوم super آشنا کرد. super مفهوم پیچیده‌ای نیست. عبارات زیر را در نظر بگیرید:

```
public CommissionedEmployee( String first_name, String last_name, double wage, double
commission ) {
super( first_name, last_name, wage ); // call the original constructor in order to properly
initialize
    this.commission    = commission;
}
```

زمانی که از super در تابع سازنده استفاده می‌کنید، به شما اجازه می‌دهد که مستقیماً تابع سازنده کلاس والد (parent) را فراخوانی کنید. در این صورت آرگومانهای ارسالی از طریق super مقداردهی اولیه شده و کلاس جهت انجام اعمال بعدی آماده می‌گردد. در صورتی که از super استفاده ننمایید، Java تلاش می‌کند به طور خودکار این کار را برای شما انجام دهد.

از super می‌توان در دیگر متدها نیز استفاده کرد. برای مثال کلاس veryHappyObject را در نظر بگیرید:

```
public class veryHappyObject extends HappyObject{
    //redefine class's mood
    protected String getMood(){
        String old_mood=super.getMood();
        return "very "+ old_mood;
    }
```

```
}
```

veryHappyObject متد getMood() را جایگزین کرده است. با این حال دستور super.getMood() اجازه می‌دهد این کلاس از متد getMood() والد خود استفاده کند. درواقع veryHappyObject متد getMood() والد خود را با انجام تغییراتی و با دریافت مقدار برگشتی از super.getMood() ویژه و اختصاصی کرده است.

بنابراین اگر کلاس خلفی، متدهایی از کلاس والد خود را جایگزین نماید، همچنان می‌تواند از کدهای نوشته شده در کلاس والد خود استفاده نماید.

همچون تابع سازنده، اگر از super.<method> برای فراخوانی متدی از کلاس والد استفاده می‌نمایید، باید همه آرگومانهای لازم در متد کلاس والد را قید نمایید.

استفاده از مفهوم super در این کارگاه مفید خواهد بود.

حال اگر احساس می‌کنید همه آنچه را که باید بدانید، می‌دانید شروع به حل کارگاه نمایید. اما اگر احساس می‌کنید به کمک بیشتری نیاز دارید، مطالب زیر را نیز مرور کنید.

توسعه تعریف مسأله

اگر احساس می‌کنید به راهنمایی بیشتری نیاز دارید، موارد زیر را در نظر بگیرید.

کلاس BankAccount باید شامل متدهای زیر باشد:

```
public void depositFunds(double amount);
public double getBalance();
public double withdrawFunds(double amount);
protected void setBalance (double newBalance);
```

کلاس SavingsAccount باید شامل متدهای زیر باشد:

```
public void addInterest();
public void setInterestRate(double interestRate);
public double getInterestRate();
```

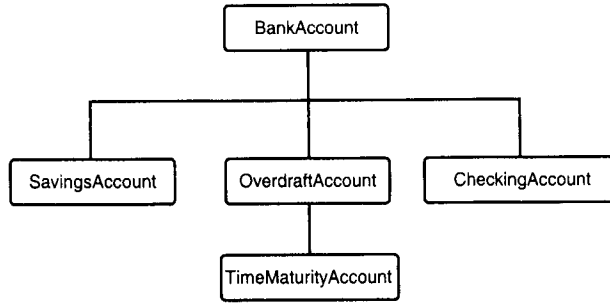
کلاس TimedMaturityAccount باید شامل متدهای زیر باشد:

```
public boolean isMature();
public void mature();
public double getFeeRate();
public void setFeeRate (double rate);
```

کلاس TimedMaturityAccount باید متد withdrawFunds() را دوباره تعریف نماید تا نسبت به محاسبه موعد و مقدار کارمزد بتواند اقدام نماید.

کلاس CheckingAccount باید متدهای زیر را شامل شود:

```
public void accessFees();
public double getFee();
public void setFee(double fee);
public int getMonthlyQuota();
public void setMonthlyQuota(int quota);
public int getTransactionCount();
```



شکل ۵ـ۲
سلسله مراتب کلاسهای
حسابهای بانکی

کلاس CheckingAccount نیز باید متد withdrawFunds() را برای شمارش تعداد نقل و انتقالات جایگزین نماید. کلاس OverdraftAccount باید شامل متدهای زیر باشد:

```
public void chargeInterest();
public double getCreditRate();
public void setCreditRate (double rate);
```

اگر کلاس BankAccount نسبت به میزان وجهی که مشتری از حساب خارج می‌کند واکنش نشان دهد (به عبارت دیگر نگذارد مشتری بیش از موجودی، از حساب خارج کند) کلاس OverdraftAccount باید متد withdrawFunds() را جایگزین نماید.

در ضمن می‌توانید از سلسله مراتب کلاسهایی که در روز سوم و کارگاه ۲ ایجاد کرده‌اید کمک بگیرید. تنها تغییری که باید اعمال نمایید در مورد متد withdrawFunds() می‌باشد.

شکل ۵ـ۲ سلسله مراتب کلاسهای ذکر شده در قبل را نشان می‌دهد.

توجه بخش بعدی راه حل کارگاه ۳ را ارایه می‌کند. بنابراین تا زمانی که نسبت به تکمیل کارگاه اقدام نکرده‌اید، به این بخش مراجعه نکنید.

حل و بحث

به خاطر سپردن سلسله مراتب فوق به هنگام پیاده‌سازی کلاسها و دنبال کردن راه حل بسیار حائز اهمیت است.

لیست ۵ـ۷ نحوه پیاده‌سازی کلاس BankAccount را نشان می‌دهد. کلاس پایه زیر میزان موجودی را در خود نگه داشته و برداشت‌ها و واریزها را مدیریت می‌کند.

لیست ۵-۷ BankAccount.java

```
public class BankAccount {

    // private data
    private double balance;

    // constructor
```

لیست ۵-۷ (ادامه)

```
    public BankAccount( double initDeposit ) {
    setBalance( initDeposit );
    }
    // deposit monies into account
    public void depositFunds( double amount ) {
    // the base class applies no policy
    // does not validate input
    setBalance( getBalance() + amount );
    }
    // query the balance
    public double getBalance() {
    return balance;
    }
    // set the balance
    protected void setBalance( double newBalance ) {
    balance = newBalance;
    }
    // withdraw funds from the account
    public double withdrawFunds( double amount ) {
        if( amount >= balance ) {
            amount = balance;
        }
    setBalance( getBalance() - amount );

    return amount;
    }

}
```

در لیست ۵ـ۸ SavingsAccount مستقیماً از BankAccount مشتق شده است.

کلاس SavingsAccount با اضافه کردن متدهای get و set برای محاسبه نرخ سود، کلاس BankAccount را برای خود ویژه کرده است.

لیست ۵-۸ SavingsAccount.java

```
public class SavingsAccount extends BankAccount {

    // private data
    private double interestRate;

    // Creates new SavingsAccount
    public SavingsAccount( double initBalance, double interestRate ) {
        super( initBalance );
        setInterestRate( interestRate );
    }
```

لیست ۵-۸ (ادامه)

```
    // calculate and add interest to the account
    public void addInterest() {
        double balance  = getBalance();
        double rate     = getInterestRate();
        double interest = balance * rate;

        double new_balance = balance + interest;

        setBalance( new_balance );
    }
    // set the interest rate
    public void setInterestRate( double interestRate ) {
        this.interestRate = interestRate;
    }
    // query the interest rate
    public double getInterestRate() {
        return interestRate;
    }

}
```

کلاس TimeMaturityAccount از کلاس SavingsAccount مشتق شده است چرا که نرخ سود به موجودی آن اعمال می‌گردد. برای این منظور باید کلاس والد خود را از طریق تعریف متدهایی برای تعیین میزان و سطح پول پرداختی و موعد آن تغییر دهد. نکته جالب توجه آن است که این کلاس متد withdrawFunds() را مجدداً تعریف می‌کند. اگرچه می‌توان از طریق فراخوانی عبارت super.withdrawFunds() همان کارکرد اصلی در کلاس والد را استفاده کرد. آنچنانکه خواهید دید برای نقل و انتقالات مالی در این حساب (و احیاناً پرداخت مبلغی جهت انجام این نقل و انتقالات) ملاحظاتی در کلاس در نظر گرفته شده است.

لیست ۵-۹ TimeMaturityAccount.java

```
public class TimedMaturityAccount extends SavingsAccount {

    // private data
    private boolean mature;
    private double  feeRate;

    // Creates new TimedMaturityAccount
    public TimedMaturityAccount(double initBalance, double interestRate, double feeRate ) {
        super( initBalance, interestRate );
        setFeeRate( feeRate );
    }
    // override BankAccount's withdrawFunds
    public double withdrawFunds( double amount ) {
        super.withdrawFunds( amount );
        if( !isMature() ) {
            double charge = amount * getFeeRate();
            amount = amount - charge;
        }
```

لیست ۵-۹ (ادامه)

```
    return amount;
    }
    // check maturity
    public boolean isMature() {
        return mature;
    }
    // make mature
    public void mature() {
        mature = true;
    }
    // % fee for early withdraw
    public double getFeeRate() {
        return feeRate;
    }
    // set % fee for early withdraw
    public void setFeeRate( double rate ) {
        feeRate = rate;
    }

}
```

در لیست ۵ـ۱۰، کلاس CheckingAccount به طور مستقیم از کلاس پایه BankAccount مشتق شده است. این کلاس متدهای لازم جهت محاسبه هزینه هر انتقال را به کلاس افزوده است. همچنین متد withdrawFunds() برای دنبال کردن تعداد نقل و انتقالات، جایگزین شده است. همچون کلاس TimedMaturityAccount، این کلاس نیز با فراخوانی super.withdrawFunds() همان منطق را دنبال می‌کند.

لیست ۵-۱۰ CheckingAccount.java

```
public class CheckingAccount extends BankAccount {

    // private data
    private int    monthlyQuota;
    private int    transactionCount;
    private double fee;

    // Creates new CheckingAccount
    public CheckingAccount( double initDeposit, int trans, double fee ) {
        super( initDeposit );
        setMonthlyQuota( trans );
        setFee( fee );
    }
    // override BankAccount's withdrawFunds
    public double withdrawFunds( double amount ) {
        transactionCount++;
    return super.withdrawFunds( amount );
    }
```

لیست ۵-۱۰ (ادامه)

```
    // access fees if went over transaction limit
    public void accessFees() {
        int extra = getTransactionCount() - getMonthlyQuota();
        if( extra > 0 ) {
            double total_fee = extra * getFee();
            double balance = getBalance() - total_fee;
            setBalance( balance );
        }
        transactionCount = 0;
    }
    // some getters and setters
    public double getFee() {
        return fee;
    }
    public void setFee( double fee ) {
        this.fee = fee;
    }
    public int getMonthlyQuota() {
        return monthlyQuota;
    }
    public void setMonthlyQuota( int quota ) {
        monthlyQuota = quota;
    }
    public int getTransactionCount() {
        return transactionCount;
    }

}
```

لیست ۵ـ۱۱ نشان می‌دهد چگونه کلاس OverdraftAccount به طور مستقیم از کلاس BankAccount مشتق شده است. در این کلاس تعدادی متد برای اعمال نرخ بهره افزوده شده است.

لیست ۵-۱۱ OverdraftAccount.java

```
public class OverdraftAccount extends BankAccount {

    // private data
    private double creditRate;

    // Creates new OverdraftAccount
    public OverdraftAccount( double initDeposit, double rate ) {
        super( initDeposit );
        setCreditRate( rate );
    }
    // charge he interest on any lent monies
    public void chargeInterest() {
```

لیست ۵-۱۱ (ادامه)

```
        double balance = getBalance();
        if( balance < 0 ) {
            double charge = balance * getCreditRate();
            setBalance( balance + charge );
        }
    }
    // query the credit rate
    public double getCreditRate() {
        return creditRate;
    }
    // set the credit rate
    public void setCreditRate( double rate ) {
        creditRate = rate;
    }

    // withdraw funds from the account
    public double withdrawFunds( double amount ) {
    setBalance( getBalance() - amount );

    return amount;
    }
}
```

هریک از کلاسهای ذکر شده در بالا به نحوی خصوصیات و رفتار کلاس والد خود را تغییر داده‌اند. تعدادی از آنها نظیر SavingsAccount تنها به اضافه کردن تعدادی متد پرداخته‌اند حال آنکه تعدادی دیگر نظیر ChekingAccount، OverdraftAccount و TimeMaturityAccount رفتار پیش‌فرض والد خود را به کلی عوض کرده‌اند.

کارگاه ٤: مطالعه موردی: همانی، مالکیت و java.util.Stack

به عنوان یک مبتدی در زمینه OO ممکن است فکر کنید که Java مثالی از یک طراحی شیءگرای کامل است. ممکن است به خودتان گفته باشید «اگر Java آن را انجام داده است، پس کارش صحیح بوده است.» متأسفانه اطمینان بی‌پرسش به هر طراحی OO بسیار خطرناک است.

بیایید نگاهی دوباره به ساختمان داده‌ای کلاسیک پشته (Stack) داشته باشیم. از این طریق می‌توان آیتمهایی را در پشته قرارداد (push) یا آیتمی را برداشت (pop) و یا بدون برداشتن و حذف آیتمی به آیتم دسترسی داشت. همچنین ممکن است پشته را از جهت خالی بودن آزمایش کنید.

Java دارای کلاسی برای پشته است. لیست ۵ـ۱۲ این امر را نشان می‌دهد.

```
public class Stack extends vector{
    public boolean empty();
    public Object peek();
    public Object pop();
    public Object push(Object Item);
    public int search(Object o);
}
```

ممکن است متوجه شده باشید که کلاس پشته در Java با تعریف کلاسیک آن اندکی متفاوت است. درواقع Java متد ()search را به آن افزوده است. همچنین متد ()push شیئی را که به پشته افزوده‌اید به عنوان خروجی تابع برمی‌گرداند. با این حال مشکل بزرگی وجود دارد. کلاس پشته در Java از vector مشتق می‌گردد. از یک دید این کار یک تصمیم هوشمندانه است. با وراثت از vector، کلاس پشته همه خصوصیات پیاده‌سازی شده در vector را به ارث می‌برد. برای پیاده‌سازی stack تنها کافی است که در بدنه متدهای آن به درستی متدهای vector را صدا بزنیم.

متأسفانه مثال کلاس پشته در Java، تعریفی ضعیف از وراثت است. آیا کلاس stack آزمایش همانی را پشت سر گذاشته است؟ آیا پشته یک vector است؟ خیر و در نتیجه آزمایش منفی است. vector شامل همه متدهایی است برای قرار دادن آیتمی در یک بردار و حذف آن. در حالی که پشته تنها اجازه می‌دهد آیتمها بر روی یکدیگر قرار گیرند. بردار اجازه می‌دهد که عناصر را هر جا که لازم باشد قرار داد و یا آن را حذف کرد.

در اینجا، وراثت اجازه می‌دهد که با کلاس پشته از روش غیرمعمول و تعریف نشده برای Stack در تعامل بود.

تعریف مسأله

پشته آزمایش مالکیت را پشت سر می‌گذارد. درواقع یک پشته شامل یک بردار است. برای این کارگاه، نسخه جدیدی از پشته را بنویسید که نحوه درست پیاده‌سازی و استفاده مجدد را مورد استفاده قرار دهد.

توجه بخش بعدی راه حل کارگاه ۴ است. تا زمانی که کارگاه ۴ را حل نکرده‌اید به این قسمت مراجعه نکنید.

حل و بحث

لیست ۵ـ۱۳ یک راه حل ممکن برای پیاده‌سازی پشته را نشان می‌دهد.

لیست ۵-۱۳ پیاده‌سازی جدید پشته

```
public class Stack {

    private java.util.ArrayList list;

    public Stack() {
        list = new java.util.ArrayList();
    }

    public boolean empty() {
        return list.isEmpty();
    }

    public Object peek() {
        if( !empty() ) {
            return list.get( 0 );
        }
        return null;
```

لیست ۵-۱۳ (ادامه)

```
    }

    public Object pop() {
        if( !empty() ) {
            return list.remove( 0 );
        }
        return null;
    }

    public Object push( Object item ) {
        list.add( 0, item );
        return item;
    }

    public int search( Object o ) {
      int index = list.indexOf( o );
      if( index != -1 ) {
          return index + 1;
      }
        return -1;
    }

}
```

اگر این کارگاه چیزی به شما آموخته باشد، آن این است که هیچ پیاده‌سازی کامل و جامع نیست!

خلاصه

امروز چهار کارگاه را انجام دادید. کارگاه ۱ شما را با وراثت به صورت ساده آشنا کرد. پس از اتمام کارگاه ۱ باید سازوکار و اصول وراثت را فراگرفته باشید. کارگاه ۲ مفهوم کلاس پایه مجرد و وراثت طرح‌ریزی شده را عنوان کرد. کارگاههای ۱ و ۲ نحوه تعریف دوباره و کار با صفات و متدهای تازه و جدید و بازگشتی را نشان دادند. همچنین ملاحظه کردید که چگونه می‌توان با جایگزین کردن یک متد به پیاده‌سازی همان متد در کلاس والد دسترسی داشت.

کارگاه ۴ مفاهیم بسیار مهم همانی(بودن) و مالکیت(داشتن) را نشان داد. گاهی اوقات بهترین کار مشتق نکردن است! همانگونه که در درس روز ۴ تأکید کردیم، اغلب اوقات ترکیب کلاسها بهترین و واضح‌ترین روش برای استفاده مجدد است.

وراثت دو شیء درواقع نشانگر رابطه همانی (Is – a) بین آن دو شیء است. اگر دو شیء رابطه‌ای از لحاظ نوع با یکدیگر نداشته باشند نباید آنها را از هم مشتق کرد. پیاده‌سازی مشترک دلیل کافی برای وراثت نیست.

در یک سیستم یا برنامه کاربردی تا آنجا که ممکن است باید از وراثت استفاده کرد. با این حال اگر برای کاربرد خاصی برنامه می‌نویسید، تنها محدود به همان برنامه هستید. ولی در مدت زمان طولانی بر روی

برنامه‌های متعدد و متفاوتی کار خواهید کرد. در این حالت پیش می‌آید که کار خاصی را بارها و بارها به صورت تکراری انجام می‌دهید. در اینگونه مواقع بهتر است سلسله مراتب وراثت را به نحوی کشف و استخراج نمایید و به نحو شایسته‌ای از آن در برنامه‌هایتان استفاده نمایید.

پرسشها و پاسخها

در کارگاه ۴، اشاره کردید که چگونه Java دچار اشتباهاتی در OO شده است. زمانی که توابع Java API و یا دیگر منابع و مثالهای را مرور می‌کنم، چگونه می‌توانم بفهمم که کدام پیاده‌سازی خوب است؟

بسیار مشکل است که بگوییم کدام پیاده‌سازی در OO خوب و کدام یک بد است. مگر آنکه تجربیات بسیاری در OO کسب کرده باشید. بهترین راه آن است که تمام آموخته‌های خود را به کار بندید و هیچوقت بر یک مثال تکیه نکنید.

کارگاه

سوالات این بخش برای فهم بیشتر شما مطرح گشته‌اند.

پرسشها

۱. **با استفاده از راه حلهای ارایه شده برای کارگاه مثالی از متد، تعریف مجدد، متد جدید و متد بازگشتی بیاورید.**

۲. **چرا کلاس را از انواع مجرد تعریف می‌کنیم؟**

۳. **در کارگاه ۴ با مفهوم همانی و مالکیت برخورد کردید. قبل از آنکه مفهوم وراثت را فرا بگیرید با رابطه همانی (Is – a) آشنا شدید. چه روابطی از نوع مالکیت در کارگاههای روز سوم مشاهده نمودید؟**

٤. **چگونه کارگاههای ارایه شده کپسوله‌سازی را مابین زیرکلاسها و کلاسهای پایه حفظ می‌کنند؟**

۵. **از روی راه حل ارایه شده، مثالی از ویژه کردن کلاس والد بیاورید.**

۶. **چگونه راه حلهای کارگاههای ۳ و ۴ دو روش متفاوت جهت استفاده مجدد (reuse) را ارایه می‌کنند؟**

تمرین‌ها

امروز تمرینی نداریم. به کارگاهها برسید!

روز ۶

Object Oriented Programming

چند شکلی بودن: بیاموزیم آینده را پیش‌بینی کنیم

تا به حال دو رکن از سه رکن اساسی برنامه‌نویسی شیءگرا را با هم بررسی کردیم. کپسوله‌سازی و وراثت. چنانکه می‌دانید کپسوله‌سازی برنامه‌نویس را قادر می‌سازد که اشیاء نرم‌افزاری خودبسنده ایجاد کند و وراثت برای استفادهٔ مجدد از کد و توسعهٔ اشیاء مزبور کاربرد دارد. با این حال هنوز چیزی کم است. نرم‌افزار همواره تغییر می‌کند. گاهی کاربران تقاضاهای جدیدی مطرح می‌کنند، گاهی خطا یافت می‌شود، یا لازم می‌شود برنامه در محیط جدیدی به کار گرفته شود. دورهٔ توسعه یک نرم‌افزار وقتی آن را برای فروش عرضه می‌کنند پایان نمی‌پذیرد. نرم‌افزاری که می‌نویسید باید بتواند خود را برای پاسخگویی به نیازهای آینده تطبیق دهد. عالی نیست اگر بتوانید نرم‌افزاری برای آینده بنویسید؟

چنین نرم‌افزاری بدون نیاز به تغییر، با نیازهای آینده سازگار است. نرم‌افزار آینده نگر (Future Proof) برنامه‌نویس را قادر می‌سازد تا اعمال تغییرات و افزودن ویژگیهای جدید را به سادگی به انجام برساند. خوشبختانه روش برنامه‌نویسی شیءگرا آگاهانه طوری طراحی شده تا ساختارهای پویا را پشتیبانی کند. برای رسیدن به این هدف برنامه‌نویسی شیءگرا مفهوم چند شکلی بودن (Polymorphism) را معرفی می‌کند.

در دو روز آتی با هم به بررسی چند شکلی بودن، آخرین رکن اساسی شیءگرا، خواهیم پرداخت.

آنچه امروز خواهید آموخت

- چندشکلی بودن یعنی چه؟
- انواع مختلف چندشکلی کدامند و هرکدام چه خاصیتی دارند؟
- چند توصیه برای اجرای صحیح چندشکلی
- نقطه ضعفهای چندشکلی بودن
- چگونه چندشکلی بودن اهداف برنامه‌نویسی شیءگرا را تأمین می‌کند.

چندشکلی

اگر کپسوله‌سازی و وراثت را زیر یک خم گرفتن و پل رفتن در نظر بگیریم، در این صورت چندشکلی بودن هم فن نهایی است که حریف را ضربه می‌کند. بدون دو رکن اولیه، نمی‌توان چندشکلی بودن داشت و بدون چندشکلی بودن، تکنیک شیءگرا کامل نخواهد بود. وقتی نوبت به چندشکلی بودن می‌رسد، الگوی شیءگرا خود را نشان می‌دهد. مهارت در اجرای چندشکلی بودن برای برنامه‌نویسی شیءگرای صحیح و اصولی واجب است.

اگر بخواهیم با اصطلاحات برنامه‌نویسی صحبت کنیم، چندشکلی بودن فرایند خودکاری است که برحسب شرایط یکی از چند کد موجود را به یک نام کلاس یا روال نسبت می‌دهد. یعنی یک نام واحد می‌تواند تعیین کنندهٔ چندین رفتار مختلف باشد.

واژه جدید

چندشکلی بودن یعنی اینکه یک موجودیت، دارای اشکال مختلفی باشد. این موضوع در اصطلاح برنامه‌نویسی یعنی با استفاده از یک نام واحد بتوان رفتارهای مختلفی را در شرایط مختلف، آن هم به صورت خودکار، انتظار داشت.

اگر بخواهیم از منظر دیگری مسأله را بررسی کنیم، می‌توان گفت چندشکلی بودن مانند اختلال چندشخصیتی بودن در دنیای برنامه‌نویسی است. چون یک موجودیت، نوع رفتارهای متفاوتی از خود بروز می‌دهد.

آنچه در مورد چندشکلی بودن گفتیم ممکن است اندکی غیرعملی به نظر برسد. اصطلاح «باز کردن» را در نظر بگیرید. می‌توان یک در، یا یک جعبه، یا پنجره و یا یک حساب بانکی باز کرد. مفهوم باز کردن در بسیاری از موارد و در رابطه با خیلی چیزها به کار می‌رود. هر شیء نوع باز کردن خاص و متفاوتی دارد. با این حال در تمام موارد می‌توان به راحتی از اصطلاح «باز کردن» برای شرح عملیات استفاده کرد.

تمام زبان‌ها، چندشکلی بودن را پشتیبانی نمی‌کنند. زبان‌هایی که چنین خاصیتی دارند را زبان‌های چندشکلی (Polymorphic) می‌نامند. در مقابل زبان‌های تک شکلی (Monomorphic) وجود دارند، که هر مفهومی را فقط به یک واحد نسبت می‌دهند.

وراثت زمینه را برای ایجاد برخی انواع چندشکلی بودن فراهم می‌کند. در درس روز چهارم، دیدید که وراثت چگونه امکان تشکیل روابط جانشین‌پذیر را فراهم می‌کند. اتصال‌پذیری برای چندشکلی بودن بسیار مهم است. چون اجازه می‌دهد با یک نوع دادهٔ خاص به صورت کلی رفتار کرد.

کلاسهای زیر را در نظر بگیرید:

```
public class PersonalityObject {
    public String speak() {
        return "I am an object.";
    }
}

public class PessimisticObject extends PersonalityObject {
    public String speak() {
        return "The glass is half empty.";
    }
}

public class OptimisticObject extends PersonalityObject {
    public String speak() {
        return "The glass is half full.";
    }
}

public class IntrovertedObject extends PersonalityObject {
    public String speak() {
        return "hi...";
    }
}

public class ExtrovertedObject extends PersonalityObject {
    public String speak() {
        return "Hello, blah blah blah, did you know that blah blah blah.";
    }
}
```

این کلاسها یک سلسله مراتب وراثت یکدست را شکل می‌دهند. کلاس پایه یک روال speak() تعریف می‌کند. هر زیرکلاس speak() را خود دوباره تعریف می‌کند و بر اساس شخصیت خودش، پیغام خاصی را برمی‌گرداند. سلسله مراتب قابلیت جانشین‌پذیری را بین انواع فرعی (زیرنوع‌ها) و والد آنها شکل می‌دهد.

تابع زیر را در نظر بگیرید:

```
public static void main( String [] args ) {
    PersonalityObject personality = new PersonalityObject();
    PessimisticObject pessimistic = new PessimisticObject();
    OptimisticObject  optimistic  = new OptimisticObject();
    IntrovertedObject introverted = new IntrovertedObject();
    ExtrovertedObject extroverted = new ExtrovertedObject();

    // substitutability allows you to do the following
    PersonalityObject [] personalities = new PersonalityObject[5];
    personalities[0] = personality;
```

```
    personalities[1] = pessimistic;
    personalities[2] = optimistic;
    personalities[3] = introverted;
    personalities[4] = extroverted;

    // polymorphism makes PersonalityObject seem to have many different behaviors
    // remember - polymorphism is the multiple personalities disorder of the OO world
    System.out.println( "PersonalityObject[0] speaks: " + personalities[0].speak());
    System.out.println( "PersonalityObject[1] speaks: " + personalities[1].speak());
    System.out.println( "PersonalityObject[2] speaks: " + personalities[2].speak());
    System.out.println( "PersonalityObject[3] speaks: " + personalities[3].speak());
    System.out.println( "PersonalityObject[4] speaks: " + personalities[4].speak());
}
```

شکل ۶ـ۱ خروجی برنامه را نشان می‌دهد.

خروجی نشان می‌دهد که روال speak() شیء PersonalityObject رفتارهای متفاوتی در پیش می‌گیرد. با اینکه آرایه حاوی عناصر PersonalityObject است، رفتار هر عنصر آرایه در صداکردن روال speak() متفاوت است، یعنی یک نام واحد، رفتارهای متفاوتی نشان می‌دهد.

واژه جدید

متغیر چندشکلی (Polymorphic Variable)، متغیری است که می‌تواند انواع داده مختلفی به خود بگیرد personalities مثالی از متغیر چندشکلی است.

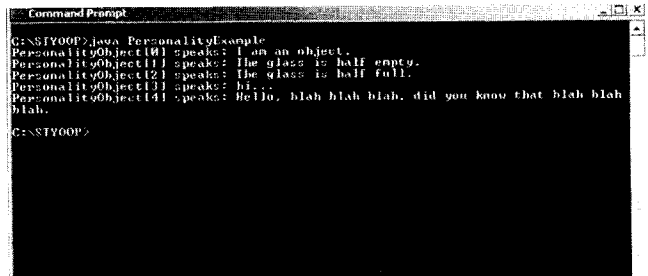
نکته

در یک زبان برنامه‌نویسی دارای نوع داده، متغیرهای چندشکلی برای نگه داشتن داده‌ها در یک رابطه جانشین‌پذیر ایجاد می‌شوند. در زبان‌های دارای نوع داده پویا، متغیرهای چندشکلی می‌توانند هر مقداری را بپذیرند.

مثال قبلی روال کار را نشان می‌دهد، اما ممکن است نتواند روح مطلب را برساند. به هر صورت شما می‌دانید که آرایه دقیقاً چه نوع داده‌هایی را در بر دارد. فرض کنید، شیئی دارید که روال آن یک متغیر PersonalityObject را به عنوان پارامتر می‌پذیرد.

```
public void makeSpeak (PersonalityObject obj) {
    System.out.println(obj.Speak());
}
```

روابط جانشین‌پذیری مجاز می‌دارند که متغیری از نوع PersonalityObject یا هریک از اخلاف آن را بتوان به makeSpeak() ارسال کرد. بنابراین، وقتی فرزند تخصیص یافته‌ای از PersonalityObject می‌سازید، برای



شکل ۶ـ۱
نمایش رفتار چندشکلی بودن

استفاده از آن در ()makeSpeak به عنوان آرگومان نیازی به ایجاد تغییر نخواهید داشت.

چندشکلی بودن از همین جا شروع به کار می‌کند. چند شکلی بودن این اطمینان را ایجاد می‌کند که در چنین شرایطی روال صحیح فراخوانده شود، نه روالی که کلاس صداکننده تصور می‌کند صداکرده. یعنی در صورتی که متغیری از نوع ExtrovetedObject به ()makeSpeak پاس شود، روال ()Speak خود آن فراخوانده شود و نه از آن PersonalityObject که ()makeSpeak تصور می‌کند با آن سروکار دارد.

از چند شکلی بودن به همین صورت می‌توان برای افزودن کارایی‌های جدید به سیستم در زمان نیاز استفاده برد. می‌توانید کلاسهای جدیدی با خواصی جدید مشتق کنید. خواصی که در زمان طراحی اولیه خوابشان را هم نمی‌دیدید و سپس از این کلاسهای جدید بدون نیاز به تغییر کد قبلی استفاده کنید. طراحی نرم‌افزار آینده‌نگر چنین چیزی است.

مثال اخیر فقط به منزله نوک کوه یخ چندشکلی بودن است. زیرا تنها یکی از اشکال متعدد آن را مورد بررسی قرار داده است. خود چندشکلی بودن هم در عمل چندشکلی است!

متأسفانه به چند شکلی بودن تا به حال به حد کافی پرداخته نشده است. در این کتاب هم ما خود را به چهار نوع آن محدود می‌کنیم. شناختن و درک توانایی‌هایی این چهار نوع اولیه، به شما زمینهٔ لازم برای استفاده از چند شکلی بودن در برنامه‌هایتان را می‌دهد. امروز درباره این چهار نوع خواهیم آموخت:

۱. چند شکلی بودن از نوع دربرداشتن (درونی – Inclusion)
۲. چندشکلی بودن پارامتری (Parametric)
۳. جایگذاری (Overriding)
٤. سربارگذاری (Overloading)

چندشکلی بودن درونی

این نوع که گاهی چندشکلی بودن خالص خوانده می‌شود، برنامه‌نویس را مجاز می‌دارد که با اشیاء به صورت کلی رفتار کند. نوع درونی را امروز در ابتدای همین فصل مشاهده کردید. روال زیر را در نظر بگیرید:

```
public void makeSpeak (PessimiticObject obj){
    System.out.pring (obj.speak());
}
```

و فرض کنید برای هریک از سه نوع دیگر فرزندان همردهٔ PessimisticObject کد مشابهی داشته باشیم. اما تمام این فرزندان به هم مربوط هستند، چون از یک کلاس واحد مشتق شده‌اند. جانشین‌پذیری و چندشکلی بودن درونی اجازه می‌دهند که برای تمام انواع PersonalityObject روال واحدی نوشت:

```
Public void makeSpeak (PersonalityObject obj){
    System.out.println(obj.speak());
}
```

جانشین‌پذیری امکان پاس کردن تمام انواع PersonalityObject به روال را فراهم می‌کند. چندشکلی بودن هم فراخوانی روالهای مناسب برای هر نوع ورودی را تضمین می‌کند. زیرا بر اساس نوع حقیقی (true–type) متغیر عمل می‌کند، نه بر اساس نوع ظاهری آن (PersonalityObject).

خاصیت مهم چندشکلی بودن درونی، کاهش کدنویسی مورد نیاز است. زیرا به جای نوشتن چندین کد، هر یک برای یک نوع داده، برای تمام انواع، یک کد واحد نوشته می‌شود. ترکیب جانشین‌پذیری و چندشکلی بودن درونی به makeSpeak() اجازه می‌دهند که با هر شیئی که PersonalityObject باشد، یعنی با آن رابطه همانی داشته باشد، کار کند.

چند شکلی بودن درونی افزودن انواع دادهٔ جدید را ساده‌تر می‌کند. زیرا نیازی به افزودن روالهای جدید مخصوص نوع جدید نخواهد بود.

فایدهٔ دیگر چندشکلی بودن درونی آن است که با استفاده از آن به نظر می‌رسد که متغیرهای از نوع PersonalityObject رفتارهای بسیار متفاوتی را از خود نشان می‌دهند. پیغام نمایش داده شده توسط makeSpeak() بر اساس نوع ورودی آن تفاوت خواهد کرد. با استفاده دقیق از چندشکلی بودن درونی می‌توان عملکرد سیستم را بهبود بخشید. بهترین قسمت این قضیه آن است که کارایی جدید بدون نیاز به تغییر کد موجود و فقط با مشتق کردن حاصل می‌شود.

چندشکلی بودن، دلیلی است که ثابت می‌کند وراثت فقط به کار استفاده مجدد از کد نمی‌آید. بلکه مهمترین کاربرد آن، ایجاد توانایی چندشکلی بودن از طریق روابط جانشین‌پذیری است. در این صورت استفاده مجدد هم در دنباله به دست خواهد آمد. زیرا خواهید توانست از کدهای کلاس اصلی، فرزندان و روالهای استفاده کننده از آنها، دوباره استفاده کنید.

احتمالاً تا به حال سازوکار سیستم را درک کرده‌اید. اما چرا باید بخواهید از چندشکلی بودن درونی استفاده کنید؟ سلسله مراتب گزارش‌دهی (Logging) زیر را در نظر بگیرید:

```
public abstract class BaseLog {

    // some useful constants, don't worry about the syntax
    private final static String DEBUG   = "DEBUG";
    private final static String INFO    = "INFO";
    private final static String WARNING = "WARNING";
    private final static String ERROR   = "ERROR";
    private final static String FATAL   = "FATAL";

    java.text.DateFormat df = java.text.DateFormat.getDateTimeInstance();

    public void debug( String message ) {
        log( message, DEBUG, getDate() );
    }
    public void info( String message ) {
        log( message, INFO, getDate() );
    }
    public void warning( String message ) {
        log( message, WARNING, getDate() );
    }
    public void error( String message ) {
        log( message, ERROR, getDate() );
    }
```

```
    public void fatal( String message ) {
        log( message, FATAL, getDate() );
    }

    // creates a time stamp
    protected String getDate() {
        java.util.Date date = new java.util.Date();
        return df.format( date );
    }

    // let subclasses define how and where to write log to
    protected abstract void log( String message, String level, String time );
}
```

Baselog یک گزارش مجرد است که علاوه بر رابط عمومی گزارش، کمی هم به پیاده‌سازی پرداخته است. تجرد کلاس به آن دلیل است که هر پیاده‌سازی باید حاوی کد روش و محل نمایش گزارش باشد. هر پیاده‌سازی باید کد log() را جداگانه تعریف کند.

با تجرید کلاس، می‌توان مطمئن بود که هر پیاده‌ساز زیرکلاسها را به صورتی صحیح پیاده‌سازی می‌کند. چنین راهبردی اجازه می‌دهد تا از این طرح گزارش در برنامه‌های کاربردی مختلف با کاربردهای مختلف، بتوان مجدداً استفاده کرد. در هیچ موردی نیازی به طراحی مجدد گزارش نیست، فقط در هر کاربرد باید کد مناسب را پیاده‌سازی کرد.

```
public class FileLog extends BaseLog {

    private java.io.PrintWriter pw;

    public FileLog( String filename ) throws java.io.IOException {
        pw = new java.io.PrintWriter( new java.io.FileWriter( filename ) );
    }

    protected void log( String message, String level, String time ) {
        pw.println( level + ": " + time + ": " + message );
        pw.flush();
    }

    public void close() {
        pw.close();
    }
}

public class ScreenLog extends BaseLog {
    protected void log( String message, String level, String time ) {
        System.out.println( level + ": " + time + ": " + message );
    }
}
```

FileLog و ScreenLog هر دو از BaseLog مشتق می‌شوند و خود روال ()log را پیاده‌سازی می‌کنند. Filelog گزارش را در فایل می‌ریزد، در حالی که ScreenLog آن را روی صفحه نمایش می‌دهد.

مثال Employee درس روز چهارم را یاد بیاورید. کلاسی را فرض کنید که بتواند Employeeها را از پایگاه داده استخراج کند.

```
public class EmployeeDatabaseAccessor(BaseLog log) throws InitDBException{
    private BaseLog error_log;
    public EmployeeDatabaseAccessor(BaseLog log) throws InitDBException{
        error_log=log;
        try{
            //initialize th db connection
        }catch(DBException ex){
            error_log.fatal("cannot access database:"+ex.getMessage());
            throw new InitDBException(ex.getMessage());
        }
    }

    public Employee retrieveEmployee(String first_name,String last_name) throws
EmployeeNotFoundException{
        try{
            //attempt to retrieve the employee
            return null;
        }catch(EmployeeNotFoundException ex){
            error_log.warning("cannot locate employee: "+last_name+", "+ first_name);
            throw new EmployeeNotFoundException(last_name,first_name);
        }
    }
    //and so on, each method uses error_log to log errors
}
```

EmployeeDatabaseAccessor یک متغیر BaseLog را به عنوان آرگومان در تابع سازنده می‌پذیرد. هر متغیر از نوع آن از log برای ثبت اتفاقات مهم استفاده می‌کند. تابع ()main زیر را در نظر بگیرید:

```
public static void main(String [] args){
    BaseLog log=new ScreenLog();
    EmployeeDatabaseAccessor eda=new EmployeeDatabaseAccessor(log);
    Employee emp=eda.retriveEmployee("Employee","Mr.");
}
```

()main می‌تواند هر زیرکلاس BaseLog را به EmployeeDatabaseAccessor پاس کند. زیرا این کلاس با دیدگاه آینده‌نگر طراحی شده و می‌تواند با هر logکار کند. چه logمزبور با فایل کار کند یا گزارش ۲۴ ساعته دهد. کسی چه می‌داند، logهای آینده چطور کار می‌کنند. اما با چند شکلی بودن درونی، برنامه‌نویس آماده هر تغییری است.

بدون چندشکلی بودن درونی، برای هر نوع logکه بخواهید از آن استفاده کنید، باید یک سازندهٔ جداگانه

بنویسید و این تمام ماجرا نیست. باید یک ساختار سوییچ (switch) در برنامه بگنجانید تا بتوانید بفهمید با چه نوع log سروکار دارید. کد زیر را مشاهده کنید:

```
public class EmployeeDatabaseAccessor{

    private FileLog file_log;
    private ScreenLog screen_log;
    private int log_type;

    //some 'usefull' constants
    private final static int FILE_LOG=0;
    private final static int SCREEN_LOG=1;

    public EmployeeDatabaseAccessor(FileLog log) throws InitDBException{
        file_log=log;
        log_type=FILE_LOG;
        init();
    }

    public EmployeeDatabaseAccessor(ScreenLog log) throws InitDBException{
        screen_log=log;
        log_toye=SCREEN_LOG;
        init();
    }

    public Employee rtrieveEmployee(String first_name, String last_name) throws
EmployeeNotFoundException{
        try{
            //attempt to retrieve employee
            return null;
        }catch(EmployeeNotFoundException ex){
            if (log_type==FILE_LOG){
                file_log.warning("cannot locate employee:"+
                    last_name+", "+first_name);
            }else if (log_type==SCREEN_LOG){
                screen_log.warning("cannot locate employee:"+
                    last_name+", "+first_name);
            }throws new EmployeeNotFoundException(last_name,first_name);
        }
    }

    private void init() throws InitDBException{
        try{
            //initialize the db connection
        }catch(DBException ex){
            if (log_type==FILE_LOG){
```

```
            file_log.fatal("cannot access database:"+
                ex.getMessage());
        }else if (log_type==SCREEN_LOG){
            screen_log.fatal("cannot access database:"+
                ex.getMessage());
        }
        throws new InitDBException(ex.getMessage());
    }
  }
  //and so on, each method uses error_log to log errors
}
```

در این مدل برای هر نوع جدید گزارش مجبورید کد مربوط به آن را جداگانه بنویسید. شما کدام مدل را ترجیح می‌دهید؟

چندشکلی بودن پارامتری

برای ایجاد انواع و روالهای کلی یا ژنریک به کار می‌رود. انواع و روالهای ژنریک این امکان را ایجاد می‌کنند که یکبار برنامه بنویسید و آن را برای انواع مختلف پارامترها به کار ببرید.

روالهای پارامتری

روش درونی شیوهٔ نگرش به اشیا را تغییر می‌داد، در حالی که نوع پارامتری خود روالها را تحت تأثیر قرار می‌دهد. چندشکلی بودن پارامتری با تأخیر انداختن تعیین انواع دادهٔ روال تا زمان اجرا، ایجاد روالهای ژنریک را ممکن می‌کند. روال زیر را در نظر بگیرید:

```
int add (int a , int b)
```

روال ()add دو عدد صحیح را می‌گیرد و حاصلجمع آنها را برمی‌گرداند. تعریف روال خیلی صریح است. روال دو عدد صحیح را به عنوان آرگومان می‌گیرد. نمی‌توان عدد حقیقی به آن پاس کرد یا دو ماتریس را توسط آن جمع زد. اگر چنین کاری کنید، در زمان ترجمه با خطا روبرو می‌شوید. اگر بخواهید دو عدد صحیح را جمع کنید یا دو ماتریس را، باید برای هریک از انواع روالهای جداگانه‌ای بنویسید:

```
Matrix add_matrix (matrix a, matrix b)
Real add_real (real a, real b)
```

به همین صورت هر نوع داده‌ای تابع ()add خاص خود را طلب می‌کند.

اگر بتوان از چنین کاری احتراز کرد خیلی بهتر است. چون اول اینکه نوشتن یک روال جداگانه برای هر نوع داده برنامه را بسیار بزرگ می‌کند. دوم اینکه کدنویسی بیشتر امکان مواجه شدن با خطا را بالاتر می‌برد. سوم اینکه نوشتن روالهای جداگانه، ()add را به صورتی طبیعی مدل نمی‌کند. طبیعی‌تر خواهد بود که بتوان فقط گفت ()add، نه matrix_add یا real_add.

چندشکلی بودن درونی راه حلی برای این معضل ارایه می‌دهد. می‌توان نوعی به نام Addable تعریف کرد که بداند چگونه یک متغیر از نوع خود را به دیگری بیافزاید. این نوع می‌تواند به صورت زیر باشد:

```
public abstract class Addable{
    public Addable add (Addable);
}
```

و روال جدید مانند زیر خواهد بود:

```
Addable add_addable (Addable a, Addable b)
    return a.add(b)
```

نکته از مثال قبلی گاهی تحت عنوان چندشکلی بودن تابعی (Function Polymorphism) یاد می‌شود.

با این کار فقط باید یک روال برای افزودن مقادیر نوشته شود، هرچند روال فقط برای آرگومانهای از نوع Addable کار می‌کند. همچنین باید مطمئن شوید که Addableهایی که به یک روال پاس می‌کنید از یک نوع واحد باشند. چنین راهبردی مستعد خطاست. به هر صورت با این کار مسأله اصلی حل نشده. هنوز هم باید برای هر نوع داده که با Addable متفاوت باشند روالهای جداگانه نوشته شود. اینجا جایی است که پای چندشکلی بودن پارامتری به میان کشیده می‌شود. زیرا به برنامه‌نویس امکان می‌دهد که یک و تنها یک روال واحد برای جمع کردن تمام انواع دادهٔ جمع‌پذیر بنویسد. چندشکلی بودن پارامتری تعریف انواع دادهٔ آرگومانها را به تأخیر می‌اندازد. با استفاده از روش پارامتری داریم:

```
add([T] a, [T] b): [T]
```

[T] آرگومانی مانند a و b است. آرگومان [T] نوع دادهٔ a و b را مشخص می‌کند. با تعریف روال از این طریق، تعریف انواع دادهٔ دخیل در عملیات تا زمان اجرا به تأخیر می‌افتد. توجه داشته باشید که a و b هر دو یک [T] داشته باشند. درون روال می‌تواند به این صورت باشد:

```
[T] add ([T] a, [T] b)
    return a+b;
```

چندشکلی بودن جادو نیست. آرگومانها باید ساختار خاصی داشته باشند. در چنین حالتی هر آرگومانی که پاس می‌شود، باید اپراتور + را برای خود تعریف کرده باشد.

نکته ساختار خاص یعنی حضور یک روال خاص یا وجود تعریف یک اپراتور خاص.

انواع پارامتری

دقیقاً همانند روالها، انواع داده هم می‌توانند پارامتری باشند. ADT صف از درس روز دوم را به یاد بیاورید:

```
Queue [T]
    void enqueue ([T])
    [T] dequeue()
    boolean isEmpty()
    [T] peak()
```

Queue نوعی پارامتری است. به جای اینکه برای هر نوع داده‌ای که می‌خواهید به صف کنید، کلاسی جداگانه بنویسید، نوع داده را به صورت پویا در زمان اجرا مشخص می‌کنید. قبلاً می‌توانستیم بگوییم صف ما، صفی

از اشیاء است. حال می‌توان گفت صفی از هر نوع داده ممکن است.

بنابراین اگر بخواهیم Employeeها را به صف کنیم، تعریف زیر را ایجاد می‌کنیم:

```
Queue [Employee] employee_queue = new Queue [Employee];
```

حال می‌توان به راحتی از روالهای ()enqueue و ()dequeue برای متغیرهای از نوع Employee استفاده کرد.

اگر انواع پارامتری در دسترس نبودند، برای هر نوع داده‌ای باید یک Queue جداگانه می‌نوشتیم. یکی برای اعداد صحیح، یکی برای اعداد حقیقی و...

در عوض با بهره‌گیری از انواع پارامتری، می‌توان یک بار نوعی را تعریف کرد (در این مورد صف) و از آن برای تمام انواع دادهٔ ممکن استفاده کرد.

نکته

چندشکلی بودن پارامتری به صورت تئوری بسیار قدرتمند به نظر می‌رسد. اما یک مشکل وجود دارد: پشتیبانی.

برای آنهایی که با Java آشنایی دارند، شاید مثال قبلی عجیب به نظر برسد. چون از Javaی نسخه ۱/۳ به بعد، انواع پارامتری یا به طور اعم چندشکلی بودن پارامتری ذاتاً پشتیبانی نمی‌شوند. می‌توان از انواع پارامتری استفاده کرد، اما قیمتی که در کاهش بهره‌وری می‌پردازید بسیار بالاست. فقط برخی نسخه‌های تأیید نشده توسط Sun این خاصیت را پشتیبانی می‌کنند.

ساختار مثال قبلی کاملاً غیرواقعی بود، اما ایدهٔ اصلی را به صورتی مناسب نشان می‌دهد.

جایگزینی

جایگزینی (overriding) نوع مهمی از چندشکلی بودن است. دیدید که چگونه هریک از زیر کلاسهای PersonalityObject روال ()speak را به صورتی متفاوت تغییر داد. اگر به یاد داشته باشید در درس روز پنجم مثال جالب‌تری از جایگزینی و چندشکلی بودن را با هم دیدیم. به صورت خاص، کلاسهای زیر را در نظر بگیرید:

```
public class MoodyObject{
    //return the Mood
    protected String getMood(){
        return "Moody";
    }

    //ask the object how it feels
    public void queryMood(){
        System.out.println("I feel "+getMood()+" today.");
    }
}

public class HappyObject extends MoodyObject{
    //redefine class's Mood
    protected String getMood(){
        return "happy";
```

```
    }

    //specialization
    public void laugh(){
        System.out.println("hehehe ... hehehe ... HAHAHAHAHAHA!!!");
    }
}
```

در اینجا می‌توان دید که HappyObject روال ()getMood را دستکاری می‌کند. جالب اینجاست که تعریف ()queryMood کلاس MoodyObject روال ()getMood را صدا می‌زند.

می‌توان مشاهده کرد که HappyObject روال ()queryMood را جایگزین نمی‌کند و آن را به عنوان روالی بازگشتی به صورت اصلی به ارث می‌برد. چندشکلی بودن در اینجا این اطمینان را فراهم می‌کند که روال دستکاری شدهٔ ()getMood فراخوانی شود، نه روال والد. یعنی لازم نیست خود برنامه‌نویس روال ()queryMood را هم در فرزند تغییر دهد تا نسخهٔ ()getMood مربوط به خود را صدا کند.

دیدید که چگونه می‌توان ()getMood والد را به صورت مجرد تعریف کرد.

```
//return the Mood
protected abstract String getMood();
```

روالهای مجرد، غالباً به نام روالهای معوق (deferred methods) خوانده می‌شوند. چون تعریف آنها به کلاس فرزند سپرده می‌شود. مانند هر روال دیگری، روالهای معوق را می‌توان در کلاسی که آنها را تعریف کرده صدا زد. چندشکلی بودن وظیفه تعیین نسخه‌ای از روال را دارد که باید اجرا شود.

سربارگذاری

با سربارگذاری یا چندشکلی بودن ویژه (ad-hoc) می‌توان از یک نام روال واحد برای شمار زیادی از روالهای متفاوت استفاده کرد. روالها فقط در تعداد و نوع پارامترها با هم متفاوتند.

روالهای زیر در java.lang.math تعریف شده‌اند:

```
public static int max (int a, int b);
public static long max (long a, int b);
public static float max (float a, int b);
public static double max (double a, int b);
```

روالهای ()max همگی مثالهایی از سربارگذاری هستند. می‌بینید که این روالها فقط در نوع داده ورودی با هم متفاوتند.

سربارگذاری وقتی به کار می‌آید که عملکرد روال مستقل از نوع داده ورودی آن باشد. روال ()max را در نظر بگیرید. ()max مفهومی کلی است که دو پارامتر را می‌گیرد و پارامتر دارای مقدار عددی بزرگتر را برمی‌گرداند. این تعریف صرفنظر از اینکه پارامترهای مورد مقایسه صحیح یا حقیقی هستند، ثابت است. عملگر + مثال دیگری از روالهای سربارگذاری شده است. مفهوم + مستقل از عملوندهای آن است. تمام انواع عناصر قابل جمع شدن با هم هستند.

نکته

در Java نمی‌توان عملگرها را جایگزین یا سربارگذاری کرد. Java خود دارای چند سربارگذاری درونی است.

اگر سربارگذاری میسر نبود، مجبور بودید چنین کاری کنید:

```
public static int max_int (int a, int b);
public static long max_long (long a, int b);
public static float max_float (float a, int b);
public static double max_double (double a, int b);
```

بدون سربارگذاری هر روالی باید نامی جداگانه می‌داشت. یعنی max() نمی‌توانست مستقل از نوع باشد و تبدیل به مفهومی مجرد می‌شد. یعنی نمی‌شد مفهوم بیشینه را به صورتی طبیعی مدل کرد. همچنین در این صورت برنامه‌نویس باید چیزهای بیشتری به ذهن می‌سپرد.

البته، نامگذاری مستقل روالها چندشکلی نیست. اگر تمام روالها دارای یک نام باشند، با رفتار چندشکلی مواجه خواهید شد. چون در پس زمینه روالهای مختلفی صدا زده می‌شوند. می‌توان به سادگی فقط max() را صدا زد و پارامترها را به آن پاس کرد. چندشکلی بودن خود فراخوانی روال مناسب را بر عهده می‌گیرد.

چگونگی عملکرد چندشکلی بودن بستگی به زبان برنامه‌نویسی دارد. برخی زبان‌ها فراخوانی روالها را در زمان ترجمه تفکیک می‌کنند و برخی آن را به زمان اجرا واگذار می‌کنند.

تحمیل

تحمیل (Coercion) و سربارگذاری اغلب دوشادوش هم عمل می‌کنند. تحمیل می‌تواند روالی را وادارد طوری عمل کند که گویی چندشکلی است. تحمیل وقتی رخ می‌دهد که در پشت صحنه تبدیل نوع در آرگومان صورت پذیرد. تعریف زیر را در نظر بگیرید:

```
public float add (float a, float b);
```

add() دو پارامتر ممیز شناور را می‌گیرد و آنها را با هم جمع می‌کند. حال کد زیر را در نظر بگیرید:

```
int iA = 1;
int iB = 2;
add (iA,iB);
```

با وجود تعریف add() که دو پارامتر ممیز شناور می‌پذیرد در اینجا دو پارامتر صحیح به آن پاس شده است. اینجا، همان جایی است که تحمیل وارد عرصه می‌شود.

مترجم (compiler) متغیرهای صحیح را به ممیز شناور تبدیل می‌کند. یعنی قبل از پاس شدن به روال add() آرگومان‌ها به نوع مطلوب تبدیل می‌شوند. به اصطلاح برنامه‌نویسان cast می‌شوند.

بنابراین تحمیل باعث می‌شود add() چندشکلی به نظر برسد، در حالی که چنین نیست. در صورتیکه نوع سربارگذاری شده‌ای برای اعداد صحیح وجود می‌داشت، تحمیل پیش نمی‌آمد و روال مناسب فراخوانی می‌شد.

چندشکلی بودن مؤثر

مانند ارکان دیگر، چندشکلی بودن صحیح هم تصادفی ایجاد نمی‌شود. لازم است چند مرحله طی شود تا از

مؤثر بودن چندشکلی بودن اطمینان حاصل شود.

قدم نخست حصول اطمینان از مؤثر بودن کپسوله‌سازی و وراثت است.

بدون کپسوله‌سازی، کد به سادگی به پیاده‌سازی کلاس وابسته می‌شود. اجازه ندهید کپسوله‌سازی نادیده گرفته شود. کپسوله‌سازی خوب نخستین گام به سوی چندشکلی بودن است.

نکته

لازم به تذکر است که مفهوم رابط (Interface) در این بحث اندکی با مفهوم آن در Java متفاوت است. در اینجا، رابط یعنی فهرست پیغام‌هایی که می‌توان به شیء ارسال کرد. این پیغامها رابط عمومی را شکل می‌دهند.

یک رابط Java، پیغامهایی که می‌توان به یک شیء Java ارسال کرد را لیست می‌کند. وقتی یک کلاس Java رابط را پیاده‌سازی می‌کند، تمام روالهای رابط جزیی از رابط عمومی کلی کلاس خواهند شد.

با این وجود، رابط Java تنها راه تعریف پیامهای قابل ارسال به شیء در Java نیست. در Java، هر روالی که در کلاس تعریف شود، جزیی از رابط عمومی خواهد شد. یعنی اگر کلاسی یک رابط را پیاده‌سازی کند و علاوه بر آن خود نیز چند روال اضافه کند، تمام این روالها در رابط عمومی آن ظاهر خواهند شد.

استفاده از رابط Java تمرین خوبی است چون تعریف رابط را از پیاده‌سازی آن رابط در کلاس جدا می‌کند. وقتی این دو را از هم جدا می‌کنید، کلاسهای بسیار دیگری هم می‌توانند از همان رابط استفاده کنند. مانند وراثت اشیایی که در یک رابط عمومی مشترکند، می‌توانند با هم روابط جانشین‌پذیری داشته باشند، بدون اینکه لازم باشد جزء یک زنجیرهٔ وراثتی باشند.

وراثت فاکتور مهمی در چندشکلی بودن درونی است. همواره سعی کنید روابط جانشین‌پذیر را بدون گسترش بیش از حد سلسله مراتب وراثت ایجاد کنید. با چنین کاری، به اشیاء بیشتری اجازه می‌دهید در برنامه شرکت کنند.

ایجاد سلسله مراتب وراثت بر اساس نقشهٔ دقیق راهی برای استفاده از جانشین‌پذیری است. اشتراکات را به کلاسهای مجرد منتقل کنید و اشیاء خود را برای استفاده از کلاسهای مجرد برنامه‌ریزی کنید. با این کار قادر خواهید بود از تمام اخلاف کلاس در برنامهٔ خود استفاده کنید.

نکته

برای حصول چندشکلی بودن مؤثر، نکات زیر را در نظر داشته باشید:

- به نکات کپسوله‌سازی و وراثت موثر توجه کنید.
- همواره برای رابط برنامه بنویسید، نه برای پیاده‌سازی. در این صورت می‌توانید تعریف کنید که چه انواعی می‌توانند در برنامه ظاهر شوند. ● بادیدی کل‌نگر بیاندیشید و برنامه بنویسید. دغدغهٔ انواع داده را به چندشکلی بودن واگذارید. در این صورت مجبور نیستید کد زیادی بنویسید.
- چندشکلی بودن را با برقراری و به کارگیری روابط جانشین‌پذیر پایه‌ریزی کنید. جانشین‌پذیری و چندشکلی بودن این اطمینان را ایجاد می‌کنند که کد مناسب برای انواع دادهٔ فرعی (sub – types) اجرا خواهد شد.
- اگر زبان مورد استفاده شما جداسازی رابط و پیاده‌سازی را پشتیبانی می‌کند، از آن در وراثت

استفاده کنید. زیرا انعطاف‌پذیری بیشتری در روابط جانشین‌پذیر ایجاد می‌کند و شانس استفاده از چندشکلی بودن را افزایش می‌دهد.

- برای جداکردن رابط و پیاده‌سازی، از کلاسهای مجرد استفاده کنید. بهتر است تمام کلاسهایی که برگ نیستند، مجرد باشند و فقط برای این کلاسهای مجرد برنامه بنویسید.

دامهای چندشکلی

وقتی از چندشکلی بودن استفاده می‌کنید، از سه تله باید احتراز کنید:

دام ۱: بالا رفتن رفتارها در سلسله مراتب

اغلب توسعه دهندگان برای افزایش چندشکلی بودن، محا تعریف رفتارها را سلسله مراتب به کلاسهای بالاتر منتقل می‌کنند. غافل از اینکه چنین کاری می‌تواند طراحی سلسله مراتب را تضعیف کند.

اگر رفتاری را بیش از حد در سلسله مراتب بالا ببرید، تمام اخلاف از آن بهره‌مند نخواهند شد. به یاد داشته باشید که اخلاف هرگز نباید کارایی ارث رسیده را حذف کنند. وراثت صحیح را به بهانهٔ افزایش چندشکلی بودن تخریب نکنید.

اگر متوجه شدید که قصد دارید رفتاری را در سلسله مراتب بالا ببرید تا چندشکلی بودن را بهبود بخشید، صبر کنید! خطر در کمین است.

اگر سلسله مراتب خود را محدود می‌بینید، آن را بازبینی کنید. عناصر مشترک را به کلاسهای مجرد منتقل کنید. اما فقط تا جایی که اولین بار به آنها نیاز دارید. همواره مطمئن شوید که برای بالاتر بردن رفتار، دلیل موجه دیگری هم دارید.

در زمان طراحی سلسله مراتب به نیازهای احتمالی آینده هم توجه داشته باشید. اما خود را فقط به آنهایی محدود کنید که احتمال می‌دهید پیش بیایند.

دام ۲: کاهش کارایی

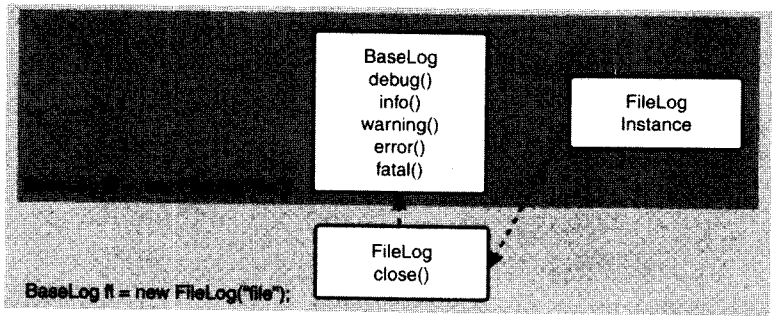
هر چیزی قیمتی دارد. در چندشکلی بودن واقعی، مشکل کاهش کارایی خود را نشان می‌دهد. روال چندشکلی نمی‌تواند از لحاظ کارایی با روالی که دقیقاً نوع آرگومان‌های خود را می‌شناسد رقابت کند. چون روال چندشکلی باید در زمان اجرا چندین بررسی انجام دهد تا روالی که باید اجرا شود را معین کند. تمام این بررسی‌ها زمان‌بر هستند و لذا روال چندشکلی کندتر انجام می‌شود.

البته مزایای چندشکلی بودن از جمله سهولت مدیریت و انعطاف‌پذیری برنامه در بسیاری از موارد این نقص را جبران می‌کنند. با این حال در صورتی که زمان پاسخدهی در برنامه اهمیت ویژه‌ای دارد، باید در مورد استفاده از چندشکلی بودن احتیاط لازمه را به خرج داد. پس، کارایی بهینه را مدنظر قرار دهید. در پیاده‌سازی دقت کنید و در موارد لزوم کارایی را بهبود بدهید.

دام ۳: محدودکنندگی ـ چشم‌بندها

چندشکلی بودن درونی یک ضعف دارد. این درست است که می‌توان یک زیرکلاس را به روالی که برای والد طراحی شده پاس کرد، با این حال روال نمی‌تواند از هیچ‌یک از قابلیتهای جدیدی که زیرکلاس به خود

شکل ۶ ـ ۲
نماهای مختلف یک شی



افزوده استفاده کند. چون نمی‌تواند آنها را ببیند. مثلاً FileLog، روال ()close را به رابط می‌افزاید. اما روال ()retrieveEmployee از EmployeeDatabaseAccessor نمی‌تواند از آن استفاده کند.

شکل ۶ ـ ۲ نشان می‌دهد، آنچه از نمونه دیده می‌شود با والد مرتبط است. استفاده از FileLog به جای BaseLog مانند بستن یک چشم‌بند است. زیرا تنها می‌توان به روالهای خود Baselog دسترسی داشت. البته اگر FileLog در جای خود استفاده شود تمام روالهای آن قابل استفاده خواهد بود.

بنابراین هرگاه انواع چندشکلی جدیدی ایجاد می‌کنید، کد قبلی قادر نخواهد بود که به هیچ روال جدیدی دسترسی داشته باشد.

بار دیگر می‌توان نتیجه گرفت که اخلاف نباید رفتارهای موجود والد را حذف کنند. روالی که بر چند شکلی بودن درونی مبتنی باشد تنها می‌تواند با روالهای تعریف شده در نوعی که برای آن نوشته شده کار کند. اگر رفتاری حذف شده باشد، کد غیرقابل استفاده خواهد بود.

علاوه بر این می‌توان فهمید که تعویض نوع قبلی مورد استفاده، با نوع جدید چندان که می‌نماید آسان نخواهد بود. برای نمونه در مثال FileLog باید راهی برای فراخوانی ()close پیدا کنید.

هشدار

در مبحث استفاده از چندشکلی بودن تنها یک محدودیت عمده وجود دارد. هر زبانی چندشکلی بودن را به صورتی متفاوت پیاده‌سازی می‌کند. این بحث تعاریف تئوری ورای چندشکلی بودن را شکل داده است.

بیشتر زبان‌ها چندشکلی بودن درونی را تا حدی پشتیبانی می‌کنند. از سوی دیگر، تعدادی کمی از آنها چندشکلی بودن پارامتری را پشتیبانی می‌کنند. مثلاً Java چندشکلی بودن پارامتری را پشتیبانی نمی‌کند، اما C++ وانمود می‌کند که آن را پیاده‌سازی می‌کند.

بیشتر زبان‌ها دارای برخی انواع سربارگذاری و تحمیل هستند. با این حال پیاده‌سازی در زبان‌های مختلف، متفاوت است. بنابراین وقتی برنامه‌نویسی چندشکلی انجام می‌دهید، تئوری را به یاد داشته باشید، اما هرگز محدودیتهای زبان را از جلوی چشم دور نکنید.

چگونه چندشکلی بودن اهداف برنامه‌نویسی شیءگرا را تأمین می‌کند

چندشکلی بودن، نرم‌افزاری ایجاد می‌کند که دارای خواص زیر باشد:

۱. طبیعی بودن
۲. قابلیت استفاده مجدد
۳. توسعه‌پذیری
٤. قابلیت اعتماد
۵. مدیریت‌پذیر
۶. صرفه زمانی

و هر هدف را به صورتی خاص تأمین می‌کند.

- طبیعی بودن: به جای برنامه‌نویسی برای شرایط خاص، با چندشکلی بودن در حدی عام‌تر و مفهومی‌تر می‌توان کار کرد. در این راه سربارگذاری و چندشکلی بودن پارامتری به مدلسازی اشیاء یا روالها در سطح مفهومی کمک شایانی می‌کنند. چندشکلی بودن درونی هم این امکان را فراهم می‌کند که با انواع اشیاء کارکرد و نه با پیاده‌سازی‌های خاص.

چنین برنامه‌نویسی ژنریکی طبیعی‌تر است زیرا برنامه‌نویسی را برای کار در سطح مفهومی مسأله آزاد می‌گذارد.

- قابلیت اعتماد: نخست آنکه چندشکلی بودن کد را ساده‌تر می‌کند. به جای نوشتن کد خاص برای هر مورد خاص، تنها برای یک حالت کد می‌نویسیم. اگر چنین کاری نکنیم مجبوریم برای افزودن هر زیرکلاس کد را تغییر دهیم. این کار برنامه را مستعد ایجاد خطا می‌کند.

دوم اینکه، با چندشکلی بودن می‌توانیم حجم کد کمتری داشته باشیم و لذا در صورت ایجاد خطا تغییرات کمتری هم خواهیم داشت.

علاوه بر اینها چندشکلی بودن اجازه می‌دهد بخش‌هایی از کد را از تغییرات زیرکلاسها ایزوله کنیم. با این کار می‌توان مطمئن بود که کد تنها با سطوحی از سلسله مراتب وراثت تماس دارد، که لازم است.

- استفادهٔ مجدد: برای اینکه یک شیء بتواند از دیگری استفاده کند فقط لازم است رابط عمومی آن را بشناسد و نه روش پیاده‌سازی را. در نتیجه استفاده مجدد به سهولت رخ می‌دهد.
- مدیریت‌پذیر بودن: همچنانکه تا به حال دیده‌اید، چندشکلی بودن کد پیوسته‌تری تولید می‌کند. لذا در صورتی که بخواهید از کد رفع اشکال کنید، کد کمتری را باید بررسی کنید.
- توسعه‌پذیری: کد چندشکلی بسط‌پذیرتر است. چندشکلی بودن درونی، افزودن زیرکلاسهای جدید به برنامه را بدون نیاز به تغییر کد قبلی، ممکن می‌کند. سربارگذاری هم امکان افزودن متدهای جدید بدون ایجاد نگرانی در مورد اسامی روال‌ها را فراهم می‌کند. نهایتاً، با چندشکلی بودن پارامتری می‌توان به طور خودکار کلاسها را برای استفاده از انواع جدید بسط داد.
- صرفهٔ زمانی: اگر بتوانید کدنویسی کمتری داشته باشید، کار کدنویسی زودتر تمام می‌شود. از آنجایی که با چندشکلی بودن می‌توان کد ژنریک نوشت، افزودن بلادرنگ انواع جدید به برنامه ممکن است. در نتیجه تغییر و توسعه برنامه در زمان کمتری ممکن خواهد شد.

خلاصه

چندشکلی بودن یعنی اینکه چیزی چندین شکل مختلف داشته باشد. چندشکلی بودن فرایندی است که

اجازه می‌دهد یک نام واحد معرف کدهای مختلفی باشد. لذا یک نام می‌تواند رفتارهای مختلفی را بیان کند.

در این بحث به چهار نوع مختلف چندشکلی بودن پرداخته شد.

- چندشکلی بودن درونی
- چند شکلی بودن پارامتری
- جایگزینی
- سربارگذاری

انواع فوق‌الذکر، اشکال رایجتر چندشکلی بودن هستند. درک مفهوم این اشکال مختلف زیربنای خوبی در نظریه چندشکلی بودن به برنامه‌نویس می‌دهد.

چندشکلی بودن درونی ممکن می‌دارد که یک شیء رفتارهای متفاوتی را در زمان اجرا از خود نشان دهد. به همان صورت چندشکلی بودن پارامتری به شیء امکان می‌دهد با انواع پارامترهای مختلفی کار کند.

با جایگزینی می‌توان روالی را در فرزند تغییر داد و به چندشکلی بودن در مورد اجرای روال مناسب اعتماد کرد.

در پایان، سربارگذاری برنامه‌نویس را قادر می‌سازد که یک روال را چندین بار تعریف کند. هر تعریف فقط در تعداد یا نوع پارامترها متفاوت است. تحمیل از طریق تغییر نوع آرگومانها به نوع مناسب برای روال، آن روال را چندشکلی نشان می‌دهد.

با چندشکلی بودن می‌توان کدی کوتاهتر و قابل فهم‌تر نوشت که در مقابل نیازهای آینده انعطاف‌پذیرتر باشد.

پرسش‌ها و پاسخ‌ها

اگر تمام ارکان برنامه‌نویسی شیءگرا با هم در برنامه‌ای رعایت نشوند، آیا آن نرم‌افزار غیرشیءگرا است؟

دست پایین، همواره کپسوله‌سازی باید استفاده شود. بدون کپسوله‌سازی نمی‌توان وراثت مؤثر داشت. یا از چندشکلی بودن استفاده کرد. دو رکن دیگر را فقط در موقع لزوم باید استفاده کنید. آنها را فقط برای اینکه استفاده کرده باشید، مورد استفاده قرار ندهید.

غیبت وراثت یا چندشکلی بودن خللی در شیءگرا بودن برنامه ایجاد نمی‌کند.

چرا در جوامع برنامه‌نویسی شیءگرا، این قدر در مورد چندشکلی بودن اختلاف نظر وجود دارد؟

به نظر می‌رسد هر نویسنده فرهنگ واژگان خاص خود را مورد استفاده قرار می‌دهد. بیشتر این اختلاف به دلیل تفاوت در طرز پیاده‌سازی چندشکلی بودن در زبان‌های مختلف است.

آنچه اهمیت دارد، درک این چهار نوعی است که امروز مورد بررسی قرار گرفتند. این چهار نوع هرچند ممکن است به صورتهای متفاوتی نامگذاری شده باشند، اما به طور کلی مورد توافقند.

کارگاه

پرسشها

۱. **چهار شکل مختلف چندشکلی بودن کدامند؟**

۲. **چندشکلی بودن درونی چه امکانی را فراهم می‌کند؟**

۳. چرا سربارگذاری و چندشکلی بودن پارامتری دنیای واقعی را طبیعی‌تر مدل می‌کنند؟
٤. وقتی برنامه می‌نویسید، چرا بهتر است رابط را در نظر بگیرید و نه پیاده‌سازی را؟
۵. چندشکلی بودن و جایگزینی چگونه با هم عمل می‌کنند؟
۶. نام دیگر سربارگذاری چیست؟
۷. سربارگذاری را تعریف کنید.
۸. چندشکلی بودن پارامتری را تعریف کنید.
۹. سه دامی که در راه اجرای چندشکلی بودن پهن شده‌اند کدامند؟
۱۰. چگونه کپسوله‌سازی و وراثت در چندشکلی بودن تضامنی لحاظ می‌شوند؟

تمرین‌ها

۱. یک مثال از برنامه‌نویسی عملی بزنید که فکر می‌کنید می‌توان در آن از چندشکلی بودن درونی استفاده کرد.

۲. یک مثال در مورد تحمیل پیدا کنید. چرایی آن را شرح دهید.

۳. در میان APIهای Java مثالی از سربارگذاری پیدا کنید و آن را شرح دهید. سپس سلسله مراتب یک کلاس را بیابید که می‌توان آن را برای چندشکلی بودن درونی به کار برد. سلسله مراتب را شرح دهید و توضیح دهید که چگونه می‌توان چندشکلی بودن درونی را بر آن اعمال کرد.

روز ۷

Object Oriented Programming

چند شکلی بودن زمان نوشتن کد

دیروز مطالبی را در مورد چندشکلی (Polymorphism) یاد گرفتید. باید فهم خوبی از چهار نوع مختلف از چند شکل داشته باشید. امروز با حل چند تمرین و انجام چند کارگاه به صورت عملی با چندشکلی آشنا خواهید شد. در انتهای درس امروز، باید مطالب تئوری ارایه شده در درس روز ششم را به خوبی پیاده نمایید.

بپردازیم به آنچه امروز خواهید آموخت

- چگونه اشکال مختلف چندشکلی را اعمال کنید.
- چگونه نرم‌افزارهای آینده‌نگر بنویسید.
- چگونه چندشکلی منطق switch را بلا استفاده می‌کند.

کارگاه ۱: اعمال کردن چند شکلی

روز پنجم، کارگاه ۲ سلسله مراتب کلاسهای یک کارمند/کارگر را به شما نشان داد. لیست ۷ ـ ۱ کلاس پایه Employee متفاوتی را نشان می‌دهد.

لیست ۷-۱ Employee.java

```
public abstract class Employee {
    private String first_name;
    private String last_name;
```

لیست ۷-۱ (ادامه)

```
    private double wage;

    public Employee( String first_name, String last_name, double wage ) {
        this.first_name = first_name;
        this.last_name  = last_name;
        this.wage = wage;
    }

    public double getWage() {
         return wage;
    }

    public String getFirstName() {
        return first_name;
    }

    public String getLastName() {
        return last_name;
    }

    public abstract double calculatePay();

    public void printPaycheck() {
        String full_name = last_name + ", " + first_name;
        System.out.println( "Pay: " + full_name + " $" + calculatePay() );
    }
}
```

کلاس جدید Employee متد مجردی به نام colculatePay() دارد. هر زیرکلاس باید این متد را پیاده‌سازی کند. لیستهای ۷ـ۲ و ۷ـ۳ دو نمونه زیر کلاس را نشان می‌دهند.

لیست ۷-۲ ComissionedEmployee.java

```
public class CommissionedEmployee extends Employee {

    private double commission; // the $ per unit
    private int    units;      // keep track of the # of units sold

public CommissionedEmployee( String first_name, String last_name, double wage, double
commission ) {
super( first_name, last_name, wage ); // call the original constructor in order to properly
initialize
        this.commission  = commission;
```

لیست ۷-۲ (ادامه)

```
    }
    public double calculatePay() {
        return getWage() + ( commission * units );
    }

    public void addSales( int units ) {
        this.units = this.units + units;
    }

    public int getSales() {
      return units;
    }

    public void resetSales() {
        units = 0;
    }

}
```

لیست ۷-۳ HourlyEmployee.java

```
public class HourlyEmployee extends Employee {

    private int hours; // keep track of the # of hours worked

    public HourlyEmployee( String first_name, String last_name, double wage ) {
super( first_name, last_name, wage ); // call the original constructor in order to properly
initialize
    }

    public double calculatePay() {
        return getWage() * hours;
    }

    public void addHours( int hours ) {
        this.hours = this.hours + hours;
    }

    public int getHours() {
       return hours;
    }
```

لیست ۷-۳ (ادامه)

```
    public void resetHours() {
        hours = 0;
    }

}
```

هریک از زیرکلاسها به نحو خاصی متد calculatePay() را پیاده‌سازی کرده‌اند. HourlyEmployee به سادگی با ضرب تعداد ساعتها در نرخ ساعتی میزان حقوق را محاسبه کرده است. در حالی که CommissionedEmployee حقوق را بر اساس مجموع حقوق پایه و کمیسیون (تعداد محصولات فروخته شده) محاسبه کرده است. در ضمن هریک از زیرکلاسها شامل تعدادی متد دیگر هستند. برای مثال در کلاس HourlyEmployee متدی برای صفر کردن میزان ساعت (reset) در نظر گرفته شده است. یا در کلاس CommissionedEmployee متدی برای اضافه کردن میزان فروش در نظر گرفته شده است.

همچنانکه در روز چهارم فراگرفتید، هر دو کلاس CommissionedEmployee و HourlyEmployee اجازه می‌دهند نمونه‌های هر دو کلاس مواردی را به اشتراک بگذارند. می‌توان هریک از نمونه‌های کلاسهای CommissionedEmployee و HourlyEmployee را به جای کلاس Employee به کار گرفت. با این توصیفات خاصیت چندشکلی به شما اجازه چه کاری می‌دهد؟

کلاس Payroll در لیست ۷ـ۴ را در نظر بگیرید.

لیست ۷-٤ Payroll.java

```
public class Payroll {

    private int    total_hours;
    private int    total_sales;
    private double total_pay;

    public void payEmployees( Employee [] emps ) {
        for( int i = 0; i < emps.length; i ++ ) {
            Employee emp = emps[i];
            total_pay += emp.calculatePay();
            emp.printPaycheck();
        }
    }

    public void recordEmployeeInfo( CommissionedEmployee emp ) {
        total_sales += emp.getSales();
    }

    public void recordEmployeeInfo( HourlyEmployee emp ) {
        total_hours += emp.getHours();
    }
```

لیست ۷-٤ (ادامه)

```
    public void printReport() {
        System.out.println( "Payroll Report:" );
        System.out.println( "Total Hours: " + total_hours );
        System.out.println( "Total Sales: " + total_sales );
        System.out.println( "Total Paid: $" + total_pay );
    }

}
```

توجه

متدهای get و set زیاد، نشان دهنده طراحی OO بدی است. به ندرت پیش می‌آید که از یک شیء درخواست داده‌های درونی آن کنید. در عوض از یک شیء می‌خواهید که کار خاصی را انجام دهد.

در مثال کارمند (Employee) بهتر است که به شیء Employee در جاهایی که نیاز به ذخیره ساعتها، حقوق و... است از شیء Report استفاده کرد.

البته OO خوب امری نسبی است. درواقع اگر اشیاء کلی می‌نویسید و می‌خواهید از آن در حالات و موقعیتهای مختلف استفاده کنید برای مدیریت رابط کلاس بهتر است از تعدادی متد get و set نیز استفاده کنید.

متد payEmployee(Employee[] emps) را در نظر بگیرید. رابطهای زیرکلاسها اجازه می‌دهند که هر زیرکلاس از Employee را به متد ارسال کرد. درواقع این متد با کلاسهای HourlyEmployees و CommissionedEmployee همچون نمونه‌های ساده‌ای از کلاس Employee رفتار می‌کند.

خاصیت چندشکلی باعث جذابیت این مثال می‌شود. وقتی متد payEmployees() رابطه زیر را بیان می‌کند:

```
total_pay + = emp.calculatePay();
```

چندشکلی باعث می‌شود که Employee رفتارهای بسیار متفاوتی داشته باشد. زمانی که emp.calculatePay() بر روی شیء نظیر HourlyEmployee واقعی فراخوانی می‌شود، calculatePay() حقوق را بر اساس ضرب ساعتها در نرخ ساعتی محاسبه می‌کند و اگر بر روی شیء از نوع CommissionedEmployee فراخوانی گردد حقوق بر اساس پایه حقوق و جمع آن با کمیسیون حاصل از فروش محاسبه می‌گردد.

payEmployees() مثالی از چندشکلی درونی است. چراکه این متد برای هر نوع کارمندی کار می‌کند. به عبارت دیگر این متد نیازی به کد ویژه‌ای ندارد تا به ازای هر کلاس جدید که به سیستم اضافه می‌گردد، به کار روز گردد. به عبارت دیگر به سادگی برای تمام زیر کلاسهای کلاس Employee می‌کند.

متدهایی نظیر نحوه سربارگذاری را نمایش recordEmployeeInfo (commissionedEmployee emp) و می‌دهد که متدی به چند شکل recordEmployeeInfo(HourlyEmployee emp) می‌دهند. سربارگذاری اجازه ظاهر گردد. برای مثال، نوشتن کدهای زیر را ممکن می‌سازد:

```
Payroll payroll= new Payroll();
CommissionedEmployee emp1=new CommissionedEmployee("Mr.","Sales",25000.00,1000.00);

HourlyEmployee emp2=new HourlyEmployee("Mr.","Minimum wage",6.50);
payroll.recordEmployeeInfo(emp2);
payroll.recordEmployeeInfo(emp1);
```

متد recordEmployeeInfo() به چند شکل ظاهر گشته است چراکه هر دو نوع کارمند را پشتیبانی می‌کند.

البته سربارگذاری محدودتر از قابلیت چندشکلی درونی است. همچنانکه دیدید با استفاده از چندشکلی درونی تنها نیاز به یک متد است. درواقع بدون آنکه نگران زیرکلاسهای کلاس Employee باشید، برای همه آنها کار می‌کند و این قدرت چندشکلی درونی را نشان می‌دهد.

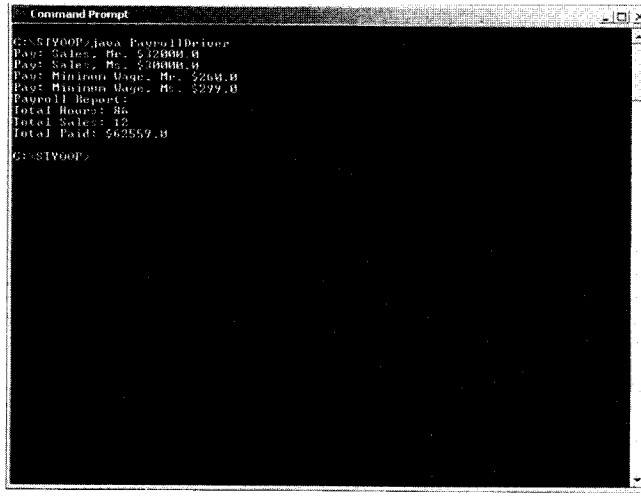
همچنین متدهایی که از روش سربارگذاری استفاده می‌کنند، از پایداری خوبی برخوردار نیستند زیرا هر بار که زیرکلاسی به کلاس پایه اضافه می‌گردد باید متدی را برای پشتیبانی از آن زیرکلاس اضافه کرد. متد recordEmployeeInfo() مبین این امر است.

لیست ۷ـ۵ مثال کوچکی از Payroll نشان می‌دهد.

لیست ۷-۵ PayrollDriver.java

```
public class PayrollDriver {
   public static void main( String [] args ) {

      // create the payroll system
      Payroll payroll = new Payroll();

      // create and update some employees
CommissionedEmployee emp1 = new CommissionedEmployee( "Mr.", "Sales", 25000.00,
1000.00);
CommissionedEmployee emp2 = new CommissionedEmployee( "Ms.", "Sales", 25000.00,
1000.00);
      emp1.addSales( 7 );
      emp2.addSales( 5 );

      HourlyEmployee emp3 = new HourlyEmployee( "Mr.", "Minimum Wage", 6.50 );
      HourlyEmployee emp4 = new HourlyEmployee( "Ms.", "Minimum Wage", 6.50 );
      emp3.addHours( 40 );
      emp4.addHours( 46 );

      // use the overloaded methods
      payroll.recordEmployeeInfo( emp2 );
      payroll.recordEmployeeInfo( emp1 );
      payroll.recordEmployeeInfo( emp3 );
      payroll.recordEmployeeInfo( emp4 );

      // stick the employees in an array
      Employee [] emps = new Employee[4];
      emps[0] = emp1; emps[1] = emp2; emps[2] = emp3; emps[3] = emp4;

      payroll.payEmployees( emps );
      payroll.printReport();
   }
}
```

شکل ۷ـ۱ خروجی برنامه فوق را نشان می‌دهد.



شکل ۷ـ۱
خروجی PayrollDriver

اگر حقوق هریک از کارمندان را به صورت دستی محاسبه کنید، خواهید دید که حقوق هریک از آنها به درستی محاسبه شده است. همچنین خواهید دید که همه اطلاعات مربوط به کارمندان به درستی ضبط گردیده است.

تعریف مسأله

در روز پنجم با MoodyObject کار کردید. لیست ۷ـ۶ نسخه تغییر یافته MoodyObject را نشان می‌دهد.

لیست ۷ـ۶ MoodyObject.java

```
public abstract class MoodyObject {

    // return the mood
    protected abstract String getMood();

    // ask the object how it feels
    public void queryMood() {
    System.out.println("I feel " + getMood() + " today!");
    }

}
```

لیست ۷ـ۷ و ۷ـ۸ دو زیرکلاس HappyObject و SadObject را نشان می‌دهد.

لیست ۷ـ۷ HappyObject.java

```
public class HappyObject extends MoodyObject {

    // redefine class's mood
    protected String getMood() {
```

لیست ۷ـ۷ (ادامه)

```
    return "happy";
    }

    // specialization
    public void laugh() {
    System.out.println("hehehe... hahaha... HAHAHAHAHAHA!!!!!");
    }
}
```

لیست ۷ـ۸ SadObject.java

```
public class SadObject extends MoodyObject {

    // redefine class's mood
    protected String getMood() {
    return "sad";
    }

    // specialization
    public void cry() {
    System.out.println("'wah' 'boo hoo' 'weep' 'sob' 'weep'");
    }
}
```

کار شما هم اکنون تمرین قابلیت چندشکلی است. برای همین منظور کلاسی به نام PsychiatristObject بنویسید. کلاس PsychiatristObject باید سه متد داشته باشد. متد examine() باید نمونه‌هایی از کلاسهای MoodyObject را گرفته و در مورد احساسشان از آنها سؤال کند. این کلاس همچنین باید متدی به نام observe() را سربارگذاری کند. متد observe() باید متدهای cry() و یا laugh() از آن اشیاء را فراخوانی کند. در آخرین کلاس باید برای هریک از رفتارها، یک توضیح پزشکی ارایه کند.

راه حل خود را با آنچه که بعداً می‌آید مقایسه کنید تا نسبت به جواب خود مطمئن شوید.

توجه بخش بعدی راه حل کارگاه ۱ را ارایه می‌کند. تا قبل از تکمیل کارگاه ۱ به این بخش مراجعه نکنید.

حل و بحث

لیست ۷ـ۹ یک راه حل ممکن برای کلاس PsychiatristObject ارایه می‌کند.

لیست ۷ـ۹ PsychiatristObject.java

```
public class PsychiatristDriver {
```

لیست ۷ـ۹ (ادامه)

```
    public static void main( String [] args ) {
        HappyObject happy = new HappyObject();
        SadObject sad = new SadObject();
        PsychiatristObject psychiatrist = new PsychiatristObject();

        // use inclusion polymorphism
        psychiatrist.examine( happy );
        psychiatrist.examine( sad );

        // use overloading so that we can observe the objects
        psychiatrist.observe( happy );
        psychiatrist.observe( sad );
    }

}
```

متد examine(MoodyObject obj) با همه انواع MoodyObject به طور کلی می‌تواند کار کند. درواقع کلاس PsychiatristObject از کلاس MoodyObject در مورد احساسش سؤال می‌کند و این کار را از طریق متد queryMood() آن کلاس انجام می‌دهد. برای کلاس‌های زیر مجموعه MoodyObject باید برای هریک، متد observe() را پیاده‌سازی کرد.

با اتمام این کارگاه باید با مبانی چندشکلی (پلی مورفیسم) به خوبی آشنا شده باشید.

کارگاه ۲: حساب بانکی ـ اعمال چندشکلی بر روی مثالی آشنا

در کارگاه ۲ آنچه را که در کارگاه قبل آموخته‌اید، بر روی مسأله‌ای واقعی‌تر پیاده خواهید کرد. این کارگاه بر روی سلسله مراتب کلاسهای BankAccount که در روز پنجم نشان داده شده است، متمرکز می‌شود. سلسله مراتبی که در اینجا به نمایش گذاشته می‌شود تقریباً همانی است که در روز پنجم مشاهده نمودید. تنها تفاوت در این است که کلاس BankAccount در این کارگاه به صورت مجرد تعریف شده است. دیگر هیچ نیازی به استفاده مستقیم از کلاس BankAccount نیست. با ایجاد کلاس BankAccount و تعریف آن به صورت مجرد، حسابهای بانکی به صورت واقعی‌تر مدل می‌شوند. با باز کردن یک حساب بانکی، شما درواقع یا یک حساب بانکی پس‌انداز باز کرده‌اید یا حساب جاری. دیگر نیازی به باز کردن یک حساب کلی نیست. لیست ۷ ـ ۱۰ تنها تفاوتهای ایجاد شده در سلسله مراتب کلاسهای قبلی را نشان می‌دهد.

لیست ۷ـ۱۰ BankAccount.java

```
public abstract class BankAccount {
    // the rest is the same
}
```

تعریف مسأله

در این کارگاه باید یک کلاس Bank نوشت. این کلاس شامل تعدادی متد می‌شود. نمونه‌های کلاس Bank

باید انواع حسابها را در خود نگهدارند. برای این منظور راهی را باید مدنظر قرار دهیم تا بتوان به راحتی حسابها را مدیریت کرد. متد addAccount اجازه می‌دهد هر زمان که حسابی را اضافه می‌کنیم، صاحب حساب (Owner) آن را نیز مشخص نماییم:

```
Public void addAbount (String name, BankAccount account);
```

می‌توان با استفاده از نام صاحب حساب، به حساب دسترسی پیدا کرد.

متد totalHoldings() گزارشی کامل از میزان پولی که در همه حسابهای بانکی قرار داده ارایه می‌دهد:

```
public double totalHodlings()
```

totalHoldings() باید در تمام حسابها گشته و میزان پول موجود در همه آنها را جمع کند.

از طریق totalAccounts() می‌توان تعدادحسابهای بانکی در یک نمونه از کلاس Bank را به دست آورد:

```
public int totalAccounts();
```

از طریق متد deposit() می‌توان پولی را به حساب واریز کرد:

```
public void deposit (String name, double amount);
```

در این حالت دیگر نیازی نیست که ابتدا حساب خاصی را پیدا کرده و سپس پول را واریز کنید. در عوض deposit() اجازه می‌دهد که مستقیماً پول را به بانک انتقال دهید.

متد balance() میزان موجودی یک حساب خاص را گزارش می‌کند.

```
public double balance (String name);
```

متد addAcount() حساب را تحت نام داده شده، ذخیره می‌کند. برای پیاده‌سازی این امر روشهای مختلف وجود دارد. با این حال برخی از روشها ساده‌تر از بقیه هستند.

برای این کارگاه می‌توانید از java.util.Hashtable استفاده کنید. Hashtable اجازه می‌دهد که کلیدواژه/مقدار را ذخیره کرده و یا دریافت کنید.

توابع زیر را در نظر بگیرید:

```
public Object get(Object key);
public Object put(Object key, object value);
public int size();
public java.util.Enumeration elements();
```

مثالی از Hashtable در زیر آمده است:

```
java. util.Hashtable table = new java.util.Hashtable();
table.put("LANGUAGE", "JAVA");
String name = table.get("LANGUAGE");
```

در این مثال ابتدا مقدار JAVA تحت کلیدواژه LANGUAGE ذخیره شده است. برای دریافت مقدار کافی است به سادگی متد get() را به همراه کلیدواژه مورد نظر فراخوانی کنیم.

با نگاهی دقیق به توابع گفته شده، خواهید دید که هر دو متد get و put، Object را به عنوان خروجی برمی‌گردانند. بنابراین اگر رشته‌ای را ذخیره کنید، مقدار بازگشتی توسط get، شی از نوع object خواهد بود.

در Java، همه اشیاء از object مشتق می‌شوند. کلاس Hashtable از این رو همراه با object نوشته شده است که بتواند با همه انواع اشیاء کار کند. با این تفصیلات اگر مبادرت به ذخیره CheckingAccount در Hashtable نمایید، چگونه می‌توانید بعداً آن را از Hashtable دریافت نمایید؟ این کار را در Java چگونه انجام می‌دهید؟

Java برای این امر راهی را در نظر گرفته است. این مکانیزم تبدیل نوع (casting) نامیده می‌شود. برای مثال، دستورات زیر در Java معتبر نیستند:

```
CheckingAccount account = table.get("CECKING_ACCOUNT");
```

به هنگام تبدیل نوع، دقت به خرج دهید. چرا که می‌تواند گاهی اوقات خطرناک باشد! برای مثال، دستورات زیر معتبر نیستند:

```
HappyObject o=new HappyObject();
table.put("HAPPY", o);
(CheckingAccount) table.get("HAPPY");
```

زمانی که مبادرت به تبدیل نوع می‌کنید باید مطمئن باشید که شیء را که می‌خواهید نوع آن را تغییر دهید، حتماً از نوع (CAST-TYPE) باشد. برای این کارگاه، زمانی که یک کلاس BankAccount را از Hashtable فراخوانی می‌کنید نوع آن را تبدیل به BankAccount می‌کنید. به عنوان یک مثال، تبدیل زیر را در نظر بگیرید:

```
BankAccount b = (BankAccount) table.get("ACCOUNT1");
```

اگر بخواهید تبدیل نا معتبری انجام دهید، Java پیغام خطایی صادر می‌کند که از طریق کلاس ClassCastException قابل ردگیری است.

همچون کارگاه ۱، کارگاه ۲ روشی را برای آزمایش راه حل ارایه شده نشان می‌دهد. حتماً کلاس BankAccount خود را مورد آزمایش قرار دهید.

توجه بخش بعدی راه حل کارگاه ۲ را ارایه می‌کند. تا زمانی که کارگاه ۲ را به اتمام نرسانده‌اید به این بخش مراجعه نکنید.

حل و بحث

لیست ۷ـ ۱۱ یک پیاده‌سازی ممکن را نشان می‌دهد:

لیست ۷ـ۱۱ Bank.java

```
public class Bank {
```

لیست ۷-۱۱ (ادامه)

```
    private java.util.Hashtable accounts = new java.util.Hashtable();

    public void addAccount( String name, BankAccount account ) {
        accounts.put( name, account );
    }

    public double totalHoldings() {
        double total = 0.0;

        java.util.Enumeration enum = accounts.elements();
        while( enum.hasMoreElements() ) {
            BankAccount account = (BankAccount) enum.nextElement();
            total += account.getBalance();
        }
        return total;
    }

    public int totalAccounts() {
        return accounts.size();
    }

    public void deposit( String name, double amount ) {
        BankAccount account = retrieveAccount( name );
        if( account != null ) {
            account.depositFunds( amount );
        }
    }

    public double balance( String name ) {
        BankAccount account = retrieveAccount( name );
        if( account != null ) {
            return account.getBalance();
        }
        return 0.0;
    }

    private BankAccount retrieveAccount( String name ) {
        return (BankAccount) accounts.get( name );
    }
}
```

درون راه حل ارایه شده، از java.util.Hashtable استفاده شده است تا از آن طریق بتوان همه حسابهای BankAccount را ذخیره کرد. به جای آنکه بخواهیم راه حل خودمان را جهت ذخیره و دستیابی بنویسیم، این راه حل نشان می‌دهد چگونه می‌توان از کلاسهای ارایه شده توسط Java استفاده مجدد کرد.

متدهای balance()، deposit()، addAccount() و totalHoldings() همگی نشان دهنده چندشکلی درونی هستند. این متدها برای همه انواع زیرکلاسهای کلاس BankAccount کار خواهند کرد.

پس از اتمام این کارگاه با جنبه‌های بسیار مهم چندشکلی آشنا شده‌اید. در روز پنجم، کلاس

BankAccount نحوه کارکرد وراثت را به شما نشان داد. وراثت اجازه می‌دهد به سادگی بتوان زیرکلاسهایی را ایجاد کرد که تنها در پاره‌ای از جزییات با یکدیگر تفاوت دارند. چندشکلی از طریق ارایه دادن مکانیزمی برای نوشتن برنامه‌های کلی، کدهای شما را ساده می‌نماید.

کارگاه ۳: حساب بانکی ـ استفاده از قابلیت چندشکلی برای نوشتن کدهای آینده‌نگر

در خلال بحث چندشکلی با اصطلاح نرم‌افزارهای آینده‌نگر آشنا شدید. به طور دقیق، نرم‌افزارهای آینده‌نگر چه چیزی هستند؟ به زبان ساده، نرم‌افزارهایی هستند که خود را با تغییر نیازمندیها مطابقت می‌دهند.

نیازمندیها همیشه در طول زمان تغییر پیدا می‌کنند. زمانی که اقدام به نوشتن برنامه‌ای می‌کنید، در خلال حل مسأله‌ای که با آن روبرو هستید نیازمندیهای آن به تدریج تغییر می‌کنند. زمانی هم که برنامه کاملاً نوشته شد و به کاربران ارایه شد، پس از مدتی کاربران درخواست ارایه ویژگیها و قابلیتهای جدید می‌کنند. در این صورت باید نرم‌افزارتان را مطابق نیازهای روز تغییر دهید. اگر برنامه‌تان را متناسب با آینده بنویسید نیازی نیست که تمام کدهای برنامه را از ابتدا تا انتها بازنویسی کنید.

اجازه دهید مثالی از تغییر نیازها را با هم بررسی کنیم. لیست ۷ ـ ۱۲ نمونه جدید MoodyObject، CareFreeObject را نشان می‌دهد.

لیست ۷-۱۲ CarefreeObject.java

```
public class CarefreeObject extends MoodyObject {

    // redefine class's mood
    protected String getMood() {
        return "carefree";
    }

    // specialization
    public void whistle() {
    System.out.println("whistle, whistle, whistle...");
    }
}
```

لیست ۷ ـ ۱۳ PsychiatristDriver به روز شده را نشان می‌دهد.

لیست ۷-۱۳ PsychiatristDriver

```
public class PsychiatristDriver {
    public static void main( String [] args ) {
        HappyObject happy = new HappyObject();
        SadObject sad = new SadObject();
        CarefreeObject carefree = new CarefreeObject();
```

لیست ۷-۱۳ (ادامه)

```
        PsychiatristObject psychiatrist = new PsychiatristObject();

        // use inclusion polymorphism
        psychiatrist.examine( happy );
        psychiatrist.examine( sad );
        psychiatrist.examine( carefree );

        // use overloading so that we can observe the objects
        psychiatrist.observe( happy );
        psychiatrist.observe( sad );
    }
}
```

شکل ۷ ـ ۲ خروجی برنامه را به هنگام اجرای PsychiatristDriver را نشان می‌دهد:
همانگونه که ملاحظه کردید PsychiatristObject متناسب با آینده ایجاد شده است. می‌توانید اشیاء جدید MoodyObject را همراه با خواص و رفتارهای جدید در هر زمان که خواستید اضافه کنید.

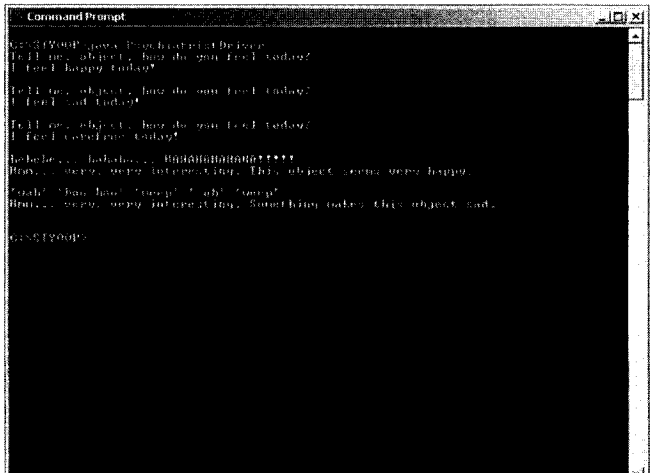
توجه

همانگونه که ممکن است متوجه شده باشید مثال قبل بر روی متد examine() متمرکز شده است. چندشکلی درونی اجازه ایجاد نرم‌افزارهای متناسب با آینده واقعی را می‌دهد. با این حال اگر PsychiatristObject بخواهد از طریق متد نوع جدیدی را دریافت کند باید کدهای آن را تغییر داد.

MoodyObject یک مثال ساده است. با این حال تصور کنید که چگونه می‌توانید از این قابلیت برای توسعه برنامه‌های پیچیده استفاده نمایید!

تعریف مسأله

وظیفه شما هم اکنون آشنا شدن با تکنیک نرم‌افزارهای متناسب با آینده است. در آخرین کارگاه شما کلاس Bank را نوشتید. این کلاس می‌تواند با همه انواع BankAccount کار کند. حال وظیفه دارید نوع جدیدی از BankAccount به نام RewardsAccount ایجاد نمایید.



شکل ۷ ـ ۲
خروجی PsychiatristDriver

همچون حساب پس‌انداز RewardsAccount به موجودی حساب سود پرداخت می‌کند. با این حال برای افزایش میزان موجودی و تعداد موجودیها، بانک اقدام به برگزاری قرعه‌کشی و اعطای جوایز می‌کند.

در RewardsAccount تعداد و مقدار موجودیها حول و حوش مقدار مشخصی پول باید باشد. برای مثال فرض کنید که میزان موجودی باید ۵۰۰$ باشد. هر زمان که میزان موجودی بیش از ۵۰۰ دلار شود، بانک اقدام به دادن جایزه و یا امتیاز برای دریافت جایزه می‌کند.

نکته

RewardsAccount را ساده فرض کنید. در صورتی که میزان موجودی برای امتیاز ۵۰۰ دلار باشد و این میزان موجودی وجود داشته باشد، صاحب حساب ۱ امتیاز دریافت می‌کند و اگر موجودی ۳۰۰۰ دلار هم باشد، صاحب حساب باز هم ۱ امتیاز می‌گیرد.

با استفاده از متدهای BankAccount، کلاس RewardsAccount باید بتواند مکانیزمی جهت دریافت یا تنظیم تعداد امتیازات توسعه دهد. همچنین باید راهی وجود داشته باشد که حداقل میزان موجودی برای دریافت امتیاز را تنظیم کرد. برای این کار مناسب است که به روز پنجم برگردیم و نگاهی دوباره به SavingsAccount و BankAccount داشته باشیم. این کارگاه همچنین شامل RewardsAccountDriver و BankDriver به روز شده است. برای آزمایش کدهای نوشته خود می‌توانید از این دو کلاس استفاده کنید. BankDriver همچنین نشان می‌دهد که چگونه می‌توان انواع اشیاء جدیدی را به برنامه اضافه کرد بدون آنکه دیگر اشیاء برنامه را تغییر داد.

توجه

بخش بعدی راه حل کارگاه ۳ را ارایه می‌دهد. تا زمانی که کارگاه را انجام نداده‌اید، سراغ این بخش نروید.

حل و بحث

لیست ۷ـ۱۴ یک راه ممکن برای RewardsAccount ارایه می‌دهد.

لیست ۷-۱۴ RewardsAccount.java

```
public class RewardsAccount extends SavingsAccount {
    private double min_reward_balance;
    private int qualifying_deposits;

    public RewardsAccount( double initDeposit, double interest, double min ) {
        super( initDeposit, interest );
        min_reward_balance = min;
    }

    public void depositFunds( double amount ) {
        super.depositFunds( amount );
        if( amount >= min_reward_balance ) {
            qualifying_deposits++;
        }
    }
```

لیست ۷-۱۴ (ادامه)

```
    public int getRewardsEarned() {
        return qualifying_deposits;
    }

    public void resetRewards() {
        qualifying_deposits = 0;
    }

    public double getMinimumRewardBalance() {
        return min_reward_balance;
    }

    public void setMinimumRewardBalance( double min ) {
        min_reward_balance = min;
    }
}
```

کلاس RewardsAccount متد depositFunds() را جایگزین کرده تا از این طریق بتواند میزان موجودی و امتیازات را چک کند. همچنین متدهایی برای دریافت امتیازات موجودی در کلاس تعبیه شده است.

لیست ۷_۱۵ کلاس BankDriver به روز شده را نشان می‌دهد.

لیست ۷-۱۵ BankDriver.java

```
public class BankDriver {
    public static void main( String [] args ) {
        CheckingAccount  ca = new CheckingAccount( 5000.00, 5, 2.50 );
        OverdraftAccount oa = new OverdraftAccount( 10000.00, 0.18 );
        SavingsAccount   sa = new SavingsAccount( 500.00, 0.02 );
        TimedMaturityAccount tma = new TimedMaturityAccount( 10000.00, 0.06, 0.05 );

        Bank bank = new Bank();
        bank.addAccount( "CHECKING", ca );
        bank.addAccount( "OVERDRAFT", oa );
        bank.addAccount( "SAVINGS", sa );
        bank.addAccount( "TMA", tma );

        System.out.println( "Total holdings(should be $25500.0): $" + bank.totalHoldings() );
        System.out.println( "Total accounts(should be 4): " + bank.totalAccounts() );

        RewardsAccount ra = new RewardsAccount( 5000.00, .05, 500.00 );
        bank.addAccount( "REWARDS", ra );

        System.out.println( "Total holdings(should be $30500.0): $" + bank.totalHoldings() );
        System.out.println( "Total accounts(should be 5): " + bank.totalAccounts() );
```

لیست ۷-۱۵ (ادامه)

```
        bank.deposit( "CHECKING", 250.00 );
        double new_balance = bank.balance( "CHECKING" );
        System.out.println( "CHECKING new balance (should be 5250.0): $" + new_balance );
    }
}
```

برای استفاده از کلاس جدید حساب بانکی باید دو قدم را طی کنید.

در قدم اول باید نوع داده‌ای جدید (کلاس جدید) را ایجاد کنید. پس از ایجاد کلاس جدید، باید تغییری در برنامه بدهید تا برنامه از کلاس جدید استفاده کند.

در مورد RewardsAccount باید تابع main() تغییر پیدا کند تا از کلاس جدید استفاده نماید. پس در قدم دوم نیاز است که نمونه‌ای از کلاس جدید ایجاد شود و البته دیگر نیازی به ایجاد تغییری در برنامه نیست.

در زیر خواهید دید چگونه می‌توان برنامه‌ای پایدار نوشت به نحوی که هیچ نیازی به تغییر در کدهای برنامه نباشد.

کارگاه ٤ - مطالعه موردی - دستور switch در Java و قابلیت چندشکلی

Java همچون بسیاری از زبان‌های دیگر، مکانیزمی به نام switch ارایه می‌دهد. برای مثال متد day_of_the_week() را در نظر بگیرید:

```
public void day_of_the_week( int day ) {
    switch ( day ) {
        case 1:
            system.out.println( "Sunday" );
            break;
        case 2:
            system.out.println( "Monday" );
            break;
        case 3:
            system.out.println( "Tuesday" );
            break;
        case 4:
            system.out.println( "Wednesday" );
            break;
        case 5:
            system.out.println( "Thursday" );
            break;
        case 6:
            system.out.println( "Friday" );
            break;
        case 7:
            system.out.println( "Saturday" );
            break;
        default:
            system.out.println( day + " is not a valid day." );
            break;
    }
}
```

این متد تنها یک پارامتر می‌پذیرد: یک عدد صحیح که نشان دهنده شماره روز در هفته است. سپس از طریق دستور سوئیچ تمام شماره‌ها بررسی شده و سپس روز مناسب چاپ می‌گردد.

به طور کلی برای پیاده‌سازی منطق شرطی از دستور switch استفاده می‌کنیم. در منطق شرطی داده‌ای مورد آزمایش قرار می‌گیرد و در صورتی که شرط برقرار باشد، دستوراتی انجام می‌گیرد. هر شخص با یک پیش‌زمینه در برنامه‌نویسی با این منطق آشناست.

در صورتی که با منطق شرطی احساس راحتی می‌کنید، زمان آن است که با مطلب جدیدی آشنا گردید. منطق سوئیچ و کلاً منطق شرطی از لحاظ طراحی شیءگرا بسیار نامناسب است. در حقیقت این منطق از آنجا بد است که بسیاری از زبان‌های شیءگرا مکانیزم سوئیچ را ارایه نمی‌کنند. تنها خوبی منطق شرطی یک چیز است: کمک می‌کند تا طراحی ضعیف در OO را به راحتی تشخیص دهید. کلاً منطق شرطی به طور ذاتی از لحاظ OO بد انگاشته می‌شود. این منطق به صورتهای دیگر هم نمایان می‌شود. مثال زیر نمایش دیگری از این متد است:

```
public void day_of_the_week( int day ) {
    if ( day == 1 ) {
        System.out.println( "Sunday" );
    else if ( day==2 ) {
        System.out.println( "Monday" );
    else if ( day==3 ) {
        System.out.println( "Tuesday" );
    else if ( day==4 ) {
        System.out.println( "Wednesday" );
    else if ( day==5 ) {
        System.out.println( "Thursday" );
    else if ( day==6 ) {
        System.out.println( "Friday" );
    else if ( day==7 ) {
        System.out.println( "Saturday" );
    else {
        System.out.println( day + " is not a valid day.");
    }
}
```

اما مشکل استفاده از منطق شرطی در کجاست؟

شروط در تضاد با مفاهیم شیءگرایی هستند. چراکه در OO اجازه ندارید که از داده‌های درون یک شیء پرس‌وجو کنید و سپس با آنها کاری را انجام دهید. در عوض از شیء می‌خواهید تا با داده‌های درونش کاری، را انجام دهد. در مورد متد day_of_the_week()روز را می‌توانید از شیء دریافت کنید. در این حالت نیازی به پردازش بر روی فوجی از اطلاعات خام ندارید.

گاهی پیش می‌آید که هیچ راهی جز استفاده از دستورات کنترلی و شرطی نیست. با این ترتیب چگونه می‌توان طراحی ضعیف را تشخیص داد؟

راههای متعددی جهت تشخیص شروط خوب از بد وجود دارد. در صورتی که برنامه به نحوی بود که به ازای هر نوع داده‌ای جدید باید بلوک‌های سوئیچ و یا if/else را تغییر دهید، در این صورت با منطق شرطی بدی مواجه هستید. نه تنها این امر از لحاظ طراحی OO بد است بلکه از لحاظ نگهداری نرم‌افزار نیز با مشکل مواجه خواهید شد.

حل مشکل

متد زیر را در نظر بگیرید:

```
public int calculate( String operation, int operand1, int operand2 ) {
    if ( operation.equals( "+" ) ) {
        return operand1 + operand2;
    } else if ( operation.equals( "*" ) ) {
        return operand1 * operand2;
    } else if ( operation.equals( "/" ) ) {
        return operand1 / operand2;
    } else if ( operation.equals( "-" ) ) {
        return operand1 - operand2;
    } else {
        System.out.println( "invalid operation: " + operation );
    }
}
```

از این متد جهت انجام محاسبات می‌توان استفاده کرد. پارامترهای ورودی متد فوق عملگر مورد نظر و دو عملوند جهت انجام محاسبات است. چگونه می‌توان مشکل منطق شرطی را حل کرد؟ بسیار ساده است. با استفاده از اشیاء! برای حل مشکل کافی است با داده‌هایی که دستورات کنترلی با آنها کار می‌کنند شروع کرده و آنها را تبدیل به شیء نمایید.

در مورد مسأله فوق، نسبت به ایجاد اشیاء add (جمع)، subtract (تفریق)، multiply (ضرب) و divide (تقسیم) مبادرت کنید.

لیست ۷_۱۶ تا ۷_۲۰ حل مسأله فوق را نشان می‌دهد.

لیست ۷-۱۶ Operation.java

```
public abstract class Operation {
    public abstract int calculate( int operand1, int operand2 );
}
```

لیست ۷-۱۷ Add.java

```
public class Add extends Operation {
    public int calculate( int operand1, int operand2 ) {
        return operand1 + operand2;
    }
}
```

لیست ۷-۱۸ Subtract.java

```
public class Subtract extends Operation {
    public int calculate( int operand1, int operand2 ) {
        return operand1 - operand2;
    }
}
```

لیست ۷-۱۹ Multiply.java

```
public class Multiply extends Operation {
    public int calculate( int operand1, int operand2 ) {
        return operand1 * operand2;
    }
}
```

لیست ۷-۲۰ Divide.java

```
public class Divide extends Operation {
    public int calculate( int operand1, int operand2 ) {
        return operand1 / operand2;
    }
}
```

هریک از اپراتورها متد ()calculate را به شیوه خودش پیاده‌سازی کرده است. حال برای هریک از عملگرها یک شیء بخصوص داریم. حال می‌توانیم ()calculate اصلی را به صورت زیر بازنویسی کنیم:

```
public int calculate (Operation operation, int operator1. int operator2) {
    return  operation.calculate(operand1,operand2);
}
```

با تبدیل عملگرها به شیء توانسته‌ایم پایداری بسیار خوبی را ایجاد نماییم. در گذشته نیاز بود برای هر عملگر جدید، متد ()calculate را تغییر دهیم. این در حالی است که با تبدیل عملگرها به اشیاء، با اضافه شدن عملگر جدیدی دیگر نیازی به تغییر کدهای اولیه نیست.

تعریف مسأله

Java عملگری را ارایه می‌کند به نام instanceof. از طریق این عملگر می‌توانیم نوع یک شیء و یا نمونه ایجاد شده از آن را تشخیص دهیم.

```
String s = "somestring";
Object obj = s;
system.out.println(obj instacneof String));
```

کد ارایه شده در بالا، عبارت true (به معنای صحیح) را چاپ می‌کند. درواقع obj نمونه‌ای از String است. بسیاری از زبان‌های شیءگرا مکانیزمی شبیه آنچه Java ارایه داده است، فراهم کرده‌اند.

خوب، حالا کلاس جدید Payroll را که در لیست ۷ ـ ۲۱ ارایه شده است، در نظر بگیرید.

لیست ۷-۲۱ Payroll.java

```
public class Payroll {

    private int    total_hours;
    private int    total_sales;
    private double total_pay;

    public void payEmployees( Employee [] emps ) {
        for( int i = 0; i < emps.length; i ++ ) {
            Employee emp = emps[i];
            total_pay += emp.calculatePay();
            emp.printPaycheck();
        }
    }

    public void calculateBonus( Employee [] emps ) {
        for( int i = 0; i < emps.length; i ++ ) {
            Employee emp = emps[i];
            if( emp instanceof HourlyEmployee ) {
System.out.println("Pay bonus to " + emp.getLastName() + ", " + emp.getFirstName() + "
$100.00." );
            } else if ( emp instanceof CommissionedEmployee ) {
                int bonus = ( (CommissionedEmployee) emp ).getSales() * 100;
System.out.println("Pay bonus to " + emp.getLastName() + ", " + emp.getFirstName() + "
$" + bonus );
            } else {
                System.out.println( "unknown employee type" );
            }
        }
    }

    public void recordEmployeeInfo( CommissionedEmployee emp ) {
        total_sales += emp.getSales();
    }

    public void recordEmployeeInfo( HourlyEmployee emp ) {
        total_hours += emp.getHours();
    }
    public void printReport() {
        System.out.println( "Payroll Report:" );
        System.out.println( "Total Hours: " + total_hours );
```

لیست ۷-۲۱ (ادامه)

```
        System.out.println( "Total Sales: " + total_sales );
        System.out.println( "Total Paid: $" + total_pay );
    }
}
```

کلاس Payroll متدی به نام calculateBonus() دارد. این متد آرایه‌ای از نوع Employees دریافت کرده و مشخص می‌کند نوع هریک چیست و در ضمن میزان پاداش را برای هریک تعیین می‌کند. HourlyEmployee تنها ۱۰۰ دلار و CommissionedEmployee به ازای هر فروش ۱۰۰ دلار پاداش می‌گیرند.

وظیفه شما تغییر منطق شرطی در این مدت است. ابتدا با داده‌هایی شروع کنید که متد با آنها کار می‌کند. در این حالت دستورات کنترلی بر روی اشیاء مانور می‌دهند. پس مشکل چیست؟

به جای پرسش از شیء در مورد پاداش، متد داده‌هایی را از شیء پرسیده و سپس پاداش را بر مبنای آن محاسبه می‌کند. در حالی که متد باید از اشیاء در مورد داده‌هایشان پرسش کند.

می‌توانید کدهای نوشته شده را به کلاسهای Payroll، Employee، HourlyEmployee و CommissionedEmployee منتقل کنید. برای آزمایش راه حل می‌توانید از PayrollDriver استفاده کنید.

توجه بخش بعدی راه حل کارگاه ۴ را ارایه می‌دهد. تا قبل از تکمیل کارگاه به این بخش مراجعه نکنید.

حل و بحث

برای حل مسأله باید متد calculateBonus() را مستقیماً به کلاس‌های مشتق شده از Employee اضافه کنید. از آنجا که تمام زیرکلاسها باید این متد را داشته باشند، می‌توانید متد را به صورت مجرد به کلاس پایه اضافه کنید. لیستهای ۷ـ۲۲ تا ۷ـ۲۵ تغییرات لازم را نشان می‌دهند.

لیست ۷-۲۲ Employee.java

```
public abstract class Employee {

    public abstract double calculateBonus();

    //snipped for brevity, the rest stays the same
}
```

لیست ۷-۲۳ HourlyEmployee.java

```
public class HourlyEmployee extends Employee {

    public double calculateBonus() {
      return 100.00;
    }

    //snipped for brevity, the rest stays the same
}
```

لیست ۷-۲۴ CommissiondEmployee.java

```
public class CommissionedEmployee extends Employee {

    public double calculateBonus() {
      return 100.00 * getSales();
    }

    //snipped for brevity, the rest stays the same
}
```

لیست ۷-۲۵ Payroll.java

```
public class Payroll {

   public void calculateBonus( Employee [] emps ) {
      for( int i = 0; i < emps.length; i ++ ) {
         Employee emp = emps[i];
System.out.println("Pay bonus to " + emp.getLastName() + ", " + emp.getFirstName() + " $"
+ emp.calculateBonus() );
      }
   }

   //snipped for brevity, the rest stays the same
}
```

همانگونه که ملاحظه می‌کنید، دیگر از منطق شرطی خبری نیست!

نکته

نکات مرتبط با دستور سوئیچ (switch):

- از بکار بردن دستور switch بپرهیزید.
- به بلوک‌های بزرگ switch نگاه کرده و منطق آنها را بررسی کنید.
- از تغییرات پشت سر هم برحذر باشید. اگر یک تغییر نیازمند تغییرات شرطی زیادی باشد. با یک مشکل روبرو شده‌اید.
- دستور instanceof پرچم بزرگ قرمزی است!
- دستورات instanceof, switch, if/else نشانه‌های بدی هستند!

نکته

نکاتی برای تبدیل سوئیچ (switch):

- داده‌ها را تبدیل به شیء کنید.
- اگر داده‌ها خود، شیء بودند، متدی را به اشیاء اضافه کنید.
- از چک کردن اشیاء با instaceof بپرهیزید. در عوض برای این کار از قابلیت چندشکلی سود بجویید.

خلاصه

در درس امروز چهار کارگاه را به پایان رساندید. کارگاه ۱ شما را با اصول اولیه چندشکلی آشنا کرد. کارگاه ۲ باعث شد تا شما آنچه را در کارگاه قبل آموختید، اعمال کنید و کارگاه ۳ در انتها به این سؤال پاسخ گفت: نرم‌افزارهای آینده‌نگر کدامند؟ در انتها کارگاه ۴ شما را با منطق شرطی و نحوه پرهیز از آن آشنا کرد. همه این کارگاه‌ها یک هدف مشترک را دنبال می‌کردند: استفاده از قابلیت چندشکلی. درواقع کارگاهها آنچه راکه باید می‌دانستید تا مفاهیم را به عمل تبدیل کنید، به شما آموزش دادند. OO واقعی نیازمند طرز تفکری متفاوت در حوزه نرم‌افزار است. قدرت OO واقعی زمانی نمایان می‌شود که بتوانید به چند صورت مختلف فکر نمایید.

پرسشها و پاسخها

دیده شد که چندشکلی درونی نسبت به سربارگذاری مزایای بیشتری دارد. چرا که در این حالت تنها کافی است یک متد نوشته شود تا برای همه انواع مختلف کار کند. با این توصیف چرا باید از سربارگذاری استفاده کنیم؟

مواردی پیش می‌آید که سربارگذاری انتخاب بهتری است. متدی که تنها با چندشکلی درونی کار می‌کند، تنها می‌تواند پردازشهای مرتبط با اشیاء را انجام دهد. سربارگذاری اجازه می‌دهد از نام یک متد برای گروهی از متدها که آرگومانهایشان با یکدیگر ارتباطی ندارند، مورد استفاده قرار گیرد.

کارگاه

سؤالاتی که در زیر می‌آید، تنها برای افزایش فهم شما از مطالب ارایه شده تدوین شده است.

پرسشها

۱. **با توجه به راه حلهای ارایه شده برای کارگاهها، مثالی از متد سربارگذاری شده ارایه دهید.**
۲. **جایگزینی توابع (overriding) چه مشکلی را می‌تواند ایجاد کند؟**
۳. **چه مراحلی را باید طی کرد تا رفتار را در یک سلسله مراتب چندشکلی بتوان تغییر داد؟**
٤. **از روی کارگاهها مثالی از چندشکلی درونی ارایه دهید.**
۵. **چگونه منطق شرطی را در برنامه مشخص می‌کنید؟**
۶. **مزایای چندشکلی درونی نسبت به سربارگذاری چیست؟**
۷. **دو ارتباط میان اشیاء و داده‌ها در چیست؟**
۸. **منطق شرطی چه مشکلی را می‌تواند ایجاد کند؟**
۹. **چه چیزی مبین آن است که دستورات کنترلی/شرطی به کار رفته بد هستند؟**
۱۰. **چندشکلی را آنگونه که فهمیده‌اید، بیان کنید.**

تمرین‌ها

امروز تمرینی ندارید، به کارگاههایتان برسید!

روز ۸

آشنایی با UML

Object Oriented Programming

در هفته گذشته نظریه‌های بنیادی برنامه‌نویسی شیءگرا را آموختید. با این حال، فقط دانستن چند تکنیک و تعریف کسی را برای اعمال آنها در زمان نیاز آماده نمی‌کند. آیا فقط با نشان دادن آجر و فرغون به کسی و شرح طرز کار آنها برای او می‌توان شخص را برای ساختن یک خانه مأمور کرد؟ البته که نه! برنامه‌نویسی هم چندان متفاوت نیست. مهارت برنامه‌نویسی فقط با تمرین و تجربه به دست می‌آید. در این هفته خواهید آموخت چگونه از ابزارهای برنامه‌نویسی شیءگرا استفاده کنید.

در درس امروز زبان یکپارچهٔ مدلسازی Unified Modeling Language (یا UML) و نیز برخی روابط درون اشیاء را با هم بررسی خواهیم کرد. در درس امروز زبان مشترکی را خواهیم آموخت که برای توصیف مسأله و راه حل به کار خواهد رفت.

آنچه امروز خواهید آموخت:

- چرا باید UML را بشناسیم.
- چگونه کلاسها را در UML مدل کنیم.
- چگونه روابط مختلف بین کلاسها را مدل کنیم.
- چگونه همه چیز را به هم پیوند دهیم.

آشنایی با زبان مدلسازی یکپارچه

وقتی یک معمار خانه می‌سازد، بدون طرح و نقشه پیش نمی‌رود. این طرح و نقشه دقیقاً نما و ساختار خانه را نشان می‌دهد. هیچ چیزی به شانس یا تصادف واگذار نمی‌شود.

حال چند بار خود شما بدون طرح و نقشه برنامه نوشته‌اید؟ چقدر دچار مشکل شدید؟ UML سعی می‌کند نقشهٔ ساخت نرم‌افزار را در اختیار برنامه‌نویس قرار دهد. UML یک زبان مدلسازی استاندارد صنعتی است. این زبان دارای چندین نماد گرافیکی است که با استفاده از آنها می‌توان تمام معماری برنامه را تشریح کرد. برنامه‌نویس، معمار نرم‌افزار، و تحلیل کننده از زبان مدلسازی برای شرح گرافیکی طراحی یک نرم‌افزار استفاده می‌کنند.

واژه جدید زبان مدلسازی، مجموعه نمادهای گرافیکی مورد استفاده برای تشریح طراحی نرم‌افزار است. علاوه بر این، زبان مدلسازی دارای قوانینی برای تشخیص مدل‌های صحیح از ناصحیح است. این قوانین هستند که UML را به جای تنها مجموعه‌ای از اشکال و پیکان‌ها به یک زبان مدلسازی مبدل کرده‌اند.

زبان مدلسازی با یک فرایند یا روش‌شناسی متفاوت است. روش‌شناسی (Methodology) چگونگی طراحی نرم‌افزار را تعیین می‌کند. در حالی که زبان مدلسازی طرحی را نشان می‌دهد که با پی‌گرفتن روش‌شناسی ایجاد خواهد شد.

واژه جدید روش‌شناسی روال طراحی نرم‌افزار را مشخص می‌کند. زبان‌های مدلسازی این طرح را به صورت گرافیکی نشان می‌دهند.

UML تنها زبان مدلسازی موجود نیست. با این حال به عنوان استاندارد پذیرفته شده است. در هنگام مدلسازی نرم‌افزار، بهتر است از یک زبان معمول و عام استفاده شود. در این صورت توسعه دهندگان دیگر به راحتی و به سرعت نمودارهای طراحی را درک می‌کنند. در حقیقت سازندگان UML سه زبان مدلسازی رقیب را با هم ترکیب کرده‌اند، تا زبانی واحد و مشترک پدید آید.

واژه جدید UML یک زبان استاندارد صنعتی برای مدلسازی است. UML برای هریک از وجوه مختلف طراحی نرم‌افزار دارای نمادهای گرافیکی خاصی است.

نکته

> در این کتاب تنها به بخش‌هایی از UML اشاره خواهیم کرد که مستقیماً در طراحی به آنها نیاز داریم. لازم به ذکر است که زبان مدلسازی هیچ کمکی در رسیدن به نرم‌افزار نهایی نمی‌کند.

واژه جدید روش‌شناسی یا فرایند شرح چگونگی طراحی نرم‌افزار است. روش‌شناسی عمدتاً دارای یک زبان مدلسازی است.

تذکر

> UML مجموعه‌ای غنی از ابزارهای مدلسازی را ارایه می‌دهد. در نتیجه اطلاعات بسیار زیادی موجود هستند که می‌توان در مدل نرم‌افزار آنها را لحاظ کرد. توجه داشته باشید که لازم نیست در مدلسازی از تمام نمادها استفاده کنید. تنها پارامترهایی را به کار بگیرید که مورد نیاز هستند. هدف UML ساده کردن طراحی است.

مدلسازی کلاس‌ها

اگر دقیق نگاه کنید، خود کد پایین‌ترین سطح مستندسازی برنامه است. اگر کد کار کند، طراحی شما مستند شده است.

اگرچه کد خود کاملترین سند دربارهٔ طراحی است، اما برای دیگران (بخصوص آنهایی که با آن آشنا نیستند) بسیار مشکل است که در آن دست برند. علاوه بر این، مستندسازی برای کسی که با زبان برنامه‌نویسی مزبور آشنا نیست بی‌مفهوم خواهد بود.

در عوض باید از یک نوتاسیون (Notation) استفاده کنید که اجازه دهد طرای خود را طوری مستند کنید که دیگران هم بتوانند آن را با یک نظر متوجه شوند. در این صورت دیگران می‌توانند ساختار سطح بالای کلاسها را ببینند و فقط وقتی به جزییات بپردازند که واقعاً نیاز باشد. به عبارت دیگر علایم گرافیکی جزییات را کپسوله می‌کند تا بتوان ساختار سطح بالای برنامه را درک کرد.

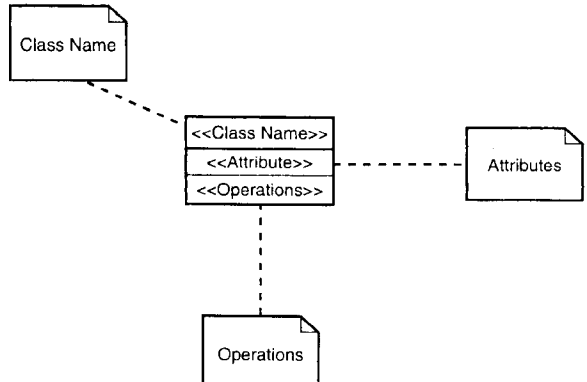
توجه تهیه مستندات جدای از کد نیازمند تعهد در به‌روزآوری آن در زمان تغییر کد است.

یک روش کمک UML در این مسأله فراهم آوردن مجموعه‌ای گسترده از نمادها برای شرح عملکرد کلاسها است. با استفاده از این علایم، دیگران می‌توانند به راحتی کلاسهای اصلی شکل دهندهٔ برنامه را مشاهده کنند. همچنانکه خواهید دید با UML می‌توان کلاسها را تعریف کرد و نیز روابط سطح بالای بین آنها را هم شرح داد.

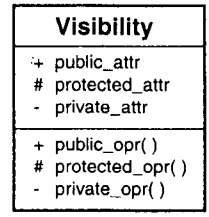
نمادهای بنیادی کلاس

در UML یک جعبه نمایانگر یک کلاس است. بالاترین جعبه همواره حاوی نام کلاس است. جعبهٔ وسطی حاوی صفاتی است که ممکن است موجود باشند و پایین‌ترین جعبه محتوی اعمال (Operation) است. تذکرات (Note) در مورد طراحی در جعبه‌هایی می‌آیند که گوشهٔ برگشته دارند. شکل ۸ـ۱ این ساختار بنیادی را نمایش می‌دهد.

نکته UML بین روال و عملیات (Method & Operation) فرق قایل می‌شود. یک عمل، سرویس یا خدمتی است که می‌توان آن را از کلاس درخواست کرد در حالی که روال یک پیاده‌سازی خاص آن عمل است.

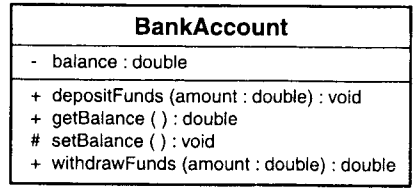


شکل ۸-۱
نماد کلاس UML



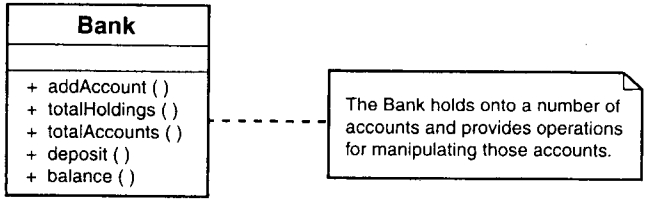
شکل ۸ـ۲
نمادهای UML برای مرئی بودن اعمال

درون یک مدل می‌توان از کاراکترهای ـ # و + استفاده کرد. این کاراکترها نمایانگر صفات یعنی مرئی بودن عملیات هستند. علامت - یعنی اختصاصی، # یعنی محافظت شده و + یعنی عمومی. (ر.ک. شکل ۲.۸) شکل ۸ـ۳ کلاس BankAccount معرفی شده در درس روزهای ۵ و ۷ را به طور کامل نشان می‌دهد.



شکل ۸ـ۳
شرح کامل یک کلاس

گاهی یک تذکر یا یادداشت (Note) به رساندن مفهوم که در غیر آن صورت نادیده گرفته می‌شد، کمک می‌کند. این یادداشت‌ها کاملاً از روی یادداشت‌های دنیای واقعی شبیه‌سازی شده‌اند.(شکل ۸-۴ را نگاه کنید)



شکل ۸ـ۴
یک مثال کامل از یک یادداشت

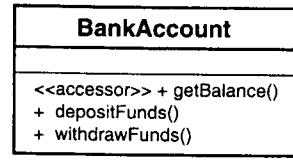
نمادهای پیشرفته کلاس

این قبیل نمادها برای ایجاد مدل‌های کامل‌تر مورد استفاده قرار می‌گیرند. با استفاده از کلیشه‌ها (Stereotype) می‌توان غنای فرهنگ واژگان UML را افزایش داد.

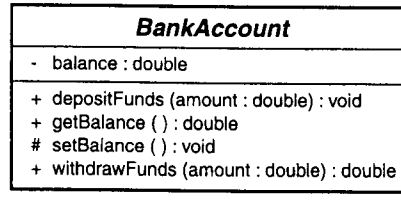
تعریف یک کلیشه، عنصری از UML است که به کاربر اجازه می‌دهد به غنای فرهنگ واژگان UML، بیافزاید. کلیشه یک کلمه یا عبارت است که بین دو علامت << >> قرار می‌گیرد. کلیشه را می‌توان در بالا یا کنار یک عنصر موجود قرار داد.

مثلاً در شکل ۸ـ۱ می‌توان کلیشهٔ <<Attribute>> ملاحظه کرد. این کلیشه نشان می‌دهد که در کجا باید صفات کلاس را وارد کرد. شکل ۸ـ۵ نمونه دیگری را نشان می‌دهد که دربارهٔ عمل (Operation) توضیحاتی می‌دهد.

اگر به یاد داشته باشید در درس روز هفتم کلاس BankAccount را دوباره به صورت مجرد تعریف کردیم. در شکل ۸ـ۶ نمادی معرفی شده که تجرید کلاس را نشان می‌دهد. در UML نام کلاس مجرد با حروف کج نوشته می‌شود.



شکل ۸ـ۵
کلیشه‌ای که درباره عملیات توضیح می‌دهد



شکل ۸ـ۶
نمایش کلاس مجرد

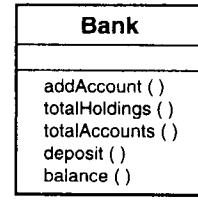
مدلسازی هدفمند کلاس‌ها

دو بخش قبلی نمادهای متفاوتی زیادی را معرفی کردند. حال باید بپردازیم به اینکه چگونه از آنها استفاده کنیم. همواره این پرسشها را در ذهن داشته باشید: «چه چیزی را می‌خواهم بیان کنم؟» و «برای چه کسی می‌خواهم مسأله را بیان کنم؟» هدف اساسی مدلسازی تبیین طراحی به بهترین و ساده‌ترین نحو ممکن است.

شاید هدف بیان رابط عمومی یک کلاس باشد. شکل ۸ـ۷ را ببینید. این شکل رابط عمومی Bank را بدون درگیر کردن بیننده با جزییات آرگومان‌ها شرح می‌دهد. این علایم اگر بخواهید فقط آنچه دیگر اشیاء با Bank می‌توانند انجام دهند را بیان کنید، کفایت می‌کند.

دوباره به شکل ۸ـ۳ رجوع کنید. نمودار کل صفات و روالهای (عام، اختصاصی و حفاظت شده) کلاس BankAccount را مستند می‌کند. چنین مدلی به کار معرفی کلاس به یک برنامه‌نویس دیگر می‌آید. یا در صورتی که در برنامه‌نویسی شیءگرا پیشرفت کنید شاید بتوانید به یک طراح یا معمار بدل شوید. در این صورت می‌توانید چنین مدلی را به یک توسعه دهنده نرم‌افزار بدهید تا کلاس را ایجاد کند.

سپس پاسخ پرسش «چگونه بدانم از کدام نماد استفاده کنم؟» این است که، بستگی دارد. هنگامی که یک فرد غیرفنی می‌پرسد که چه کار می‌کنید، باید به صورتی پاسخ دهید که او بتواند درک کند. هنگامی که یک همکار همین پرسش را مطرح می‌کند، عموماً پاسخی که می‌دهید فنی است. مدلسازی طراحی هم چندان متفاوت نیست. از فرهنگ واژگانی بهره بگیرید که مناسب کاری باشد که می‌خواهید انجام دهید.



شکل ۸ـ۷
نمادسازی ساده‌ای از Bank

نکته

تذکراتی برای مدلسازی مؤثر

- همواره پرسش «می‌خواهم چه چیزی را بیان کنم؟» را از خود بپرسید. پاسخ این پرسش دقیقاً نشان خواهد داد چه قسمتی را باید مدل کرد.

- همواره از خود بپرسید: «برای چه کسی می‌خواهم کار خود را شرح دهم؟» پاسخ روش مدلسازی را به شما دیکته خواهد کرد.
- همواره سعی کنید ساده‌ترین مدل ممکنی را طراحی کنید که هدف شما را برآورده کند.
- سعی کنید در پیچ و خم زبان مدلسازی گرفتار نشوید. اگرچه نباید پا را از روشهای استاندارد و قوانین فراتر بگذارید، اما در همان زمان نباید بگذارید دقت در این مسأله شما را از تکمیل طرحتان باز دارد. این خطر، خطری واقعی است. به خصوص وقتی تازه شروع به این کار می‌کنید. اگر مدل ۱۰۰٪ کامل نبود، نگران نشوید. تنها نگرانی شما باید از گویا نبودن مدل باشد.
- در نهایت به یاد داشته باشید که UML (یا هر زبان مدلسازی دیگر) تنها وسیله‌ای برای تبیین طراحی است. بعد از تکمیل طرح، تازه باید شروع به کدنویسی کنید.

مدلسازی روابط یک کلاس

کلاس‌ها در خلاء به سر نمی‌برند، بلکه دارای روابطی پیچیده با یکدیگر هستند. این روابط چگونگی ارتباط کلاس با کلاسهای دیگر را شرح می‌دهند.

واژه جدید یک رابطه، چگونگی ارتباط کلاسها با یکدیگر را شرح می‌دهد. در UML یک رابطه، اتصالی بین دو یا چند عنصر نمادین است.

UML سه نوع رابطه سطح بالای بین اشیاء را تشخیص می‌دهد:

- وابستگی (Dependency)
- تشریک (Association)
- تعمیم (Generalization)

هرچند ممکن است UML برای هرکدام از این روابط نمادی داشته باشد، اما این روابط منحصر به UML نیستند. UML فقط فرهنگ واژگان و نماد لازم را فراهم می‌کند. فهمیدن خود روابط مستقل از UML در جای خود ارزشمند است. درواقع اگر نمادها را نادیده بگیرید و فقط به مفهوم بپردازید، گامی به جلو برداشته‌اید.

وابستگی

وابستگی مهمترین رابطه بین اشیاء است. وابستگی یعنی اینکه یک شیء به مشخصات شیء دیگر وابسته است.

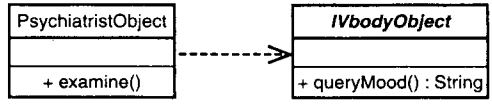
نکته مشخصات یعنی همان صفات یا روال‌ها

واژه جدید در یک رابطه وابستگی یک شیء به مشخصات شیء دیگر وابسته است. اگر مشخصات تغییر کنند، مجبور خواهید شد شیء وابسته را نیز تغییر دهید.

برگردیم به کارگاههای روز هفتم. می‌توان گفت PsychiatristObject به دو دلیل به MoodyObject وابسته است. اول آن روال ()examine از PsychiatristObject یک MoodyObject را به عنوان پارامتر می‌گیرد. دوم اینکه روال ()examine متد ()queryMood از MoodyObject را فراخوانی می‌کند. اگر نام یا پارامترهای

()queryMood تغییر کند، مجبور خواهیم شد روش فراخوانی آن در ()examine را تغییر دهیم. به همان صورت، اگر نام کلاس MoodyObject تغییر کند، مجبوریم در پارامترهای روال ()examine تغییر لازم را ایجاد کنیم.

شکل ۸ـ۸ نماد UML رابطهٔ وابستگی بین PsychiatristObject و MoodyObject را نشان می‌دهد.



شکل ۸ـ۸
یک رابطه وابستگی ساده

نکته

به آنچه شکل ۸ـ۸ بیان نمی‌کند، توجه کنید. عنصر PsychiatristObject در شکل تمام روال‌های شیء اصلی را ندارد. این حقیقت در مورد MoodyObject هم صادق است. در عوض، این مدل وابستگی فقط حاوی ویژگیهای لازم برای مدلسازی رابطه مزبور است.

به یاد داشته باشید که نمادسازی UML فقط برای بیان و انتقال اطلاعات است، نه برای اینکه تمام نمادهای ممکن را در آن به‌کار ببرید.

در برنامه‌نویسی شیءگرا همواره سعی می‌کنیم تا جای ممکن وابستگی‌ها را کاهش دهیم. با این حال حذف کامل آن‌ها غیرممکن است. تمام وابستگی‌ها هم یکسان نیستند. وابستگی به روابط عموماً مشکلی ندارد، در حالی که وابستگی به پیاده‌سازی تقریباً هیچگاه قابل قبول نیست.

تذکر

چه وقتهایی وابستگی‌ها را مدلسازی می‌کنید. معمولاً وقتی وابستگی‌ها مدل می‌شوند که می‌خواهیم نشان دهیم یک شی از دیگری استفاده می‌کند. مثلاً به عنوان آرگومان روال.

تشریک

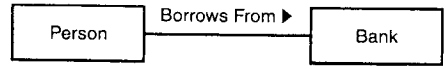
روابط تشریک اغلب عمیق‌تر از وابستگی‌ها هستند. روابط تشریک ساختاری هستند. تشریک یعنی یک شیء به شیء دیگر اتصال دارد یا حاوی نمونه‌ای از آن است.

واژه جدید

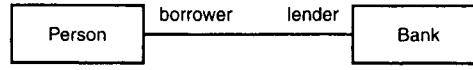
تشریک یعنی اینکه شیئی حاوی شیئی دیگر است. در اصطلاح UML، وقتی دو شیء به هم متصل باشند.

از آنجایی که دو شیء به هم پیوسته‌اند، می‌توان از یکی به دیگری منتقل شد. رابطهٔ تشریک بین بانک و شخص را که در شکل ۸ـ۹ نشان داده شده است را ملاحظه کنید.

شکل ۸ـ۹ نشان می‌دهد که شخص از بانک قرض می‌کند. در نمادسازی UML هر تشریکی نام مخصوص به خود را دارد. در این مورد تشریک Borrows From نام دارد. پیکان جهت رابطه را مشخص می‌کند.



شکل ۸ـ۹
تشریک بین فرد و بانک



شکل ۸ـ ۱۰
نقش‌ها در تشریک

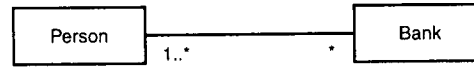
واژه جدید نام تشریک، نامی است که رابطه را شرح می‌دهد.

هر شیء در رابطه تشریک، نقشی (Role) دارد که در شکل ۸ـ ۱۰ نشان داده شده است.

واژه جدید در تشریک نقش شخص قرض کننده (Borrower) و نقش بانک قرض دهنده (Lender) است.

واژه جدید در تشریک، نقش، کاری است که شیء در رابطه انجام می‌دهد.

در پایان، تعدد (Multiplicity) نشان می‌دهد که چند شیء در رابطه نقش دارند.
شکل ۸ـ ۱۱ تعدد اشیاء موجود در تشریک بین Person و Bank را نشان می‌دهد.



شکل ۸ـ ۱۱
تعدد

این نمادسازی به آن معنی است که یک بانک می‌تواند یک یا بیشتر قرض کننده داشته باشد و یک شخص می‌تواند با صفر یا بیشتر بانک مرتبط باشد.

نکته

تعدد را می‌توان با عدد، لیست اعداد یا با * نشان داد. یک مقدار عددی یعنی همان تعداد اشیاء ـ نه کمتر و نه بیشتر ـ می‌توانند در رابطه شریک باشند. پس ۶ یعنی ۶ و فقط ۶ شی می‌توانند در رابطه شریک باشند.

* یعنی هر تعدادی از اشیاء می‌توانند در رابطه نقش داشته باشند. لیست اعداد، فاصله‌ای را مشخص می‌کند که هریک از اعداد آن می‌تواند تعداد اشیاء مربوطه باشد. مثلاً ۴..۱ یعنی ۱ الی ۴ شی می‌توانند در رابطه شرکت کنند. *..۳ هم یعنی تعداد اشیاء باید ۳ به بالا باشد.

تذکر

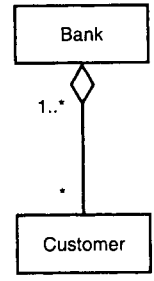
چه وقتی تشریک را مدل می‌کنیم؟ وقتی یک رابطه مالکیت برقرار باشد یا شی از دیگری استفاده کند. تشریک وقتی مورد استفاده قرار می‌گیرد که بخواهیم اینکه چه شی چه می‌کند را مدل کنیم.

UML دارای تعریف دو نوع تشریک متفاوت است: اجماع و ترکیب (Aggregation & Compositing). این دو نوع تشریک مدل را بهتر و سازگارتر می‌کنند.

اجماع

اجماع که نوع مخصوصی از تشریک است، رابطهٔ مالکیت (has – a) بین یک زوج همسان (peer) را مدل می‌کند. جفت، زوج همسان یا نظیر یعنی یک شیء که هم‌ارز دیگری باشد.

واژه جدید رابطه جزء/کل (whole/part) یعنی اینکه یک شیء حاوی دیگری باشد.



شکل ۸ـ۱۲
اجتماع بانک و مشتری

واژه جدید اجماع نوعی خاص از تشریک است که مالکیت در رابطه کل/جزء رامدل می‌کند.

در مبحث اجتماع، اهمیت (Importance) بدان معنی است که اشیاء می‌توانند مستقل از یکدیگر وجود داشته باشند. هیچ یک شیئی در رابطه از دیگری مهم‌تر نیست.

می‌توان دید که یک بانک می‌تواند چند شیء مشتری (Customer) داشته باشد. لوزی چسبیده به شیء Bankیعنی در این اجماع Bank کل و Customer جزء است. زیرا Bank شیئی است که در رابطه مالک است. Bank دارای چندین Customer است. در برنامه‌نویسی این یعنی اینکه Bank ممکن است آرایه‌ای از Customer داشته باشد.

نکته

یک لوزی خالی سمبل اجماع است. لوزی به شیء می‌چسبد که در رابطه نقش کل را دارد، یعنی به کلاس دیگر ارجاع می‌دهد. یک کل از چندین جزء ساخته شده است. در مثال قبلی Bank کل و Customerها جزء هستند.

مثال دیگر اتومبیل و موتور است. اتومبیل دارای موتور است. در این اجماع اتومبیل کل و موتور جزء می‌باشند.

از آنجایی که Bank و Customer مستقل از یکدیگر هستند، پس این دو هم‌ارز (peer) هستند. یعنی می‌توانند مستقل از یکدیگر وجود داشته باشند. این بدان معنی است که اگر بانک تعطیل شود، مشتری‌ها هم با آن تعطیل نمی‌شوند! به همان صورت مشتری هم می‌تواند پس‌انداز خود را به بانک دیگری منتقل کند، بدون اینکه بانک ورشکست شود.

اجماع بین اشیاء یاکلاسها هم مانند همین مثالهای زندگی روزمره عمل می‌کند. یک شیء می‌تواند حاوی یک شیء مستقل از آن باشد. Queue یا Vector هم مثالی دیگر از امکان مالکیت اشیاء دیگر از طریق اجماع است.

تذکر

اجماع چه زمانی مدلسازی می‌شود؟ وقتی هدف مدلسازی شرح ساختار رابطه دو نظیر باشد، اجماع صریحاً تبیین کننده یک رابطه جزء/کل است. توجه داشته باشید، در صورتی که هدف مدل کردن این باشد که چه شیء چه‌کاری انجام می‌دهد، بهتر است از تشریک ساده استفاده شود.

ترکیب

ترکیب (Composition) کمی از اجماع مستحکم‌تر است. زیرا رابطه‌ای بین دو شیء هم‌ارز نیست. لذا اشیاء از هم مستقل نیستند. شکل ۸ـ۱۳ یک رابطهٔ ترکیب را نشان می‌دهد.

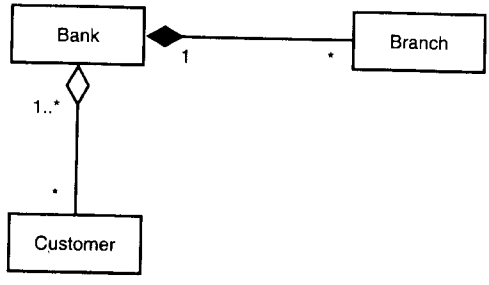
شکل ۸ـ۱۳
ترکیب Bank و شعبه‌های آن

یک بانک می‌تواند دارای شعبه (Branch) باشد. لوزی توپر این نوع رابطه (ترکیب) را نشان می‌دهد. همچنین وجود لوزی توپر به آن معنی است که رابطه از نوع مالکیت (has – a) است. در این حالت بانک دارای چند شعبه است.

نکته لوزی توپر سمبل ترکیب است و به شیئی می‌چسبد که نقش کل را دارد.

از آنجایی که رابطه از نوع ترکیب است، شعبه‌ها نمی‌توانند مستقل از بانک وجود داشته باشند. ترکیب یعنی در صورتی که بانک تعطیل شود، لزوماً شعبه‌ها هم تعطیل خواهند شد. در همین حال، رابطهٔ عکس هم برقرار نیست. یعنی شعبه‌های یک بانک می‌توانند تعطیل شوند، بدون اینکه الزامی در تعطیل بانک موجود باشد.

یک شیء می‌تواند همزمان هم در یک ترکیب و هم در یک اجماع شرکت کند. شکل ۸ـ۱۴ چنین رابطه‌ای را نشان می‌دهد.



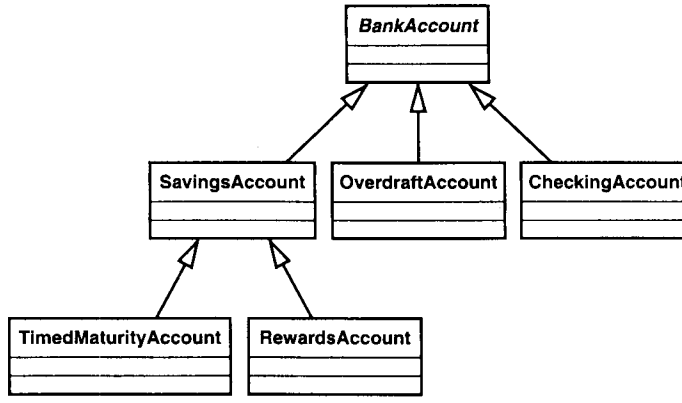
شکل ۸ـ۱۴
Bank هم می‌تواند در ترکیب و هم در اجتماع شرکت کند.

تذکر چه وقت ترکیب مدل می‌شود؟ دقیقاً مانند اجماع وقتی که هدف نشان دادن ساختار رابطه باشد. ترکیب صریحاً وجود رابطه کل/جزء را تبیین می‌کند.

برخلاف اجماع، ترکیب روابط کل/جزء بین دو هم ارز را مدل نمی‌کند. بلکه در ترکیب جزء به کل وابسته است. در مثال بانک، در این مورد می‌توان به فرض تعطیل شدن بانک اشاره کرد.

در اصطلاح برنامه‌نویسی وقتی شیء Bank نابود (Destroy) می‌شود Branchها هم با آن نابود می‌شوند. تذکر این نکته لازم است که در صورتی که مدل فقط برای نشان دادن تشریک به کار می‌رود، به جزییات زاید نپردازید و فقط خود تشریک را مدل کنید.

نکته به یاد داشته باشید که اجماع و ترکیب فقط تخصیص یا زیرنوع تشریک هستند. یعنی می‌توان هم ترکیب و هم اجماع را به عنوان تشریک ساده مدل کرد. این موضوع بستگی به هدف مدلسازی دارد.



شکل ۸ـ۱۵
سلسله مراتب وراثت
BankAccount

تعمیم

تعمیم رابطهٔ بین شیء عام (General) و شیء خاص (Specific) است. درواقع تعمیم (Generalization) همان وراثت است.

واژه جدید

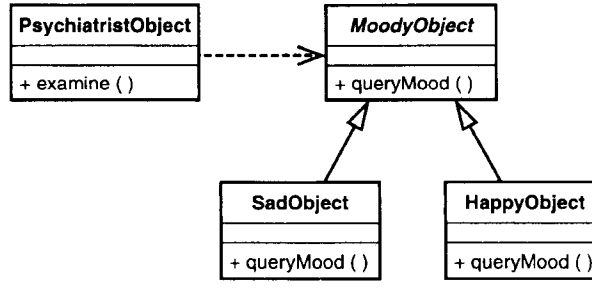
تعمیم نشاندهنده رابطه بین یک مفهوم عام و یک مورد خاص از آن است. وقتی رابطه تعمیم وجود داشته باشد، می‌توان کلاس فرزند را جایگزین کلاس والد کرد.

تعمیم در برگیرنده رابطهٔ همانی (Is – a) است که در درس روز چهارم به آن پرداختیم. همانطور که در درس روز چهارم آموختیم، رابطه همانی پایه‌ی روابط جانشین‌پذیری است. از طریق این روابط می‌توان از فرزندان به جای نیاکان استفاده کرد.

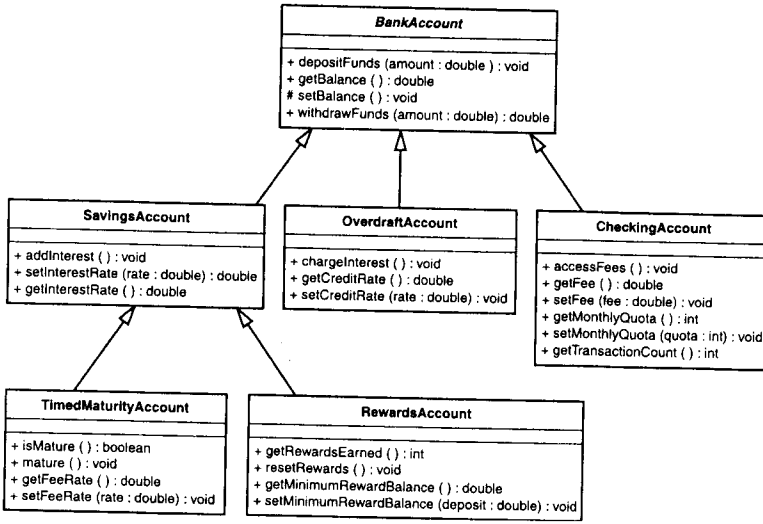
UML نمادهایی هم برای مدل کردن تعمیم دارد. شکل ۸ـ۱۵ نشان می‌دهد که چگونه می‌توان سلسله مراتب وراثت BankAccount را مدل کرد. یک خط توپر که به یک پیکان توخالی ختم می‌شود، نماد تعمیم در UML است.

جمع‌بندی

اکنون که اصول اولیه مدلسازی کلاسها را با هم بررسی کردیم، می‌توانیم شروع به ایجاد مدلهایی گویا کنیم. در شکل ۸ـ۸ نمونهٔ ساده‌ای از وابستگی را مدل کردیم. با استفاده از بقیهٔ آنچه آموختیم می‌توانیم مدل را کمی گویاتر کنیم. شکل ۸ـ۱۶ مدل توسعه یافته را نشان می‌دهد.



شکل ۸ـ۱۶
مدل گویاتر وابستگی



شکل ۸ـ۱۷
مدل کاملتر سلسله مراتب وراثت BankAccount

در شکل ۸ـ۱۶، یک تعمیم به مدل افزوده شده، لذا می‌توان با یک نگاه فهمید که کدام کلاسها را می‌توان در این رابطه جایگزین MoodyObject کرد.

به همان صورت در شکل ۸ـ۱۷ سلسله مراتب وراثت شکل ۸ـ۱۵ را بسط داده‌ایم.

با نگاهی به این سلسله مراتب می‌توان فهمید که کدام کلاس چه چیزی به سلسله مراتب می‌افزاید. چنین مدلی برای دیگر توسعه دهندگان کمک بزرگی در فهم کد می‌کند.

تمام این مدل‌ها یک عنصر مشترک دارند. هر کدام از آنها تنها حاوی اطلاعات و نمادهای لازم برای انتقال ایدهٔ آن است. هدف این مدل هم به هیچ وجه استفاده از تمام نمادهای UML نیست.

علاوه بر این، تمام این مدلها عناصر متفاوتی از UML را با هم ترکیب می‌کنند. از این طریق می‌توان مدلهایی کاملاً گویا ایجاد کرد.

خلاصه

امروز اصول اولیهٔ مدلسازی کلاس‌ها و روابط را آموختیم. بعد از درس امروز شما باید قادر باشید مدلهای ساده‌ای با استفاده از UML ایجاد کنید.

UML نمادهایی برای مدل کردن کلاس‌ها و روابط بین آنها فراهم کرده است. UML دارای نمادهایی برای سه نوع رابطه است:

- وابستگی
- تشریک
- تعمیم

علاوه بر این UML دو نوع تشریک را می‌شناسد: ترکیب و اجماع. با ترکیب تمام این عناصر می‌توان مدلهایی کامل و گویا از کلاسها و روابط بین آنها ایجاد کرد. مهارت در UML برای مستندسازی و انتقال نکات طراحی به دیگران بسیار مفید است.

پرسش‌ها و پاسخ‌ها

آیا می‌توان هر سه نوع رابطه را در یک مدل با هم استفاده کرد؟

آری. مدل می‌تواند حاوی هر ترکیبی از روابط باشد. مدل برای تشریح روابط بین کلاس‌ها است.

چگونه از UML استفاده می‌کنید؟ آیا ابزارهای خاصی موجود هستند؟

UML را به هر صورتی که بخواهیم، می‌توانیم مورد استفاده قرار دهیم. نمودارها را می‌توان روی ابزارهای مدلسازی، وایت برد یا کاغذ رسم کرد. انتخاب رسانه بستگی به شرایط مسأله دارد.

ابزارهای مدلسازی کامپیوتری وقتی مفید هستند که بخواهیم طراحی را به صورت رسمی مستند کنیم.

کارگاه

پرسشها

۱. **UML چیست؟**

۲. **تفاوت بین روش‌شناسی و زبان مدلسازی کدام است؟**

۳. **رابطه بین Employee و Payroll در کارگاه روز هفتم کدام است؟**

٤. **به مدل شکل ۸ ـ ۱۵ بادقت نگاه کنید. فقط با استفاده از اطلاعات مدل، درباره MoodyObject چه‌می‌توان گفت؟**

۵. **در کارگاههای روز هفتم، مثالی از وابستگی بیابید.**

۶. **کاراکترهای +، # و – در UML به چه کار می‌آیند؟**

۷. **در درس روز دوم رابط زیر را مشاهده کردید**

Queue با عناصری که در آن قرار می‌گیرند چه رابطه‌ای دارد:

```
public interface Queue{
   public void enqueue(Object obj);
   public Object dequeue();
   public boolean isEmpty();
   public Object peek();
}
```

۸. **در درس روز سوم، کارگاه ۳، کلاس Deck چند کارت تولید می‌کند. در این مثال چه نوع رابطه مالکیتی وجود دارد؟**

۹. **چگونه مجرد بودن کلاس یا متد (روال) را نشان می‌دهیم؟**

۱۰. **هدف نهایی مدلسازی چیست؟ نتایج این هدف کدامند؟**

۱۱. **تشریک، ترکیب و اجماع را توضیح دهید.**

۱۲. **هریک از روابط خودآزمایی ۱۱ در چه مواردی به کار می‌روند؟**

تمرین‌ها

۱. کلاس Queue پرسش ۷ را مدل کنید.

۲. رابطه ترکیب زنبور عسل/کندو را مدل کنید.

۳. رابطه Bank و BankAccount در روز هفتم، کارگاه ۲ را مدل کنید.
٤. رابطه تشریک بین یک بقال و یک مشتری را مدل کنید. نقش‌ها، تعدد و وابستگی‌ها را مشخص کنید.
۵. سلسله مراتب کارمند از کارگاه ۲ درس روز هفتم را مدل کنید. در مدل خود، آنچه هر کلاس به سلسله مراتب می‌افزاید را بیان کنید.
۶. سلسله مراتب وراثت PesonalityObject از درس روز ششم را مدل کنید.

روز ۹

مقدمه‌ای بر تحلیل شیءگرا (OOA)

Object Oriented Programming

دیروز فرا گرفتید که چگونه طرحهای کلاسهایتان را با استفاده از مدلهای کلاس، بصری (Visualize) نمایید. دیدید که چگونه مدلهای کلاسی می‌تواند دیگر برنامه‌نویسان را جهت فهم طراحی‌تان کمک کند و از این طریق اشیاء و رابطه آنها با هم به راحتی تشخیص داده شوند. زبان‌های مدلسازی نظیر UML به شما و دیگر برنامه‌نویسان کمک می‌کند تا از طریق یک زبان مشترک با یکدیگر صحبت کنید.

با این حال یک سؤال همچنان باقی است: چگونه نرم‌افزارهای شیءگرا طراحی می‌کنید؟ به طور ساده مدلها نمایی از طراحی‌تان را نمایش می‌دهند. مدلها کمکی برای فهم مسأله و یا حل مشکل نمی‌کنند. درواقع مدلها نتیجه طراحی نرم‌افزار هستند.

در خلال درس دو روز آینده در مورد تحلیل شیءگرا (Object Oriented Analysis – OOA) و طراحی شیءگرا (Object Oriented Design) مطالبی را فرا خواهید گرفت. OOA روشی شیءگرا برای فهم مسأله است. درواقع از OOA کمک می‌گیریم تا بتوانیم مسأله‌ای را که با آن مواجه هستیم، را بشناسیم. پس از فهم مسأله، طراحی راه حل آغاز می‌شود و این نقطه شروع OOD است. امروز با مفاهیم و کاربرد OOA سروکار خواهید داشت.

آنچه امروز خواهید آموخت

- مراحل توسعه نرم‌افزار
- چگونه OOA در فهم مسأله به ما کمک می‌کند.
- چگونه می‌توان با استفاده از مفهوم موارد استفاده (Use Case) فهم درستی از مسأله به دست آورد.
- چگونه می‌توان از UML برای نمایشی کردن تحلیلهایتان استفاده کرد.
- چگونه می‌توان مدل دامنه (Domain) ساخت.
- با چیزهایی که در خلال OOA ساخته می‌شوند، چه کاری می‌توان انجام داد.

مراحل توسعه نرم‌افزار

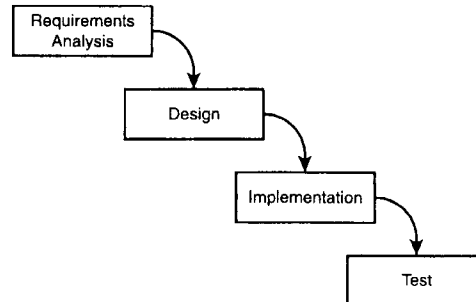
راههای بسیار زیادی برای توسعه نرم‌افزار پیش روی برنامه‌نویسان است. با این حال یک تیم توسعه نرم‌افزار باید از یک روش از پیش تعیین شده و مشخص جهت نوشتن نرم‌افزار استفاده کنند. اگر هریک از برنامه‌نویسان از روشی منحصر به خودش جهت نوشتن برنامه استفاده کند، در آخر چیز در خور توجهی ساخته نمی‌شود. متدولوژیهای نرم‌افزاری یک روش مشترک برای نوشتن نرم‌افزار در اختیار برنامه‌نویسان قرار می‌دهند. هر متدولوژی عموماً شامل یک زبان مدلسازی نظیر UML و تعدادی فرایند (process) است.

واژه جدید مراحل توسعه نرم‌افزار نمایانگر مراتبی است که باید برای توسعه نرم‌افزار طی شود.

یک مثال آشنا برای مراحل توسعه نرم‌افزار، روند آبشاری (waterfall) است.
همانگونه که شکل ۹ ـ ۱ نشان می‌دهد، روند آبشاری به صورت سری و در یک جهت هستند. این روند شامل چهار مرحله جدا از هم می‌باشد:

۱. تحلیل نیازمندیها (Requirement Analysis)
۲. طراحی (Design)
۳. پیاده‌سازی (Implementation)
٤. آزمایش (Test)

زمانی که از شیوه آبشاری استفاده می‌کنید به ترتیب از هر مرحله به مرحله بعدی می‌روید. با اتمام هر مرحله بازگشتی به مراحل گذشته در کار نیست. درواقع در این روش سعی بر عدم تغییر در مراحل گذشته است.



شکل ۹ ـ ۱
روند آبشاری

نتیجه آنکه نرم‌افزار تولید شده در انتها ممکن است آنچه که شما و یا مشتری انتظار دارید، نباشد!

با تحلیل یک مسأله باید راه حلی برای آن طراحی نمود. پس از آن نوبت به پیاده‌سازی می‌رسد. فهم بهتر مسأله بستگی کامل به نحوه تحلیل و طراحی‌تان دارد. توجه داشته باشید که نیازمندیها ممکن است در خلال توسعه نرم‌افزار تغییر کنند (مثلاً ممکن است رقیب شما قابلیت تازه‌ای به نرم‌افزارش اضافه کرده باشد که شما هم مایل باشید آن را اضافه نمایید). متأسفانه شیوه آبشاری در اینگونه مواقع کمکی به شما نمی‌کند.

این کتاب بر روی متدولوژی خاصی تأکید ندارد. تنها یکی از روشها موجود است که در توسعه شیءگرا مورد تأکید قرار گرفته است: روش تکراری، در این کتاب به این متدولوژی تکیه شده است.

فرایندهای تکراری

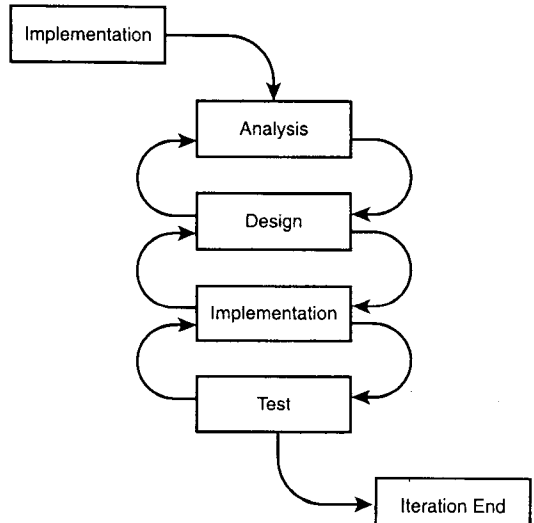
این شیوه کاملاً در جهت مخالف شیوه آبشاری قرار دارد. در این شیوه اجازه دارید هر زمان که احساس نیاز پیدا کردید، به مراحل قبل بازگشته و تغییراتی را اعمال کنید. درواقع در این شیوه می‌توان با یک روش تکرار و پیشرفت تغییراتی را به هنگام توسعه نرم‌افزار ایجاد کرد.

روش تکراری (Iterative Approach)

برخلاف روند آبشاری، روش تکرار اجازه می‌دهد که به صورت مکرر به مراحل قبلی بازگشته و هر مرحله از توسعه را دوباره طی کرد. برای مثال اگر متوجه شدید آنچه که طراحی کرده‌اید به خوبی جواب نمی‌دهد می‌توانید به مراحل قبل بازگشته و تغییراتی در طراحی و یا حتی تحلیل انجام دهید. شکل ۹ـ۲ این روش را نشان می‌دهد.

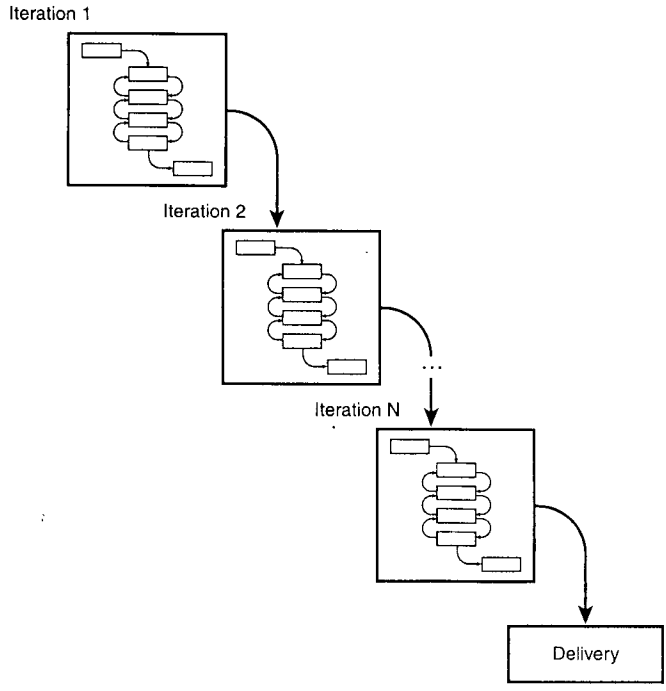
روش افزایشی (Incremental Appreach)

زمانی که روش تکرار را دنبال می‌کنید، اینگونه نیست که یک تکرار بسیار طولانی که کل برنامه را می‌سازد، به اتمام برسانید. بلکه روش تکراری مراحل توسعه را به تکرارهای کوچک تقسیم می‌کند و تنها باید این تکرارهای کوچک را به اتمام رساند. شکل ۹ـ۳ این روش را نشان می‌دهد.



شکل ۹ـ۲
یک تکرار

شکل ۹ـ۳
روش تکراری



در هر تکرار تکه‌ای از برنامه تکمیل می‌شود. درواقع بهبودی در عملکرد برنامه حاصل می‌شود. هر بهبود یک ویژگی جدید در برنامه و یا تغییری مناسب در ویژگیهای موجود است. بدین ترتیب هر تکرار مقصود خاصی را دنبال می‌کند و در انتهای تکرار باید بهبود قابل توجهی در عملکرد مشاهده شود.

تصور کنید در حال ساخت یک پخش‌کننده MP3(MP3 Player) هستید. در خلال یک تکرار از پروژه، باید کدهای مربوط به پخش یک فایل MP3 را به اتمام برسانید. برای تشخیص اینکه برنامه به درستی کار می‌کند یا خیر، باید یک فایل MP3 را بتواند باز کند و سپس آن را پخش نماید. در تکرار بعدی قابلیت انتخاب فایل را به برنامه می‌دهید. در هر تکرار میزان پیشرفت پروژه را بسنجید. در انتهای اولین تکرار باید پخش یک آهنگ را بشنوید و در انتهای دومین تکرار مکانیزمی باید پیش‌بینی شده باشد تا به صورت پویا (انتخاب فایل) آهنگی را پخش کند.

با اتمام هر تکرار میزان پیشرفت پروژه نمایان است. به عبارت دیگر اگر بخواهید همه مراحل را یکجا طی کنید بسیار مشکل است که میزان پیشرفت پروژه را بتوان تعیین کرد. عدم پیشرفت پروژه و ندانستن آنکه در مرحله بعد چه کاری باید انجام داد، باعث تکه تکه شدن و سپس مرگ پروژه خواهد شد.

توجه

در روش تکراری باید مرتب مراحل پیشرفت پروژه را چک کرد و مراقب بود پروژه از مسیر اصلی خود (که همانا حل مسأله است) دور نشود. OOA و OOD ابزارهایی را برای این منظور در اختیار قرار می‌دهند.

پیشرفت مستمر اعلام کننده آن است که شما در مسیر درستی قرار دارید. اگر بخواهید کل پروژه را به یکباره انجام دهید تا اتمام پروژه نخواهید فهمید آیا روشی که در پیش گرفته‌اید درست بوده است یا خیر؟ بازگشت به گذشته و تصحیح اشتباهات صورت گرفته بخصوص در مواقعی که کل پروژه با هم دیده شده و

توأمان انجام می‌گیرد، بیشترین خسارت را به تیم برنامه‌نویس وارد می‌کند. چراکه در این حالت، کل برنامه باید از ابتدا نوشته شود! با استفاده از روش تکراری، این میزان خسارت به حداقل مقدار خود می‌رسد. چرا که کل پروژه، به قسمتهای کوچکتر تقسیم شده و در هر مرحله یکی از این قسمتها به پایان می‌رسد. از آنجا که هر مرحله نمایانگر مقداری پیشرفت و اعلام پسخورد(Feedback) از همان مرحله و مراحل گذشته است، پیدا کردن اشتباهات ساده‌تر صورت می‌پذیرد. با این ترتیب هر چه اشتباهات سریعتر پیدا شوند بازگشتن به مراحل قبلتر و تصحیح خطاها سهلتر خواهد بود. اگر مراحل تکرار راکوچکتر انتخاب کنیم، در صورتی که با اشتباهی مواجه شویم زمان کمتری را از دست خواهیم داد.

توجه

اگر مشکل و اشتباهی به اصل خود تکرار بازگردد، هزینه‌ای که باید متقبل شوید بسیار زیاد و خسارت وارده زیانبار خواهد بود و ممکن است به کیفیت محصول نهایی خسارت جبران‌ناپذیری وارد نمایید.

یک متدولوژی سطح بالا

در این کتاب از متدولوژیهایی استفاده شده است که تکنیک آنها قبلاً به صورت موفقیت‌آمیزی اثبات شده است. متدولوژی فرایند تکرار شامل چهار مرحله است:

- تحلیل
- طراحی
- پیاده‌سازی
- آزمایش

نکته

پس از مرحله تست و آزمایش دو مرحله ارایه و نگهداری نیز وجود دارد که در چرخه عمر هر پروژه نرم‌افزاری دارای اهمیت ویژه‌ای هستند. با این حال به دلیل اهدافی که این فصل دنبال می‌کند، این دو مرحله حذف شده‌اند. در درس امروز تمرکز بر روی تحلیل، طراحی، پیاده‌سازی و تست و آزمایش است.

متدولوژیهای واقعی اغلب شامل مراحل بیشتری هستند. ولی با توجه به اینکه این اولین باری است که با این مباحث روبرو می‌شوید بهتر دیدیم که فعلاً بر روی این چهار مرحله تأکید بیشتری داشته باشیم.

تحلیل شیءگرا (OOA)

تحلیل شیءگرا (OOA) فرایندی است که در طی آن با صورت مسأله‌ای که باید نسبت به حل آن مبادرت نمایید، آشنا می‌شوید. پس از اتمام تحلیل، باید نیازمندیهای مسأله را تشخیص داده و دامنه مسأله‌ای راکه با آن روبرو هستید تعیین کنید.

واژه جدید

تحلیل شیءگرا فرایندی است که از یک روش شیءگرا کمک گرفته و از آن برای حل مسأله استفاده می‌کند. در انتهای تحلیل باید دامنه مسأله و نیازمندیهای آن در قالب کلاسها و اشیاء و تعاملات آنها با یکدیگر روشن گردد.

برای طراحی یک راه حل باید بفهمید که کاربران چگونه با آن سیستم کار خواهند کرد. جواب این سؤال پنیازمندیهای سیستم را تعیین می‌کند. درواقع نیازمندیها مشخص می‌کنند کاربران از سیستم چه کاری را انتظار دارند و چه جوابی از سیستم باید دریافت نمایند.

واژه جدید سیستم یک عبارت OOA برای گروهی از اشیاء در حال تعامل است. می‌توان به این گروه از اشیاء مدلی از مسأله و یا سیستم اطلاق نمود.

این اشیاء نمونه‌های در حال اجرای کلاسهایی هستند که از کلاسهای پایه و یا مجرد موجود در دامنه مسأله مورد مطالعه مشتق شده‌اند. تحلیل مسأله مزیت دیگری هم دارد: آشنایی بیشتر با دامنه مسأله که مطالعه آن باعث شناسایی اشیایی است که به طور صحیح سیستم را مدل می‌کنند.

OOA همچنان که از نامش پیداست یک روش شیءگرا برای تحلیل نیازندیهاست. OOA با در اختیار گرفتن روشی مبتنی بر شیءگرایی و مدلسازی اشیاء و روابط آنها با یکدیگر اقدام به حل مسأله می‌نماید. دو نوع مدل اصلی وجود دارد. مدل کاربرد (Use Case Model) رابطه کاربر با سیستم را تشریح می‌کند. مدل دامنه (Domain Model) یک نمای کلی از سیستم را در اختیار می‌گذارد. با استفاده از مدل دامنه شناسایی اشیاء موجود در سیستم صورت می‌گیرد. مدل دامنه‌ای که به خوبی ایجاد شده باشد بسیاری از مسایل و مشکلات در همان حوزه را حل می‌کند.

استفاده از مدل کاربرد برای استنتاج کاربردهای سیستم

برای تحلیل یک مسأله اولین قدم شناخت نحوه استفاده و تعامل کاربران با سیستم است. درواقع این نحوه استفاده کاربران است که نیازمندیهای سیستم و نحوه ایجاد آن را به برنامه‌نویس دیکته می‌کند. با فراهم آوردن همه نیازمندیهای کاربران یک سیستم مناسب و قابل استفاده تولید می‌گردد.

واژه جدید نیازمندیها درواقع خواص و ویژگیهایی هستند که سیستم باید دارای آنها باشد تا مسأله حل شده تلقی گردد.

واژه جدید یک روش تشخیص کاربریها استفاده از تحلیل کاربردها (Use Case Analysis) است. از این طریق تعریف کاربردهای سیستم صورت می‌گیرد و نحوه کار یک کاربر با سیستم تشریح می‌گردد.

واژه جدید تحلیل کاربردها فرایند تشخیص و کشف کاربردها از طریق تعدادی سناریوی از پیش نوشته شده توسط کاربران خبره و یا کاربران فعلی سیستم است.

واژه جدید مدل کاربرد درواقع نحوه تعامل کاربران یک سیستم با خود سیستم را مشخص می‌کند. به عبارت دیگر نحوه استفاده کاربران از سیستم (از دیدگاه کاربران)

ایجاد کاربردها یک فرایند تکراری است. درواقع برای این منظور باید مراحلی طی گردد. برای تعریف یک کاربرد باید:

۱. عاملها را شناسایی کنید.
۲. یک لیست ابتدایی از کاربردها تهیه کنید.
۳. کاربردها را نامگذاری کرده و یا آنها را دوباره تعریف کنید.
٤. هر کاربرد را به صورت مجموعه‌ای از رخدادها تعریف کنید.
۵. کاربردها را مدلسازی کنید.

نکته

نباید کاربردها را بدون مطالعه ایجاد کرد بلکه در زمان ایجاد کابردها با مصرف کنندگان اصلی سیستم (که همان مشتریان نرم‌افزار) هستند باید در ارتباط بود. خواسته‌های مشتریان مهمترین چیز برای تشخیص کاربردها است مگر آن که در حال نوشتن نرم‌افزاری برای شخص خودتان باشید.

مشتریان درواقع خبرگان دامنه مسأله هستند. مشتریان بهتر از برنامه‌نویس می‌دانند چه خواص و ویژگیهایی باید در نرم‌افزار قرار داده شوند. همیشه از دانش و راهنمایی آنها حداکثر استفاده را بکنید.
از کاربرانتان بخواهید سناریو و مراحل کاریشان را مشخص کرده و سیستم ایده‌آل خود را توضیح دهند.

توجه

قبل از آنکه مباحث امروز را ادامه دهید لازم به ذکر است مثال ارایه شده تحلیل کاملی از یک سایت وب را ارایه نمی‌دهد. در عوض هدف از ارایه این مثال آموزش مراحلی است که برای تحلیل باید آنها را طی کنید. به همین منظور بسیاری از کاربردها نادیده انگاشته شده‌اند.

شناسایی عاملها

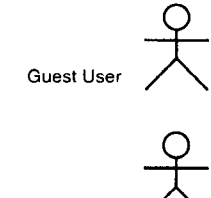
اولین قدم در تعریف کاربردها، تعریف عاملهایی است که با سیستم کار خواهند کرد.

واژه جدید

یک عامل هر موجودیتی است که با سیستم در حال تعامل است. در این صورت یک فرد، یک سیستم کامپیوتری و یا حتی یک دلفین ممکن است عامل به حساب آیند!
باید از مشتریان خود بپرسید کاربران سیستم چه کسانی هستند؟ این مسأله باید ابعاد زیر را مشخص کند:

- چه کسی به طور اخص با سیستم کار خواهد کرد؟
- آیا سیستمهای دیگری نیز وجود دارند که از این سیستم استفاده کنند؟ به طور مثال آیا کاربران غیرانسانی نیز وجود دارند؟
- آیا این سیستم با دیگر سیستمها در حال ارتباط است؟ برای مثال آیا بانک اطلاعات وجود دارد که باید با سیستم تلفیق شود؟
- آیا سیستم نسبت به محرکهای غیرانسانی باید پاسخگو باشد؟ مثلاً آیا سیستم در روزهای خاصی از ماه کار خاصی باید انجام بدهد؟

برای نمونه یک فروشگاه اینترنتی را در نظر بگیرید. این فروشگاه به کاربران مهمان (guest) اجازه دیدن کالاها، قیمت آنها و دریافت اطلاعات بیشتر را می‌دهد و به کاربران ثبت شده به غیر از موارد فوق اجازه خرید از فروشگاه را نیز می‌دهد.
با این توضیحات کوتاه دو نوع عامل شناسایی می‌شوند: کاربران مهمان و کاربران ثبت شده. هر جفت این عاملها با سیستم در حال تعامل هستند. شکل ۹ ـ ۴ نماد UML برای یک عامل را نشان می‌دهد: شکل یک شخص همراه با یک نام. باید برای هریک از عاملها نامی مجزا تعریف کرد.



شکل ۹ـ۴
عاملهای UML

توجه برای هریک از عاملها نامی انتخاب کنید که آن را از دیگر عاملها مجزا کند. نامگذاری صحیح از اهمیت به‌سزایی برخوردار است. نامها باید ساده بوده و یادآوری آنها آسان باشد.

یک کاربر سیستم می‌تواند نقش چندین عامل را ایفا کند. هر عامل درواقع یک نقش است. برای مثال کاربری ممکن است ابتدا به صورت مهمان وارد سایت شده و سپس با استفاده از یک نام کاربری ثبت شده به همراه کلید رمز به عنوان کاربر ثبت شده وارد سایت شود.

نکته یک کاربر می‌تواند در حین تعامل با سیستم نقشهای متفاوتی را داشته باشد. هر عامل یک نقش را تشریح می‌کند که کاربر در حین کار با سیستم می‌تواند آن نقش را ایفا کند.

توجه در ابتدای تعریف موارد کاربرد، یک لیست مقدماتی از عاملها تهیه کنید و البته بیش از اندازه مته به خشخاش نگذارید! در مرحله اول شناسایی همه عاملها بسیار مشکل است. در عوض عاملهایی که برای شروع کار لازم هستند را شناسایی کرده و دیگر عاملها را به تدریج اضافه کنید.

با شناسایی عاملها زمان آن فرا رسیده است که نسبت به تعریف موارد کاربرد اقدام کنیم.

تهیه فهرستی مقدماتی از موارد کاربرد

برای تعریف موارد کاربرد نیاز به پرسش تعدادی روال است. با عاملها این کار را شروع می‌کنیم. باید بپرسیم هریک از عاملها چه کاری در سیستم انجام می‌دهند؟

در مورد یک فروشگاه اینترنتی دو نوع کاربر وجود دارد: مهمان و ثبت شده. هریک از این عاملها چه کاری انجام می‌دهند؟

کاربران مهمان قادر به انجام موارد زیر هستند:

۱. مرور کاتالوگ محصولات و کالاها
۲. جستجو در فهرست کالاها
۳. جستجو به دنبال یک آیتم مشخص
٤. جستجو در سایت
۵. اضافه کردن آیتمهایی به کارت خرید و مشخص کردن تعداد هریک
۶. مشاهده قیمت هریک از کالاها
۷. تغییر تعداد هریک از آیتمها در کارت خرید
۸. مشاهده محصولات جدید و عامه‌پسند
۹. مشاهده لیست انتخابهای برتر دیگر کاربران
۱۰. درخواست اطلاعات بیشتر در مورد یک کالا

کاربران ثبت شده قادر به انجام موارد زیر هستند:

۱. همه مواردی که کاربران مهمان قادر به انجام آن هستند.
۲. انجام خرید
۳. اضافه کردن آیتمی به لیست کالاهای برتر

٤. مشاهده فهرست توصیه‌های شخص
۵. نگهداری حساب
۶. عضویت در اطلاع‌رسانی سایت
۷. استفاده از مزایای ویژه کاربران عضو
۸. دنبال کردن نحوه انجام سفارشها
۹. عضویت در لیستهای پستی گوناگون
۱۰. لغو یک سفارش

نکته می‌توان موارد کاربرد دیگری به فهرست فوق افزود. با این حال موارد فوق برای مقاصد ما کفایت می‌کند.

در حین انجام و تعیین موارد کاربرد باید این سؤال را از خود بپرسید که چگونه یک عامل می‌تواند نقش خود را عوض کند؟ در مورد یک فروشگاه اینترنتی کاربر مهمان در یکی از دو صورت زیر می‌تواند تبدیل به یک کاربر ثبت شده گردد:

- با ارایه نام کاربری و رمز عبور وارد سیستم شود (log in)
- با پر کردن فرم عضویت (sign in)

همچنین کاربر ثبت شده می‌تواند به صورت زیر تبدیل به یک کاربر مهمان شود:

- خروج از سیستم توسط لینک sign out

در انتها چیزهای مختلفی را باید در نظر داشته باشید که کاربران با آنها سروکار دارند. در اینجا محصولات و کالاها، اطلاعات در مورد حسابهای کاربران و لیستهای مختلفی از محصولات و تخفیفها وجود دارد. چگونه این چیزها را می‌توان وارد سیستم کرد؟ چه کسی کالای جدیدی را به سیستم اضافه می‌کند و یا انواع قدیمی را حذف و یا ویرایش می‌کند؟

این سیستم به عامل سومی هم احتیاج دارد: مدیر سایت (Administrator). با توجه به مطالب مطرح شده در بالا می‌توان وظایف مدیر سایت را به صورت زیر لیست کرد:

۱. اضافه کردن، ویرایش و حذف محصولات
۲. اضافه کردن، ویرایش و حذف محرکها
۳. به‌روز کردن اطلاعات حسابها

طرح برخی از پرسشها، پرسشهای دیگری را به دنبال دارد. برای مثال چه کسی لیست محصولات عمومی و عامه‌پسند را به‌روز می‌کند؟ چه کسی لیستهای پستی را تهیه کرده و آنها را ارسال می‌کند؟ یک عامل چهارم، خود سیستم، موارد فوق را انجام خواهد داد!

نامگذاری و تعریف مجدد موارد کاربرد

حال با تهیه یک لیست ابتدایی از موارد کاربرد، زمان تعریف مجدد و بازنگری آن است. درواقع باید با نگاهی دقیقتر مواردی از کاربردها را که نیاز به جداسازی و یا ترکیب دارند را مشخص نماییم.

جداسازی موارد کاربرد

هریک از موارد کاربرد باید یک هدف عمده داشته باشند. اگر مورد کاربردی را پیدا کردید که وظایف متعددی را بر عهده دارد بهتر است آن را به دو یا چند مورد کاربرد تقسیم کنید. برای مثال مورد کاربردی زیر را در نظر بگیرید:

کاربران مهمان می‌توانند آیتمهایی را به کارت خرید خود اضافه کرده و تعداد هریک از آیتمها را مشخص نمایند.

می‌توان مورد فوق را به دو قسمت تجزیه کرد:

- کاربران مهمان می‌توانند آیتمهایی را به کارت خرید خود اضافه نمایند.
- کاربران مهمان می‌توانند برای هر آیتم در کارت خرید، تعداد آن آیتم را مشخص کنند.

البته از آن جهت که این دو مقوله به یکدیگر مرتبط هستند می‌توانید از جداسازی آنها صرفنظر کنید. موارد کاربرد بسیار شبیه کلاسها هستند. فرد می‌تواند مورد کاربردی را داخل دیگری استفاده کند. بنابراین اگر مورد کاربردی نیاز به مورد کاربر دیگری داشته باشد تا کارش را به اتمام برساند، می‌تواند از آن استفاده نماید.

همچنین یک مورد کاربرد می‌تواند رفتارهای مورد کاربر دیگری را توسعه دهد. به عنوان نتیجه می‌توانید رفتارهای مشترک را داخل یک مورد کاربرد قرار داده و باقی موارد کاربرد را بر اساس آن ایجاد کنید. به عنوان مثال به جمله «کاربران عضو می‌توانند اقدام به خرید نمایند.» توجه کنید. می‌توان از طریق یک مورد کاربرد خاص که سفارش هدیه است اقدام به سفارش کالا یا محصولی نماید.

ترکیب موارد کاربرد

هیچ احتیاجی به موارد کاربرد با استفاده‌های تکراری نیست. در این صورت می‌باید موارد کاربرد با اهداف تکراری را با یکدیگر ترکیب نمود.

دو مورد کاربرد زیر را در نظر بگیرید:

- کاربران مهمان می‌توانند در فهرست محصولات اقدام به جستجو نمایند.
- کاربران مهمان می‌توانند اقدام به جستجوی آیتم خاصی نمایند.

همانگونه که ملاحظه می‌فرمایید جمله دوم گونه متفاوتی از حالت کلی بیان شده در جمله اول است.

در این حالت مورد کاربرد تنها در پارامتر جستجو تفاوت دارد. بهتر آن است که در این حالات شخص اقدام به ترکیب موارد کاربرد نماید. درواقع یک گونه همانند نمونه ایجاد شده‌ای از کلاس است. کلاس BankAccount را به عنوان نمونه به خاطر بیاورید. شیء BankAccount با موجودی ۱۰۰۰۰ $ با شیء BankAccount با موجودی ۱۰۰ $ تنها در میزان موجودی با یکدیگر اختلاف دارند. در حالی که هر دو از نوع BankAccount هستند. تمام تفاوتها بین دو نمونه از کلاس BankAccount در میزان و مقدار خواص و صفات آنهاست. موارد کاربرد هم به همین شیوه هستند.

موارد کاربرد نتیجه

پس از اتمام مرور و تصحیح موارد کاربرد، باید برای هریک نامی انتخاب کرد. همانگونه که برای عاملها این

کار را انجام دادیم. نامگذاری باید به نحوی باشد که ایجاد سردرگمی نکند. به عبارت دیگر روشن و واضح باشد. فهرست زیر موارد کاربرد نهایی را برای کاربران مهمان و ثبت شده (پس از جداسازی و یا ترکیب) نشان می‌دهد.

۱. مرور فهرست کالاها
۲. جستجو در فهرست کالاها
۳. جستجو در سایت
٤. اضافه کردن آیتمی به کارت خرید
۵. دیدن قیمت کالاها
۶. تغییر مقدار کالاها در کارت خرید
۷. دیدن فهرست کالاهای منتخب
۸. مرور فهرست انتخابی کالاها
۹. درخواست اطلاعات بیشتری برای هریک از کالاها
۱۰. سفارش خرید
۱۱. نگهداری سفارشات
۱۲. اضافه کردن آیتمی به فهرست انتخابی
۱۳. به‌روز کردن حساب شخصی
۱٤. عضویت در سایت
۱۵. اعمال محرکها
۱۶. ورود به سایت
۱۷. خروج از سایت
۱۸. تطابق لیست با فهرست سفارش داده شده

در این مرحله فهرست موارد کاربرد تهیه شده است. زمان آن رسیده است که هریک از موارد کاربرد به طور کامل تشریح گردد.

تعریف هریک از موارد کاربرد به صورت سلسله مراتبی از رخدادها

لیست خلاصه‌ای از موارد کاربرد تنها قسمتی از داستان است! درواقع هریک از موارد کاربرد کار بیشتری را طلب می‌کند. برای مثال سفارش کالا را در نظر بگیرید. کاربر نمی‌تواند سفارش یک کالا را در یک مرحله انجام دهد. درواقع سلسله مراتبی از مراحل باید طی شود تا سفارش یک کالا انجام بگیرد.

سلسله مراتب مراحلی که کاربر باید طی کند تا یکی از موارد کاربرد کامل شود، سناریو (Scenario) نامیده می‌شود. یک مورد کاربرد مجموعه‌ای از سناریوهای مختلف است.

واژه جدید

یک سناریو سلسله مراتبی از رخدادهایی است که بین کاربر و سیستم صورت می‌گیرد.

به عنوان قسمتی از تحلیل موارد کاربرد، باید سناریوهای هریک از آنها را مشخص کنید.

خوب اجازه دهید این کار را با مورد کاربرد سفارش کالا شروع کنیم. در ابتدا بهتر است این مورد کاربرد را در یک پاراگراف توصیف کنیم.

کاربر ثبت شده اقدام به بررسی آیتمهای سفارشی داده شده در کارت خرید خود می‌کند. سپس اطلاعات

مربوط به محل تحویل کالاها و اجناس توسط کاربر وارد می‌گردد. سپس سیستم جمع کل هزینه‌ای که باید پرداخت شود را به همراه آیتمهای موجود نمایش می‌دهد. در صورتی که همه چیز درست بود، مشتری درخواست ادامه کار می‌دهد. در این حالت سیستم از مشتری درخواست می‌کند اطلاعات مربوط به نحوه پرداخت هزینه سفارشات را وارد کند. زمانی که این امر از طرف کاربر (مشتری) صورت گرفت، سیستم اقدام به بررسی اطلاعات (منجمله تصدیق کاربر و بررسی کارت اعتباری) می‌نماید. در انتها همه این اطلاعات به صورت یکجا به کاربر نشان داده شده و یک پیغام الکترونیکی به صندوق پستی وی ارسال می‌گردد.

موارد جالب چندی در طی مراحل مورد کاربردی فوق دیده می‌شود. اول آنکه هیچ مطلبی در مورد نحوه پیاده‌سازی مطالب بالا دیده نمی‌شود. دوم آنکه از این طریق می‌توان شرایط قبل و بعد از مورد کاربرد را تشخیص داد.

واژه جدید

شرایط قبل (Preconditions) شرایطی هستند که باید از قبل موجود باشند تا مورد کاربردی شروع گردد. شرایط بعد (Postconditions) نتایجی هستند که پس از اجرای موارد کاربرد حاصل می‌شوند.

نکته

یکی از مشکلاتی که با اینگونه سیستمها مواجه هستید آن است که عموماً موارد کاربرد را از طرف کاربران سیستم دریافت نمی‌کنید بلکه آنها را از طرف کسانی دریافت می‌کنید که خواهان آن هستند تا شما برایشان نرم‌افزار را تهیه کنید. به خاطر داشته باشید که بسیاری از برنامه‌های کاربردی تحت وب و برنامه‌هایی که دیگر مشتریان با آن روبرو هستند نیازمند آن هستند تا شما از محل کار خود بیرون رفته و عملاً با گروههای مشخصی (برای درک عینی آنچه که می‌خواهند) کار کنید.

در اینجا شرایط قبلی که باید برقرار باشد تا این مورد کاربرد رخ دهد، اضافه شدن کالاهایی به کارت خرید است. درواقع مورد کاربردی سفارش کالا تنها با وجود آیتمهایی که در کارت خرید است اتفاق می‌افتد. شرایط بعدی چیزی نیست جز سفارش واقعی کالا! به عبارت دیگر مراحلی که باید طی شود تا کالا به دست مشتری برسد.

در طی مراحلی که به انتهای تکمیل مورد کاربردی مانده است، باید مسیرهای دیگر را نیز در نظر داشت. مثلاً در مورد کاربردی سفارش کالا ممکن است کاربر قبل از اتمام کار (پرداخت پول) از خرید کالا منصرف شود و یا آنکه مرحله تصدیق هویت (در مورد کارت اعتباری) ممکن است موفقیت‌آمیز نباشد.

پس از تکمیل مورد کاربردی نوبت به نوشتن آن می‌رسد. یک راه کلی برای این منظور نوشتن مراحلی است که باید قدم به قدم طی شود. پس از نوشتن قدمها، شرایط قبل و بعد مورد کاربردی و راههای دیگر نیز باید مدنظر قرار داده شوند. مورد کاربردی سفارش کالا را دوباره مدنظر قرار دهید:

- سفارش کالا

۱. کاربران ثبت شده امکان مرور لیست کالاهای سفارش شده را دارند.
۲. کاربران ثبت شده اطلاعات مربوط به نحوه خرید (مثل کارت اعتباری) را باید وارد کنند.
۳. سیستم، جمع کل کالاهای خریداری شده را نمایش می‌دهد.
٤. کاربران ثبت شده اطلاعات مربوط به تحویل کالا را باید وارد کنند.
۵. سیستم کاربر و اطلاعات وارد شده در مورد خرید را تصدیق می‌کند.
۶. سیستم سفارش داده شده را نهایی کرده
۷. و سپس به آدرس پست الکترونیکی کاربر پیغامی را مبنی بر دریافت اطلاعات لازم ارسال می‌کند.

- شرایط قبل از سفارش
  کارت خرید نباید خالی باشد
- شرایط بعد از سفارش
  وجود سفارش در سیستم
- مسیر دیگر: لغو سفارش

در طی مراحل ۱ تا ۴ کاربر امکان لغو سفارش را دارد. در این حالت کاربر به صفحه اصلی (Home page) بازگردانده می‌شود.

در مرحله ۵ اگر تصدیق هویت کاربر با مشکلی مواجه شد، کاربر می‌تواند اطلاعات خود را دوباره وارد کند.

باید برای هریک از موارد کاربرد چنین مراحلی طی شود. به طور کلی تعریف سناریوهای مختلف به شما کمک می‌کند تا نحوه تعامل با سیستم را به طور کامل استخراج نمایید.

نکته

به هنگام نوشتن موارد کاربرد تنها اطلاعاتی را به کار ببندید که برایتان مفید خواهد بود. دقیقاً همان کاری که برای مدل کردن کلاسها انجام می‌دادیم. تنها اطلاعاتی را اضافه کنید که شما را به مقصودتان می‌رساند.

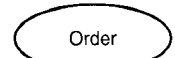
تنها از پیش شرایطی استفاده کنید که برای شروع مورد کاربرد لازم است. از شرایط زیاد و غیرضروری پرهیز کنید. برای نوشتن موارد کاربرد راههای مختلف را آزمایش کنید. استاندارد خاصی در این زمینه وجود ندارد.

نمودارهای مورد کاربرد (UseöCase Diagram)

همچنانکه UML روشی را برای مستندسازی و طراحی کلاس در اختیار گذاشته است، روشی را برای دریافت و نمایش موارد کاربرد آماده کرده است. این روش نمودارهای مورد کاربرد نمودارهای تعامل و نمودارهای فعالیت است. هریک از آنها موارد کاربرد مختلفی را نمایش می‌دهند.

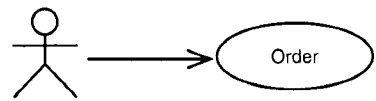
نمودارهای مورد کاربرد رابطه بین موارد کاربردی و همچنین رابطه بین موارد کاربردی و عاملها را مدل می‌کنند. می‌توان از روی توضیح متنی یک مورد کاربردی، کاربرد آن را تا حدودی فهمید. از طریق یک نمودار می‌توان رابطه یک مورد کاربردی را با دیگر موارد فهمید.

شکل ۹ ـ ۴ نشان می‌دهد چگونه می‌توان یک عامل را مدل کرد. شکل ۹ ـ ۵ نماد UML را برای مورد کاربردی سفارش کالا نشان می‌دهد.

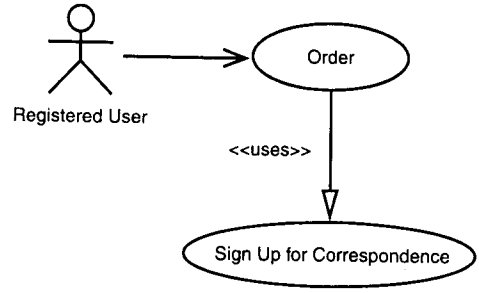


شکل ۹ـ۵
مورد کاربردی در UML

با قرار دادن یک عامل و یک مورد کاربردی در کنار یکدیگر دیاگرام یا نمودار مورد کاربردی ایجاد خواهد شد. شکل ۹ ـ ۶ نمودار مورد کاربردی سفارش کالا را نشان می‌دهد.



شکل ۹ـ۶
مورد کاربردی سفارش کالا



شکل ۹ـ۷
رابطه بین موارد کاربردی

این نمودار بسیار ساده است. اما با مشاهده آن می‌توان فهمید که چگونه کاربر ثبت شده از مورد کاربردی سفارش کالا استفاده می‌کند.

نمودارها می‌توانند پیچیده‌تر هم باشند. درواقع می‌توان در نمودارها رابطه بین موارد کاربردی را هم مشخص کرد.

همانگونه که در شکل ملاحظه می‌کنید مورد کاربردی سفارش کالا از موارد کاربردی عضویت در سایت استفاده کرده است. به عنوان قسمتی از فرایند عضویت کاربر مختار است که جهت دریافت پیغامها و پستهای الکترونیکی نیز ثبت نام کند.

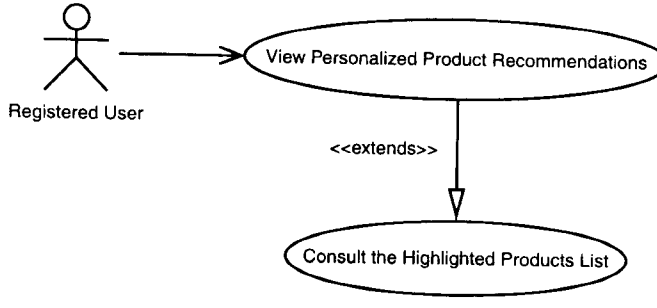
شکل ۹ ـ ۸ نوع دوم رابطه را که رابطه توسعه است نشان می‌دهد.

مشاهده توصیه‌های شخصی در مورد کالاهای مختلف حالت نمایش ساده و کلی کالاها را توسعه می‌بخشد. در این حالت کاربران ثبت شده (عضو) فهرستی از کالاهای مرتبط با اقلام خریداری شده و یا بیشترین مورد خرید و یا بیشترین درخواست را می‌توانند مشاهده نمایند. همانند کلاسها می‌توان موارد کاربردی مجرد (انتزاعی) داشت. یک مورد کاربردی مجرد موردی است که موارد کاربردی دیگر از آن استفاده می‌نمایند و یا از آن طریق توسعه می‌یابند، اما هیچگاه به طور مستقیم توسط عاملی مورد استفاده قرار نمی‌گیرند.

نمودارهای تعاملی

نمودارهای مورد کاربرد کمک خوبی برای مدلسازی روابط بین موارد کاربردی هستند. نمودارهای تعاملی کمک خوبی برای دریافت و تعاملات بین عاملهای مختلف دخیل در سیستم هستند.

حال اجازه دهید یکی از موارد کاربردی قبلی را که دیده‌اید توسعه دهید. با اضافه کردن یک عامل شروع می‌کنیم: نماینده سرویس مشتریان. اغلب پیش می‌آید که کاربران عضو رمز عبور خود را فراموش می‌کنند. بنابراین نیاز به نماینده‌ای برای مشتریان است تا در اینگونه مواقع مشکلات کاربران را حل نماید. مورد



شکل ۹ـ۸
رابطه توسعه

کاربردی جدیدی می‌سازیم. رمزهای عبور فراموش شده: یک کاربر عضو با مرکز پشتیبانی مشتریان تماس‌گرفته و اعلام می‌کند که رمز عبور خود را فراموش کرده است. قسمت پشتیبانی نام کامل کاربر و دیگر اطلاعات وی را دریافت کرده و سؤالاتی از وی می‌پرسد تا هویت وی به طور کامل احراز شود. پس از طی چندین مرحله، قسمت پشتیبانی رمز عبور قبلی را حذف کرده و رمز جدیدی را ایجاد می‌کند. سپس رمز عبور جدید از طریق آدرس پست الکترونیکی کاربر به وی ابلاغ می‌شود.
مورد کاربردی فوق به صورت زیر توصیف می‌شود:

- فراموشی رمز عبور

۱. کاربر عضو با قسمت پشتیبانی تماس می‌گیرد.
۲. کاربر عضو نام کامل خود را اعلام می‌کند.
۳. قسمت پشتیبانی اطلاعات کامل مشتری را دریافت می‌کند.
٤. کاربر عضو به تعدادی از پرسشهای قسمت پشتیبانی پاسخ می‌گوید.
۵. قسمت پشتیبانی رمز عبور جدیدی می‌سازد.
۶. کاربر رمز جدید را از طریق پست الکترونیک دریافت می‌کند.

- شرایط قبل

  فراموشی رمز عبور توسط کاربر عضو

- شرایط بعد

  رمز جدید از طریق پست الکترونیک به کاربر ابلاغ می‌شود.

- مسیرهای متفاوت: هویت احراز نشد

  کاربر ممکن است در جواب به پرسشهای قسمت پشتیبانی دچار مشکل شود و مرحله ۴ را با موفقیت طی نکند. در این حالت تماس قطع می‌شود.

- مسیرهای متفاوت: کاربر پیدا نشود.

  در صورتی که در مرحله ۲ کاربری با نام ذکر شده پیدا نگردد، قسمت پشتیبانی اقدام به ثبت نام تماس گیرنده در سایت می‌نماید.

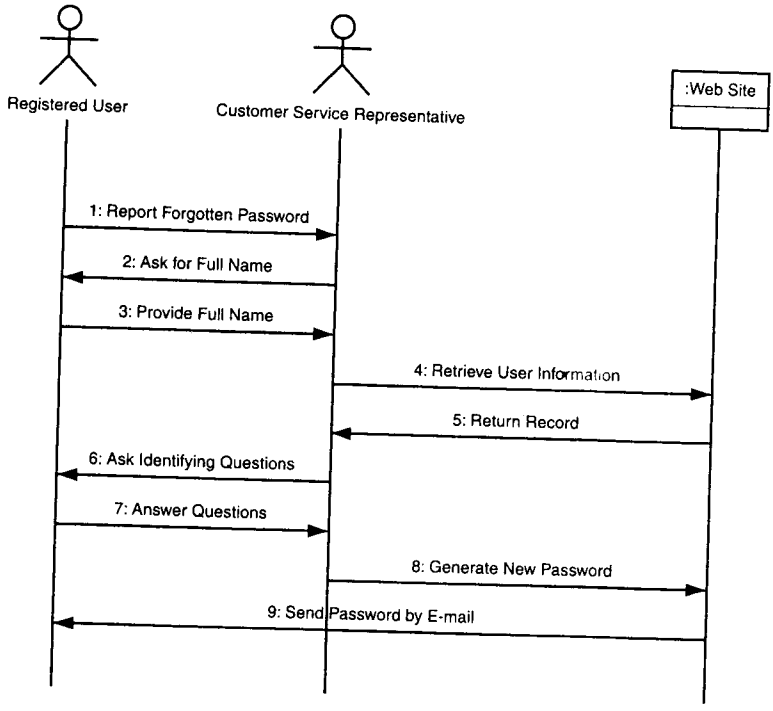
دو نوع نمودار تعاملی وجود دارد: نمودارهای متوالی (Sequence Diagrams) و نمودارهای همکاری (Collaboration Diagrams). هریک از نمودارها را در زیر بررسی می‌کنیم.

نمودارهای متوالی

یک نمودار متوالی تعاملات بین کاربر عضو، قسمت پشتیبانی و سایت وب را مدل می‌کند. از نمودارهای متوالی زمانی که بخواهیم توجه خود را به مجموعه‌ای از رخدادهای متوالی بر روی یک مورد کاربردی در طول زمان معطوف سازیم، استفاده می‌کنیم. شکل ۹ ـ ۹ نمودار متوالی برای مورد کاربردی فراموشی رمز عبور را نشان می‌دهد.

همانگونه که در شکل مشاهده می‌شود، نمودار متوالی رخدادهای مابین هریک از عاملها و سیستم (وب سایت) را نشان می‌دهد. هریک از شرکت کنندگان در مورد کاربردی در بالای نمودار نشان داده می‌شوند یا به صورت یک مستطیل و یا به شکل یک فرد (با این حال هر دو را به یک صورت ـ مستطیل ـ نیز می‌توان نشان داد).

شکل ۹ـ۹
نمودار متوالی فراموشی رمز عبور



از هریک از مستطیلها (و یا افراد) خطی که به خط عمر معروف است به سمت پایین کشیده می‌شود. خط عمر میزان موجودیت هریک از شرکت کنندگان را نشان می‌دهد. بنابراین اگر یکی از عاملها در خلال مورد کاربردی بیرون رود، خط عمر با آخرین فلش خود پایان می‌پذیرد. با خروج هریک از عاملها می‌توان گفت که عمر وی به پایان رسیده است.

واژه جدید

خط عمر، خط پیوسته‌ای است که از هریک از مستطیل‌ها در نمودار متوالی به سمت پایین کشیده می‌شود. خط عمر میزان عمر (موجودیت) هریک از اشیاء راکه با یک مستطیل مشخص شده‌اند، نمایش می‌دهد.

فلش‌ها از خط عمر شروع گردیده و به معنای آن هستند که عاملی در حال ارسال پیغامی به عامل دیگری در سیستم است. با پایین آمدن از روی خط عمر می‌توانید پیغامهای متوالی که در طول زمان بین عاملها رد و بدل می‌گردند را مشاهده کنید. زمان از بالا به پایین طی می‌شود.

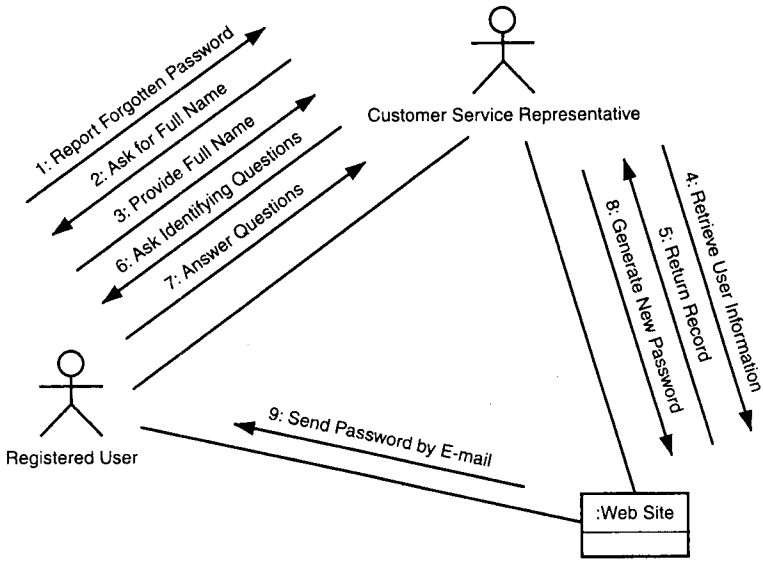
نمودارهای همکاری

از نمودارهای متوالی زمانی که بخواهید رخدادهای مختلفی راکه به صورت متوالی در طول زمان رخ می‌دهد، دنبال کنید، استفاده شود. حال اگر بخواهید رابطه میان عاملها و سیستم را مدل کنید باید نسبت به ایجاد نمودار همکاری اقدام کنید.

شکل ۹ـ۱۰ مورد کاربردی فراموشی رمز عبور را در نمودار همکاری نشان می‌دهد.

در نمودارهای همکاری تعامل بین عاملها از طریق اتصال خط نمایش داده می‌شود. بالای هر خط نام هر رخدادی که بین عاملها رخ می‌دهد، نوشته می‌شود. همچنین با شماره گذاری رخدادها می‌توان ترتیب آنها را نیز دانست.

شکل ۹ ـ ۱۰
نمودار همکاری برای
فراموشی رمز عبور



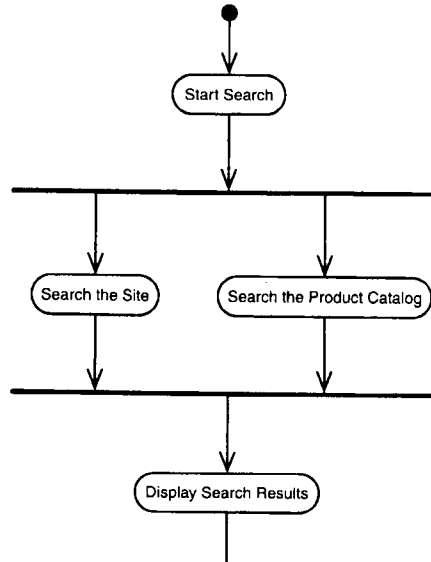
نکته

از نمودار متوالی برای مدلسازی سلسله‌ای از رخدادها در یک سناریو در طول زمان استفاده کنید. از نمودار همکاری برای مدلسازی روابط بین عاملها در یک سناریو استفاده کنید.

نمودارهای فعالیت

نمودارهای تعاملی رفتارهای ترتیبی را مدل می‌کنند. با این ترتیب نمی‌توانند رفتارهایی که به صورت موازی صورت می‌گیرند را مدل کنند. نمودارهای فعالیت کمک می‌کنند که اینگونه رفتارها به صورت موازی با یکدیگر مدل شوند.

شکل ۹ ـ ۱۱
نمودار فعالیت جستجو



برای مثال مورد کاربردی دیگر نظیر جستجو را در نظر بگیرید. از این طریق می‌توان در سایت و یا در فهرست کالاها جستجو کرد. هیچ دلیلی وجود ندارد که این دو نوع جستجو در یک زمان کار نکنند. اگر قرار باشد این دو نوع جستجو با یکدیگر کار نکنند تأثیر بدی بر روی کاربر می‌گذارد چرا که کاربر باید صبر کند تا یکی به پایان برسد تا دیگری انجام شود.

شکل ۹ ـ ۱۱ اینگونه فرایندها و اعمال را از طریق نمودار فعالیت مدل می‌کند.

هر بیضی وضعیت هر فرایند را نشان می‌دهد. خط کلفت و سیاه نقطه‌ای را نشان می‌دهد که فرایندها باید قبل از آنکه اجرای عملیات ادامه یابد، همزمان (Synchronize) شوند . همانگونه که مشاهده می‌کنید دو جستجو به صورت موازی و با هم اجرا می‌شوند و قبل از آنکه نتایج نمایش داده شوند با یکدیگر برخورد می‌کنند.

در روزهای آتی نگاه دقیقتری به نمودارهای تعاملی و فعالیت خواهیم داشت. با این حال این دو نمودار در تحلیل یک سیستم بسیار مفید هستند.

ساخت مدل دامنه

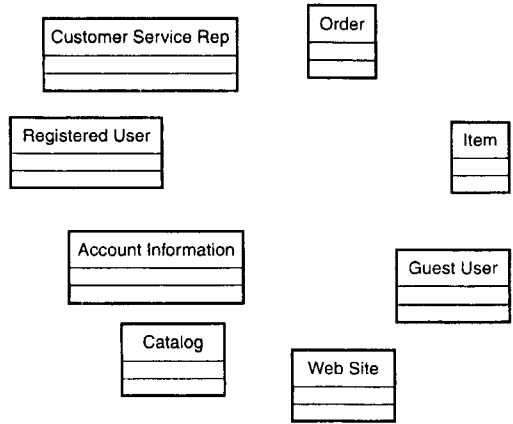
از طریق تحلیل موارد کاربرد، تعاملات سیستم شناسایی می‌شوند. درواقع از این طریق حرف حساب سیستم مشخص می‌شود! می‌توان از این راه دامنه مسأله را فهمید و اشیاء اصلی در سیستم را شناخت.

مدل دامنه آن اشیایی که برای مدلسازی صحیح سیستم به آنها نیاز دارید، را لیست می‌کند. فروشگاه اینترنتی را به خاطر بیاورید. از طریق موارد کاربرد تعدادی از اشیاء شناسایی می‌گردند. شکل ۹ ـ ۱۲ تعدادی از آنها را نشان می‌دهد.

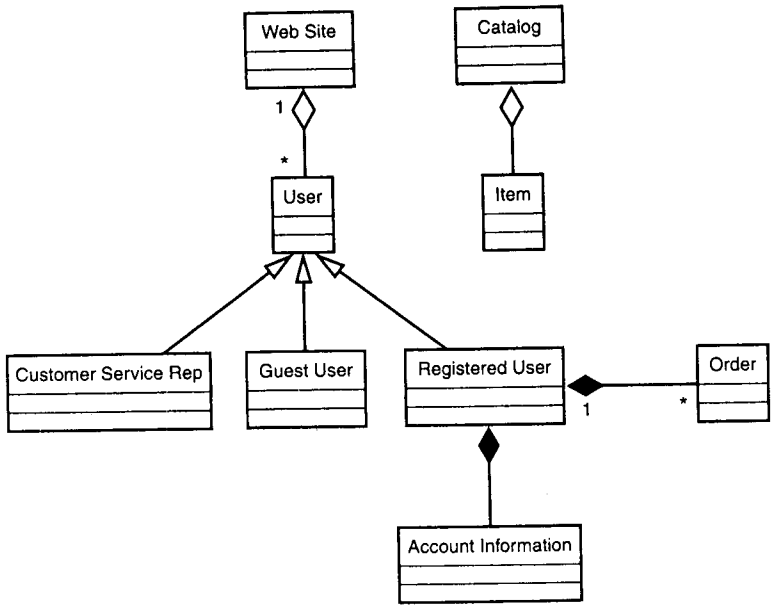
در این حالت می‌توانید روابط بین اشیاء دامنه را مدلسازی کنید. شکل ۹ ـ ۱۳ این روابط را نشان می‌دهد. مدل دامنه از برخی لحاظ مهم است. اول آنکه مدل دامنه مسأله را مستقل از هر نوع پیاده‌سازی مدل می‌کند. درواقع مدل دامنه مسأله را در سطح مفهومی مدل می‌کند. این استقلال این امکان را در اختیار می‌گذارد که بسیاری از مسایل مشکل داخل دامنه را به صورتی پایدار حل کنید.

دوم آنکه مدل دامنه پایه و اساس مدل اشیایی را که سیستم را ایجاد می‌کنند، می‌سازد. پیاده‌سازی نهایی ممکن است کلاسهایی را اضافه و یا کم کند. با این حال مدل دامنه نقطه شروع جهت طراحی و ساخت از یک پایه و اسکلت را در اختیار می‌گذارد. در آخر آنکه مدل دامنه‌ای که به صورت مناسبی تعریف شده باشد واژگان مشترکی از مسأله را برایتان فراهم می‌کند.

شکل ۹ ـ ۱۲
اشیاء دامنه



شکل ۹ ـ ۱۳
روابط اشیاء دامنه



حال چه کنیم؟

خوب، موارد کاربرد را ایجاد کردیم، نمودارهای تعاملی را ساختیم و مدل دامنه را شروع کردیم. قدم بعدی چیست؟

موارد کاربرد سه استفاده عمده دارند: اولین کاربرد آن است که اعلام می‌کند سیستم چه کاری انجام می‌دهد؟ چه کسانی با سیستم کار می‌کنند؟ تحلیل موارد کاربرد به شما کمک می‌کند تا سیستمی که می‌خواهید بسازید، را به خوبی درک کنید.

دوم آنکه موارد کاربرد فهرستی از وظایفی که باید انجام شود تا سیستم ایجاد شود را فراهم می‌کنند. می‌توان با طبقه‌بندی موارد کاربرد از نظر اهمیت و مشخص کردن میزان زمانی که لازم است تا آن را انجام داد (به صورت تقریبی) زمان کل برای ایجاد سیستم را به دست آورد. از این طریق می‌توان زمانبندی پروژه را کنترل کرد.

در آخر موارد کاربرد کمک خوبی برای ایجاد مدل دامنه هستند. مدل دامنه می‌تواند به عنوان اسکلتی برای سیستم محسوب شود. (و اگر آن را به طور صحیحی انجام داده باشید، می‌توانید از آن مدل در هر جای دیگر استفاده مجدد کنید!)

خلاصه

تحلیل شیءگرا (OOA) کمک می‌کند نیازمندیهای سیستمی که قرار است آن را ایجاد کنید شناسایی کنید. موارد کاربرد کمک می‌کنند تا نحوه تعامل کاربران با سیستم مشخص شود. موارد کاربرد تعاملات را توصیف می‌کنند و اینکه کاربران از سیستم چه انتظاری دارند. مدلها نظیر نمودارهای تعاملی و فعالیت این تعاملات را به صورتی نمادین و بصری تصویر می‌کنند. هریک از مدلها از یک نقطه متفاوت به سیستم نگاه می‌کنند. بنابراین هریک از این تفاوتها را باید به خاطر سپرده و هریک از آنها را در جای مناسب به کار ببندید.

از طریق موارد کاربرد می‌توان مدل دامنه را تعریف کرد. مدل دامنه را می‌توان اسکلت کار فرض کرد. مزیت مدل دامنه آن است که از هرگونه پیاده‌سازی مستقل است. به عنوان یک نتیجه می‌توان گفت مدل دامنه را در بسیاری از موارد می‌توان به کار گرفت. این واقعیت بسیار مهم است که OOA یک روش شیءگرای واقعی جهت حل مسأله است. موارد کاربرد چیزی جز اشیاء نیستند. تفاوتهای میان موارد کاربرد چیزی جز تفاوت موجود میان نمونه‌های یک کلاس نیستند. عاملها نیز خود به نوعی شیء هستند. OOA مسایل را بر اساس تعدادی مورد کاربردی و اشیاء دامنه می‌سازد!

پرسشها و پاسخها

اگر مورد کاربردی فراموش شود چه اتفاقی می‌افتد؟

اگر در حین کار متوجه شدید که مورد کاربردی از قلم افتاده است به عقب بازگشته و آن را اضافه کنید.

به هنگام کار با مورد کاربردی خاص آیا همیشه نیاز به ترسیم نمودارهای متوالی، همکاری و فعالیت دارید؟

خیر. همیشه به هر سه نیاز نیست. تنها چیزهایی را که نیاز دارید، انجام بدهید تا فهم درستی از آن برایتان ایجاد کند. با این حال عموماً به حداقل یکی از آنها احتیاج خواهید داشت.

از کجا می‌توان فهمید که موارد کاربرد را به طور کامل ایجاد کرده‌اید؟

در حقیقت نمی‌توان در وهله اول فهمید که آیا همه موارد کاربردی ایجاد شده‌اند، یا خیر. این امر تنها زمانی رخ می‌دهد که به صورت عملی با آنچه طراحی کرده‌اید کار کرده و تجربیاتی را به دست آورده باشید.

اگر مورد کاربردی را جا انداخته باشید کافی است به عقب بازگشته و آن را به مجموعه تحلیلهای خود اضافه کنید. با این حال باید به شدت مراقب باشید تا از اضافه کاریهای بیهوده پرهیز کنید.

کارگاه

پرسشها و تمرین‌های زیر، تنها برای افزایش درک شما ارایه می‌گردند.

پرسشها

۱. یک فرایند نرم‌افزاری چیست؟
۲. یک فرایند تکراری چیست؟
۳. در انتهای OOA، چه چیزی باید نصیب ما شود؟
٤. نیازمندیهای سیستم چه چیزی را به شما انتقال می‌دهند؟
۵. مورد کاربردی (Use Case) چیست؟
۶. چه مراحلی برای تعریف یک مورد کاربردی باید طی شود؟
۷. یک عامل چیست؟
۸. موارد کاربردی چگونه با یکدیگر در ارتباطند؟
۹. چه سؤالاتی را مطرح کنیم تا عاملهای سیستم مشخص شوند؟
۱۰. تنوع مورد کاربردی در کجاست؟
۱۱. یک سناریو چیست؟

۱۲. چه راههایی برای مدلسازی موارد کاربردی وجود دارد؟

۱۳. تفاوت میان مدلهای مختلفی که برای ترسیم موارد کاربردی مورد استفاده قرار می‌گیرند چیست؟

۱٤. مزایای مدل دامنه چیست؟

۱۵. مزایای مورد کاربردی در چیست؟

تمرین‌ها

۱. چه موارد کاربردی دیگر باید به موارد کاربردی ارایه شده در فروشگاه اینترنتی اضافه کنیم؟

۲. یکی از موارد کاربردی مطرح شده در تمرین ۱ را در نظر گرفته و آن را توسعه دهید.

۳. چه تفاوتهایی میان کاربران عضو (ثبت شده) و کاربران مهمان می‌توانید نام ببرید؟

٤. آیا اشیاء دامنه دیگری را می‌توانید پیدا کنید؟

روز ۱۰

Object Orinted Programming

آشنایی با روش طراحی شیءگرا (OOD)

در درس دیروز دیدیم که چگونه روش تحلیل شیءگرا (OOD) به فهم مسأله و نیازمندیهای آن کمک می‌کند. با تحلیل موارد واقعی و ایجاد مدل می‌توان جزییات کامل شرح مسأله را به دست آورد. با این حال، روش تحلیل شیءگرا یا OOA تنها بخشیء از فرایند توسعه نرم‌افزار شیءگرا است.

با کمک طراحی شیءگرا یا (Object Oriented Design) و نتایج تحلیل شیءگرا می‌توان طرح راه حل مسأله را به دست آورد. همچنانکه روش تحلیل شیءگرا کلیت راه حل را به دست می‌دهد، طراحی شیءگرا به شناسایی و طراحی اشیایی که در حل مسأله خاص وارد می‌شوند کمک می‌کند.

مباحث درس امروز به ترتیب عبارتند از:

- گذار از تحلیل به طراحی
- چگونه اشیایی که در طراحی مورد نیاز هستند را تعریف و طراحی کنیم
- چگونه کارت‌های CRC در درک روابط و مسئولیتهای اشیاء مفید واقع می‌شوند.
- کاربرد UML در نمایش طراحی

روش طراحی شیءگرا

واژه جدید

OOD عبارت است از فرایند ایجاد مدل اشیاء یک راه حل. به عبارت دیگر OOD فرایند شکستن راه حل به اشیاء تشکیل دهنده کلیت آن است.

واژه جدید

مدل شیء (Object Model) طرحی از شیء است که در حل مسأله ظاهر می‌شود. مدل شیء نهایی ممکن است شامل اشیایی باشد که در دامنه موجود نیستند. مدل شیء مسئولیتها، روابط و ساختارهای گوناگون را تشریح می‌کند.

فرایند OOD به پیاده‌سازی تحلیل‌هایی که در طی OOA صورت داده‌اید کمک می‌کند. عمدتاً، مدل شیء حاوی کلاسهای اصلی موجود در طرح، مسئولیتهای آنها و تعریف چگونگی رفتار متقابل آنها و استخراج اطلاعات آنها خواهد بود.

به OOD مانند فرایند ساخت یک خانه شخصی بیاندیشید. قبل از شروع به بنای یک ساختمان، باید در مورد مشخصات اتاقهایی که می‌خواهید در خانه داشته باشید تصمیم بگیرید. در این تحلیل‌ها شاید به این نتیجه برسید که در خانه احتیاج به یک آشپزخانه، دو اتاق خواب و یک و نیم حمام (!)، یک اتاق نشیمن و یک اتاق غذاخوری دارید. همچنین یک گاراژ دو ماشینه و یک جکوزی می‌توانند خانه رؤیایی شما را تکمیل کنند.

خوب بعد چه؟ بنا خبر می‌کنید؟ خیر! ابتدا یک معمار باید نقشه خانه را تهیه کند. طرح سیم‌کشیها و نورپردازیهای خانه باید تهیه شود. بر اساس این طرح‌ها و نقشه‌ها است که معمار کار بناها را هدایت می‌کند تا خانه ساخته شود.

با این پیش‌زمینه، می‌توانیم OOD را به آن نقشه نهایی (Blueprint) تشبیه کنیم.

چنین فرایند طراحی کاملی برای تعیین دقیق اشیایی که باید در برنامه ظاهر شوند و شیوهٔ برخورد متقابل آنها بسیار مفید است. طراحی ساختار اشیاء را مشخص می‌کند و طرح فرایند ایجاد، نکات بسیاری در مورد کدنویسی برنامه را مشخص خواهد کرد.

وقتی کار تیمی است، شناخت و حل مسایل و معضلات طراحی قبل از شروع به کار اهمیت فراوانی دارد. با حل پیشگیرانهٔ مسایل، تمام اعضای تیم می‌توانند با فرضهای یکسانی کار را شروع کنند. راهبرد یکسان برای تمام اعضای تیم، یکی کردن کدهای توسعه دهندگان تیم را بعد از پایان کار آسانتر می‌کند. بدون یک طرح خوب و گویا هر برنامه‌نویسی فرضهای خود را بنا خواهد نهاد. در اکثر موارد این فرضها با هم متفاوت و متناقض خواهند بود. کار دو برابر خواهد شد و تعادل مسئولیتها بین اشیاء بر هم خواهد خورد. در نتیجه طرح نمی‌تواند درست پیاده‌سازی شود و از تمام این کارها نتیجه‌ای به دست نخواهد آمد.

به علاوه حل مشکلات و رفع اشتباهات به صورت نمایی گرانتر تمام خواهند شد. زیرا دیرتر کشف خواهند شد. در حالی که خطای طراحی به سادگی با کمترین هزینه در زمان طراحی قابل رفع است.

در زمان طراحی درخواهید یافت که عمدتاً یک مسأله خاص بیش از یک راه حل دارد. OOD اجازه می‌دهد، هر راه حل دنبال شود و راه حل مناسبتر برای آن مسأله انتخاب شود. به همین صورت می‌توان تصمیم‌های آگاهانه‌ای گرفت و دلیل آنها را در مستندات پروژه ثبت کرد.

مدل شیء، اشیاء دارای اهمیت در طراحی را مشخص می‌کند. یعنی مدل شیء فرا مجموعه‌ای (Superset) از دامنه است. در حالی که بسیاری از اشیاء موجود در مدل دامنه، در طراحی مورد استفاده قرار خواهند گرفت، بسیاری دیگر هم که در دامنه موجود نیستند هم ممکن است وارد طرح شوند. دقیقاً به همان

صورتی که نقشهٔ سیم‌کشی در طرح اولیه خانه وارد نمی‌شود. وقتی طرح آماده شد، می‌توان شروع به کدنویسی کرد.

در کار طراحی از افراط خودداری کنید. دقیقاً به همان صورتی که در OOA خطر گرفتار شدن در دام اختلال تحلیل وجود دارد، در OOD هم امکان اختلال طراحی موجود است.

از افراط در جزییات حل خودداری کنید. پیش‌بینی کردن تمام تصمیم‌ها و مسایل ممکن در روند طراحی غیرممکن است. به علاوه تصمیم‌گیری در مورد برخی نکات می‌تواند به زمان کدنویسی موکول شود. به هر حال مهم شروع کردن به کدنویسی است. آن را بیش از حد به تأخیر نیاندازید.

چگونه می‌توان نکات یا جنبه‌های اساسی طراحی را شناسایی کرد؟ جنبه‌های اساسی وقتی مورد بحث هستند که یک تصمیم‌گیری بتواند در ساختار یا رفتار سیستم تغییری اساسی ایجاد کند.

دوباره مسأله خرید روی خط و کارت اعتباری مطرح شده در درس دیروز را به یاد بیاورید. در بررسی تحلیل با دانستن اینکه در برنامه نیاز به کارت خرید یا کارت اعتباری داریم، باید به طراحی تجریدهایی که به آنها نیاز داریم بپردازیم. در این صورت نیاز به یک شیء یا متغیر نگهدارنده قیمت‌ها، یک مراقب انقضای مدت اعتبار و چند مورد دیگر داریم. اما در این نقطه نیازی به طراحی یا تصمیم‌گیری در مورد ساختار داده مطلوب برای هریک نخواهیم داشت. این تصمیم مربوط به فاز پیاده‌سازی است، لذا در این فاز پرداختن به آن افراط در جزییات است.

چگونه OOD را اعمال کنیم؟

OOD فرایندی پویا برای تعیین و تبیین اشیاء و مسئولیتهای آنها و چگونگی ارتباط آنها با یکدیگر است. در فرایند طراحی مرتباً ضمن تعامل دائم با مدل اشیا به بهبود دادن و کامل کردن آن می‌پردازیم. هر تعاملی باید به طراح دیدی عمیق‌تر در طراحی و شاید هم در دامنه بدهد.

نکته

همچنانکه در ضمن تلاش برای حل مسأله بیشتر در مورد آن می‌آموزید، شاید لازم باشد حتی دوباره به تحلیل بپردازید. به یاد داشته باشید، هرگز از بازگشت و تکمیل تحلیل‌هایتان شرم نداشته باشید. تنها چیز شرم‌آور تولید نرم‌افزار بدون خاصیت است.

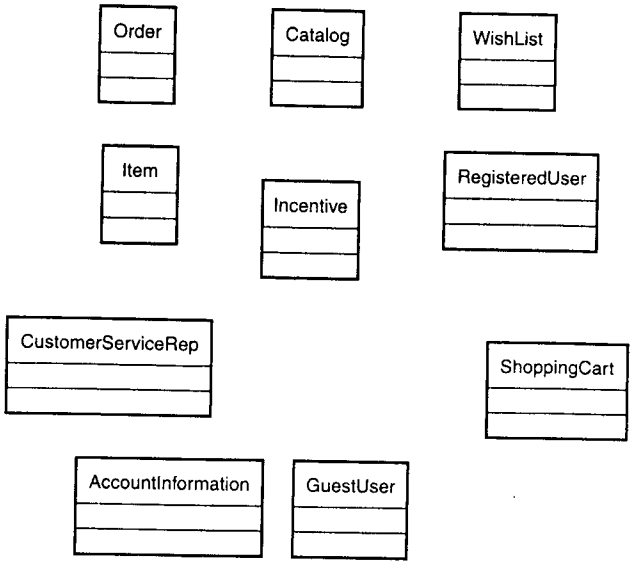
جهت ایجاد مدل شیء تقریباً به ترتیب زیر عمل می‌کنیم:

۱. فهرستی اولیه از اشیاء تهیه می‌کنیم.
۲. مسئولیتهای اشیاء را بازنگری می‌کنیم.
۳. نقاط تعامل اشیاء را بررسی می‌کنیم.
٤. جزییات روابط بین اشیاء را استخراج می‌کنیم.
۵. مدل را می‌سازیم.

درک برنامه‌نویس از طرح در ضمن تکرار این چند مرحله افزایش می‌یابد.

نکته

روش دقیق و مناسب هر مسأله به تجربه شما از مسأله، دامنه کاربرد، خواست مشتری و ذائقهٔ شما بستگی دارد. پس از پایان فرایند OOD حل مسأله باید به چند شیء تجزیه شده باشد. اینکه چگونه به این اشیاء می‌رسید بستگی به خود شما و تیم طراح همکارتان دارد.



شکل ۱۰ ـ ۱
کلاسهای هسته فروشگاه اینترنتی

گام اول: تهیه فهرست اولیه اشیاء

با دامنه‌ای که در تحلیل تعریف کرده‌اید آغاز کنید. تمام اشیاء و بازیگران درون دامنه باید در مدل به کلاس تبدیل شوند. خواهید دید برخی اشیاء دامنه به مدل شیء نهایی راه نخواهد یافت. با این حال در این مرحله که نمی‌توان تشخیص داد کدام یک به مدل وارد خواهد شد و کدام یک نه، باید همه را لحاظ کرد.

دوباره مسأله کارت اعتباری را در نظر بگیرید. شکل ۱۰ ـ ۱ کلاسهای هسته‌ای را که در مدل اولیه اشیاء ظاهر خواهند شد را نشان می‌دهد. در چنین مدلی باید رویدادهای ممکن را هم لحاظ کرد. هرکدام از این رویدادها در ابتدا باید به صورت کلاس ظاهر شوند. همین راهکار باید در مورد گزارشها، نمایش‌ها و ابزارها اعمال شود. تمام این عناصر باید به صورت کلاس مدل شوند.

تذکر

چند نکته را به یاد داشته باشید:

- هر عامل، شیء دامنه و رویدادی را به کلاس تبدیل کنید.
- شیوه نمایش اطلاعات توسط سیستم را به یک شیء تبدیل کنید.
- هر سیستم ثالث یا ابزاری را که سیستم ممکن است با آن تماس داشته باشد را به کلاس مبدل کنید.

در این زمان، دربارهٔ کلاسها چیز زیادی نمی‌توان گفت. شاید در مورد مسئولیتها و روابط اشیاء ایده‌ای داشته باشید، با این حال قبل از نهایی کردن شناختتان از کلاس باید کمی بیشتر به بررسی بپردازید.

گام دوم: بازنگری مسئولیتهای اشیاء

تهیه فهرست اشیاء شروع خوبی است، اما فقط شروع کار است. طرح کامل علاوه بر ساختار و روابط اشیاء به مسئولیتهای آنها هم می‌پردازد. همچنین شیوه اتصال اجزاء را هم نشان خواهد داد.

برای رسیدن به چنین درکی باید بدانیم هر شیء چه کاری انجام می‌دهد. برای پاسخ گفتن به این پرسش باید به دو مورد توجه کرد.

ابتدا باید مسئولیتها را بررسی کرد. از درس کپسوله‌سازی به یاد داریم که هر شیء باید تعداد محدودی مسئولیت داشته باشد. در روند طراحی باید مسئولیت هر شیء معلوم شود و در صورتی که تعداد مسئولیتهای شیء زیاد باشد، شیء به چند شیء ساده‌تر شکسته شود. توجه کنید که در این کار باید به دقت از ایجاد مسئولیتهای مضاعف خودداری کنید. یعنی هر وظیفه تنها یکبار ظاهر شود.

در وهلهٔ دوم باید به این موضوع پرداخت که هر شیء چگونه کارش را انجام می‌دهد. گاهی اشیاء وظایف را به یکدیگر واگذار (Delegate) می‌کنند. چنین همکاریهایی (Collaboration) باید در طراحی در نظر گرفته شوند.

واژه جدید شیء وظیفه را به همکار (Collaborator) واگذار می‌کند.

واژه جدید همکاری رابطه‌ای است که طی آن اشیاء برای رسیدن به هدفی با هم به تعامل می‌پردازند.

فهم دقیق روابط و وظایف شیء اهمیت بسیاری دارد. در سطح عملی هر وظیفه به یک روال ترجمه خواهد شد. روابط هم به ساختار تفسیر می‌شوند. در طراحی همواره سعی کنید که تا حد ممکن وظایف را پخش کنید.

کارت CRC چیست؟

یک راه برای تصویر وظایف و همکاریها استفاده از کارتهای Class Responsiblities Collabarations یا CRC است. یک کارت CRC چیزی جز یک کارت اندیس ۴×۶ نیست.

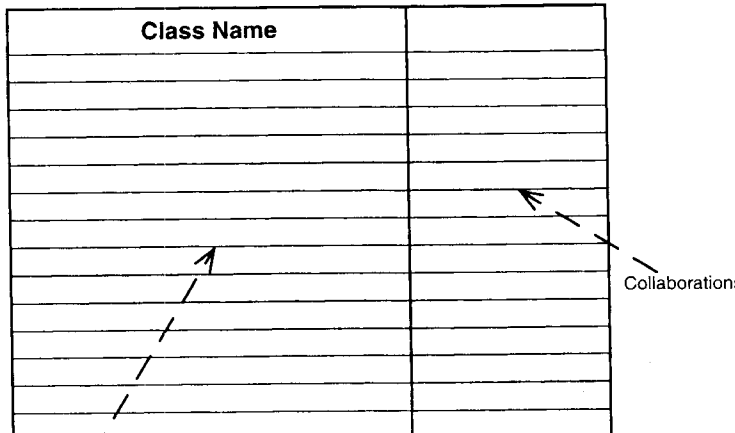
وقتی برای بار اول به طراحی می‌پردازید، فهرست کردن روالها و صفات کار مشکلی است. لذا می‌توان با معرفی هدف هر شیء شروع کرد. استفاده از کارتهای CRC جلب نظر شما به وظایف را ساده‌تر می‌کند.

به سادگی برای هر کلاس یک کارت در نظر می‌گیریم. نام کلاس را در بالای کارت می‌نویسیم. سپس کارت را با خطی به دو قسمت تقسیم می‌کنیم. وظایف را در سمت چپ و نام هر شیء دیگری که کلاس باید با آن تعامل داشته باشد را در سمت راست می‌نویسیم.

شکل ۱۰ - ۲ الگوی یک کارت CRC را نشان می‌دهد.

اگر دیدید که کارت برای لیست شما کوچک است، می‌توان نتیجه گرفت که باید کلاس را تقسیم کرد. مزیت این کارتها امکان استفاده از آنها در همه جا است و این یعنی در هر جایی می‌توانید به طراحی بپردازید.

شکل ۱۰ ـ ۲
الگوی یک کارت CRC



علاوه بر این قیمت ایجاد تغییر روی کارتها بسیار کمتر از ایجاد تغییر در مدل کامپیوتری است. چون سرعت ایجاد تغییر در اشیاء روی کارتها بسیار بالاتر است!

و در پایان کارتهای CRC مزیت مهم دیگری هم دارند. برای ایجاد طرح‌های مختلف، فقط کافیست روش چیدن کارتها را عوض کنید. با این کار حتی می‌توانید طراحی را با کمک یک تیم انجام دهید.

چگونه از کارتهای CRC استفاده کنیم؟

بهتر است از کارتهای CRC ضمن همراهی با سایر توسعه دهندگان و اعضای تیم استفاده کنید. در این صورت تعامل شما با هم بیشتر خواهد بود و روند طراحی جالبتر خواهد شد.

کار را با در نظر گرفتن حالتهای مختلف آغاز می‌کنیم. تعداد حالتها بستگی به زمان در دسترس میزان و مشکل بودن حالتها دارد.

جلسه را شروع کنید و کارتها را بین همکارانتان پخش کنید. در جلسه می‌توانید تمام حالتها را در نظر بگیرید. هر طراح باید روی مسئولیتها و همکاریهای کلاس خود تمرکز کند. وقتی به کلاسی نیاز افتاد، طراح کاربرد کلاس و همکاریهای آن را برمی‌شمرد.

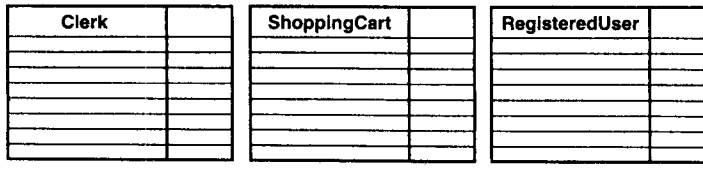
در طی چنین جلساتی، می‌توانید طرح‌های مختلف را بررسی کنید، اشیاء تازه کشف کنید و یا حتی در تیم نرم‌افزاری هماهنگی ایجاد کنید.

مثالی از کارتهای CRC

مثال سفارش(Order) از درس روز نهم را در نظر بگیرید.

- سفارش
  ۱. کاربر ثبت شده به قسمت صندوق مراجعه می‌کند.
  ۲. کاربر ثبت شده اطلاعات خرید را ارایه می‌دهد.
  ۳. سیستم جمع کل سفارشها را نشان می‌دهد.
  ۴. کاربر اطلاعات مربوط به پرداخت را ارایه می‌کند.
  ۵. سیستم اطلاعات فوق را چک می‌کند.
  ۶. سیستم سفارش را تایید می‌کند.
  ۷. سیستم تاییدیه سفارش را از طریق پست الکترونیک ارسال می‌کند.
- شرایط قبلی
  کارت خرید غیر تهی
- شرایط بعدی
  وجود سفارش در سیستم
- حالت دیگر: لغو سفارش
  طی مراحل ۱ تا ۴، کاربر می‌تواند سفارش را لغو کند. در این صورت به صفحه اول برمی گردد.
- حالت دیگر: عدم تصدیق هویت

در مرحله ۵، سیستم هویت و اطلاعات مربوط به پرداخت را تصدیق نمی‌کند. کاربر می‌تواند این اطلاعات را دوباره وارد کند و یا آنکه سفارش را لغو کند.



شکل ۱۰ ـ ۳
چند کارت CRC

با تعریف کلاسها شروع می‌کنیم. می‌توان دید که به کلاسهای زیر نیاز داریم:
ShoppingCart، OrderConfirmation، Payment، RegisteredUser، Order و ShippingInformation. می‌توان System را هم به لیست افزود. سیستم را به صورت یک جعبه سیاه در نظر می‌گیریم و در طی تحلیل آن را به بخش‌های کوچکتر تجزیه خواهیم کرد.

می‌توان مسئولیتهای System را روی یک کارت نوشت و بعداً آن را تجزیه کرد. اگر چیزی کم بود، می‌توان آن را بعداً افزود.

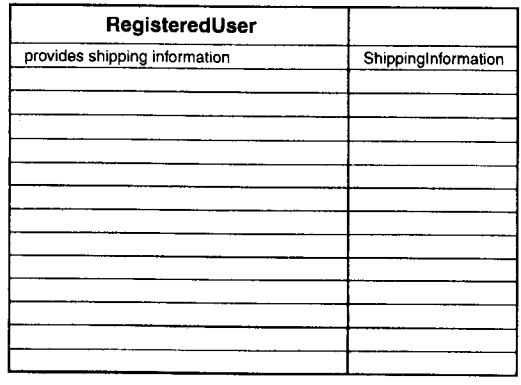
نگاهی به گامهای سناریو می‌اندازیم. سیستم کاربر را شناسایی می‌کند، سفارش را نمایش می‌دهد و سفارش را تأیید می‌کند. پس می‌توانیم فعلاً سیستم را به سه بخش OrderDisplay، PaymentTerminal و Clerk تجزیه کنیم. حال می‌توانیم کارتها را وارد بازی کنیم. شکل ۱۰ ـ ۳ را ببینید.

قدم بعدی بررسی هر گام سناریو و پیدا کردن وظایف است. گام اول تنها یک پیوند قابل کلیک کردن در رابط کاربری است. برای ساده شدن کار رابط کاربر (UI) را نادیده خواهیم گرفت. گام دوم، جالب‌تر است. در این گام RegisteredUser وظیفه دارد اطلاعات خرید را برای Clerk تأمین کند. شکل ۱۰ ـ ۴ تغییر کارتها را نشان می‌دهد.

نکته

هر کجا مشاهده کردید که شیئی اطلاعات را تأمین می‌کند، ابتدا باید مطمئن شوید که شیء ساختمان داده نیست. زیرا در این صورت بهتر است آن را به عنوان یک متغیر با شیئی که به اطلاعات نیاز دارد، ترکیب کنیم.

گام سوم، شلوغ‌تر است! قبل از اینکه سیستم بتواند چیزی نمایش دهد Clerk باید سفارش را وارد کند، قیمت را تخمین بزند و قیمت‌ها را جمع بزند. Clerk در این راه از ShoppingCart و OrderDisplay استفاده خواهد کرد.



شکل ۱۰ ـ ۴
یک کارت CRC برای RegisteredUser

شکل ۱۰ـ۵
کارت CRC برای Clerk

Clerk	
retrieve shipping and payment information	ShippingInformation
enter the order	ShippingCart Order
display the order	OrderDisplay

شاید بهتر باشد خود Clerk مسئول قیمت‌گذاری یا جمع زدن نباشد. به یاد داشته باشید هر شیء باید مسئولیتهای محدودی داشته باشد.

شیء Order باید مسئولیت قیمت‌گذاری و جمع زدن قیمتها را بر عهده بگیرد. شکل ۱۰ – ۵ مسئولیتها یا وظایف Clerk تا اینجا را خلاصه می‌کند. شیء Order، همچنین باید حاوی خریدها هم باشد. شیء Item در اینجا وارد طرح می‌شود.

گامهای باقیمانده شبیه قبلی‌ها هستند. RegisterdUser اطلاعات Payment را فراهم می‌کند، Clerk خرید را بررسی کرده و سفارش را نهایی می‌کند. شکل‌های ۱۰ – ۶ و ۱۰ – ۷ مسئولیتهای Clerk و RegisteredUser را خلاصه می‌کنند.

هر کلاس چند وظیفه می‌تواند بر عهده داشته باشد؟

دو یا سه مسئولیت برای هر کلاس کافی است. در صورتی که مسئولیتهای بیشتری موجود باشند، بهتر است کلاس به دو یا بیشتر بخش تفکیک شود.

کلاس Clerk را در نظر بگیرید. وظایف آن عبارتند از:

- تهیه اطلاعات خرید و پرداخت
- وارد کردن سفارش

شکل ۱۰ـ۶
کارت CRC کامل Clerk

Clerk	
retrieve shipping and payment information	RegisteredUser
enter the order	ShippingCart Order
display the order	OrderDisplay
authorize the order	PaymentTerminal
confirm the order	Order

RegisteredUser	
provides shipping information	ShippingInformation
provides payment information	Payment

شکل ۱۰ـ۷
کارت CRC برای RegisterdUser

- بررسی اعتبار پرداخت
- قبول سفارش

چنانکه شکل ۱۰ – ۸نشان می‌دهد، تمام این مسئولیت‌ها را می‌توان تحت عنوان یک وظیفهٔ ProcessOrder جمع‌آوری کرد.

فهرست کردن وظایف جزیی و فرعی اهمیت دارد. با این حال نباید در آن زیاده‌روی کرد.

آنچه اهمیت دارد این است که تمام این وظایف در راستای رسیدن به یک هدف باشند. در اینجا کل اعمال فرایند سفارش را شکل می‌دهند.

به یاد داشته باشید که وظیفهٔ کارتهای CRC فقط تبیین وظایف و همکاریهای ساده اشیاء است. از آنها برای تشریح روابط پیچیده استفاده نکنید.

محدودیتهای کارتهای CRC

از آنجایی که کارتهای CRC در اصل برای آموزش طراحی شده‌اند، فقط به کار بررسی موارد ساده می‌آیند. یعنی وقتی طرح پیچیده می‌شود، استفاده از آنها مشکل خواهد شد. روابط پیچیده درون شیئی و بالا رفتن تعداد کلاسها هم کارتها را بلااستفاده می‌کند. همچنین کارتها برای مدلسازی بی‌فایده هستند.

شکل ۱۰ـ۸
ProcessOrder

Clerk		
retrieve shipping and payment information	RegisteredUser	Process Order
enter the order	ShippingCart Order	
display the order	OrderDisplay	
authorize the order	PaymentTerminal	
confirm the order	Order	

تذکر

کاربرد کارتهای CRC کجاست؟

- در مراحل اولیه طراحی
- وقتی در OOP تازه کار هستید.
- برای تجسم وظایف و روابط
- برای فهم بهتر سناریو
- در پروژه‌های کوچک یا برای تمرکز بر بخش کوچکی از یک پروژه بزرگ

گام سوم: ایجاد نقاط تعامل

نقطه تعامل وقتی پدید می‌آید که شیء از شیئی دیگر استفاده کند. در بررسی و ایجاد نقاط تعامل باید به چند مورد توجه کنید.

رابط‌ها

وقتی شیء باید با شیء دیگر تعامل داشته باشد، باید یک رابط با تعریف صحیح وجود داشته باشد. در این صورت می‌توان مطمئن بود که تغییر در پیاده‌سازی یک شیء، در شیء دیگر تأثیر ندارد.

عاملها

بهتر است به برخی تعامل‌ها دقیق‌تر نگاه کنیم. برای مثال یک کتابخانه را در نظر بگیرید. کتابخانه دارای قفسه‌های حاوی کتاب است. وقتی لازم می‌شود که کتابی از قفسه برداشته شود، آیا این کتاب است که باید دارای یک روال removeBook() باشد، یا قفسه؟ درواقع این وظیفه هیچیک از این دو نیست. در دنیای واقعی برداشتن کتاب برعهدهٔ کتابدار یا ناظر است. کتابدار یا ناظر در دنیای OOP هم ظاهر می‌شوند. در دنیای OOP چنین بازیگرانی عامل (Agent) یا واسطه (Intermediary) نامیده می‌شوند.

واژه جدید عامل موجودیتی است که بین دو یا چند شیء برای رسیدن به هدفی، واسطه می‌شود.

نیازهای آینده

در بررسی تعاملات شیء، شاید لازم باشد شرایطی که شیء ممکن است با آنها روبرو شود را هم جستجو کرد. برنامه‌ریزی برای آینده، همان طراحی نرم‌افزار آینده‌نگر است. در چنین شرایطی، بهتر است از روابط اتصال‌پذیر و چندشکلی بودن استفاده کرد. در این صورت می‌توان بدون نگرانی اشیاء جدید را وارد کار کرد.

در این مورد هم توجه داشته باشید که زیاده‌روی مضر خواهد بود. فقط برای مواردی برنامه‌ریزی کنید که مطمئن هستید در آینده تغییر به وجود خواهد آمد. دو راه عمومی برای تخمین نیازهای آینده وجود دارد:

- تغییر یک نیاز است. اگر نیازمندیها در آینده تغییر می‌کنند، برای آنها برنامه‌ریزی کنید.
- اگر از یک فایل کتابخانهٔ خارجی استفاده می‌کنید که احتمال به روزآوری شدن آن وجود دارد، برای تغییر برنامه‌ریزی کنید.

راه حل مورد اول استفاده از روابط جانشین‌پذیری از طریق وراثت است. مورد دوم کار بیشتری می‌برد. باید در کلاسهای خود کتابخانهٔ خارجی را پوشش دهید (wrap). از این طریق می‌توانید از اتصال‌پذیری و چندشکلی بودن استفاده کنید.

تبدیل داده‌ها

در طراحی شاید به مواردی برخورد کنید که لازم باشد داده را قبل از انتقال ترجمه کنید. معمولاً چنین انتقال داده‌ای به کلاس دیگری واگذار می‌شود. زیرا در صورت تغییر روش انتقال فقط کلاس انتقال دهنده باید دستکاری شود. علاوه بر این، چنین کاری تمرینی برای پخش کردن وظایف هم هست.

وقتی این نقاط تعامل را مورد بررسی قرار می‌دهید و کلاسهای جدید اضافه می‌کنید، شاید لازم شود که دوباره کارتهای CRC را از کشو درآورید!

گام چهارم: تعیین جزییات روابط بین کلاسها

وقتی وظایف و همکاریهای پایه‌ای را ایجاد کردید، باید جزییات روابط پیچیدهٔ بین کلاسها را استخراج کنید. در همین جاست که وابستگی‌ها، تشریک‌ها و تعمیم‌ها تعریف می‌شوند. تعیین جزییات این روابط از آن جهت اهمیت دارد که تبیین کننده چگونگی به هم پیوستن اشیاء به یکدیگر، همچنین ساختار داخلی اشیاء گوناگون است.

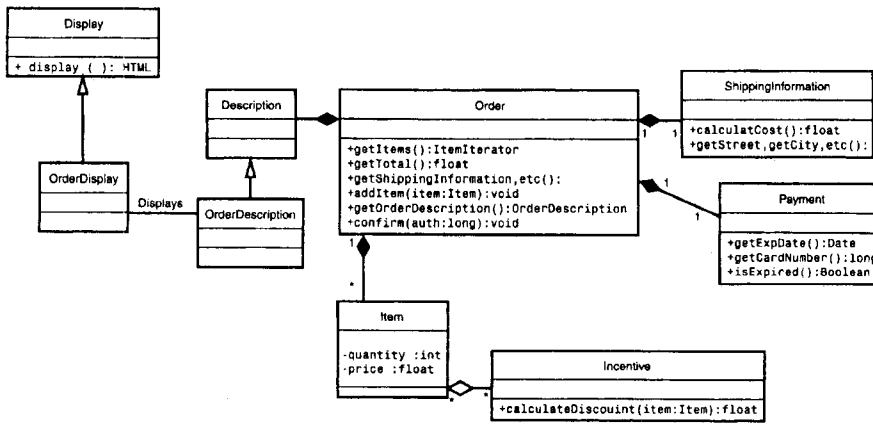
با کارتهای CRC شروع کنید. با اینکه این کارتها نمی‌توانند تمام روابط را نشان دهند، اما برای تشخیص واگذاری‌ها بسیار کارآمد هستند.

به دنبال کلاسهایی با وظایف یکسان بگردید. در صورتی که چنین چیزی یافتید، می‌توانید مورد مشترک را به یک کلاس پایه منتقل کنید. بهتر است گام سوم را هم دوباره انجام دهید تا تمام روابط جانشین‌پذیری لازم به دست آیند.

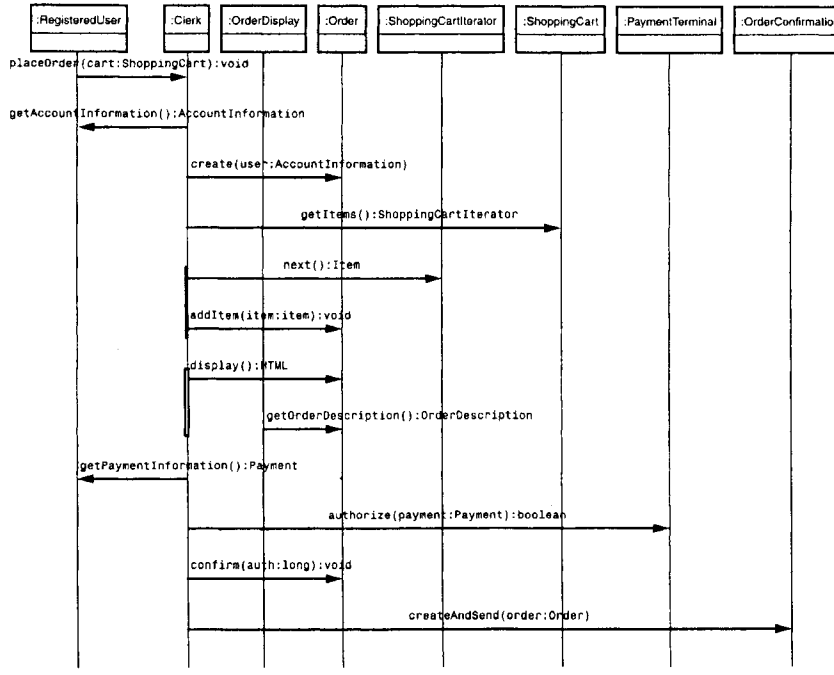
گام پنجم: ایجاد مدل

یک مدل کامل، از دیاگرامهای کلاس و تعاملهای آنها تشکیل می‌شود. این دیاگرامها ساختارها و روابط بین کلاسهای گوناگون موجود در سیستم را تشریح می‌کنند. علاوه بر این، UML مدل‌هایی برای فعالیتها و تعاملات هم فراهم کرده است.

شکل ۱۰ – ۹ ساختار سفارش را مدل می‌کند. شکل ۱۰ ـ ۱۰ هم نمودار ترتیبی حالتهای در نظر گرفته



شکل ۱۰ ـ ۹ مدل سفارش



شکل ۱۰ ـ ۱۰ دیاگرام ترتیبی سفارش

شده سفارش را نشان می‌دهد. علاوه بر این مدلها، می‌توان برای تعاملات مهم درون سیستم هم مدل تهیه کرد. وقتی ایجاد مدل به پایان رسید، باید به شرح تمام ساختارهای اصلی و روابط مهم درون سیستم بپردازید. در چنین مدلی می‌توان مشاهده کرد که چگونه اشیاء گوناگون ساخته می‌شوند، با هم مرتبط می‌شوند و چگونه برای مدل کردن راه حل به یکدیگر پیوند می‌خورند.

خلاصه

OOD گام بعد از OOA در فرایند ایجاد نرم‌افزار شیءگرا است. OOD دامنهٔ تهیه شده توسط OOA را تبدیل به راه حل برای مسأله می‌کند. در فرایند OOD از روی مدل دامنه، مدل اشیاء راه حل ساخته می‌شود. مدل اشیاء جنبه‌های مهم معماری سیستم را تشریح می‌کند. از جمله ساختار اشیاء و روابط بین آنها. در پایان فرایند OOD، باید ایده جامعی از آنچه در کد پیاده خواهید کرد داشته باشید.

سعی کنید در OOD این ۵ مرحله را تکرار کنید:

گام اول: تهیه فهرست اولیه اشیاء

گام دوم: تبیین و تصحیح وظایف شیء توسط کارتهای CRC

گام سوم: ایجاد نقاط تعامل

گام چهارم: تعیین جزییات روابط بین اشیاء

گام پنجم: ایجاد مدل

هر گام مدل را کاملتر کرده و ما را به نقشه نهایی سیستم نزدیکتر می‌کند.

پرسش‌ها و پاسخ‌ها

چرا باید پیش از شروع کدنویسی به طراحی سیستم پرداخت؟

طراحی به پیش‌بینی مشکلات اجرا و یافتن راه حل آنها کمک شایانی می‌کند. طراحی در هماهنگی اعضای تیم‌های نرم‌افزاری اهمیت حیاتی دارد. طراحی بسیاری از تصمیم‌گیریهای ساختاری را از زمان اجرا به زمان طراحی منتقل می‌کند، که برای جلوگیری از اختلاف در فرضهای طراحان مختلف بسیار مفید است.

مدلسازی قبل از کدنویسی چه مزیتی دارد؟

اهمیت مدلسازی، مانند اهمیت داشتن یادداشت برای سخنرانی است. مدلسازی طرح، چگونگی پیوند قطعات را به تصویر می‌کشد. با ایجاد مدل بصری می‌توان با دقت بیشتری پارامترهای طراحی را تنظیم کرد. گاهی وقتی به راه حل فکر می‌کنید ساده به نظر می‌رسد. نوشتن مسأله و راه حل آن مجبورتان می‌کند دربارهٔ آن با خود صادق باشید! در این صورت می‌توانید نقادانه حل خود را بررسی کنید.

چطور می‌فهمیم که طراحی پایان یافته است؟

قانون خاصی در این مورد وجود ندارد. تنها مراقب باشید در دام جزییات گرفتار نشوید. پایان طراحی در واقع بستگی به آنچه دارد که می‌خواهید از طریق مدل انتقال دهید. همچنین به تجربه خود شما و اعضای تیم نرم‌افزاری. در صورتی که تیم نرم‌افزاری دارای تجربه زیادی باشد، مدل کردن روابط سطح بالا به تنهایی کفایت می‌کند.

در هر صورت وقتی کار تمام می‌شود که مطمئن باشید می‌توانید مدل را به راحتی پیاده‌سازی کنید.

کارگاه

پرسشها

۱. **سه مزیت عمده طراحی استاندارد را بیان کنید.**
۲. **OOD چیست؟**
۳. **مدل شیء چیست؟**
٤. **جنبه‌های منفی زیاده‌روی در جزییات طراحی کدامند؟**
۵. **چگونه جنبه‌های دارای اهمیت طراحی را تشخیص می‌دهید؟**
۶. **پنج مرحله اساسی OOD کدامند؟**
۷. **لیست اولیه اشیاء را چگونه تهیه می‌کنید؟**
۸. **یک طرح کامل شامل چیست؟**
۹. **کارتهای CRC به چه کار می‌آیند؟**
۱۰. **همکاری چیست؟**
۱۱. **چرا درک عمیق از روابط و وظایف اشیاء مهم است؟**
۱۲. **کارتهای CRC چه هستند؟**
۱۳. **مشکل اساسی کارتهای CRC چیست؟**
۱٤. **نقطه تعامل چیست؟ مسایلی که باید در مورد آنها در نظر بگیرید کدامند؟**
۱۵. **عامل چیست؟**

تمرین‌ها

۱. برای تمرین ۲ از روز نهم با استفاده از کارتهای CRC وظایف را استخراج کنید. وظایف ShoppingCart کدامند؟

روز ۱۱

Object Oriented Programming

استفاده مجدد از طرحها از طریق الگوهای طراحی

در فصل پیش دریافتید که چگونه طراحی شیءگرا می‌تواند شما را در حل یک مسأله کمک کند. از طریق طراحی شیءگرا (OOD) می‌توانید طرحی کلی بسازید که اشیای تشکیل دهنده سیستم را به تصویر کشد. با این طراحی، می‌توان حل مسأله را توسعه بخشید.

با این توضیحات ممکن است پرسشهای زیر به ذهنتان خطور کند:

- چگونه می‌توان فهمید که طراحی صورت گرفته مناسب است؟
- آیا طراحی‌تان در آینده نتایج غیرمترقبه‌ای در پی نخواهد داشت؟
- آیا دیگر طراحان، این مسأله و یا مسایل نظیر آن را درگذشته حل کرده‌اند یا خیر؟
- در مورد استفاده مجدد، چطور؟ طراحی شیءگرا و به دنبال آن برنامه‌نویسی شیءگرا (OOP) قابلیت استفاده مجدد را در اختیار برنامه‌نویس قرار می‌دهد.

آیا می‌توان از این قابلیت در دیگر طراحیها نیز استفاده کرد؟

پاسخ این سؤالات را در این فصل خواهید گرفت.

آنچه امروز خواهید آموخت:

- چگونگی بکارگیری الگوهای طراحی

- چگونگی اعمال چهار الگوی مشترک
- چگونه یک طراحی می‌تواند از یک الگو بهره ببرد.
- چگونه می‌توان از طراحی الگوهای مشابه اجتناب کرد.

استفاده مجدد از طراحی

مهمترین هدف OOP استفاده مجدد از کد می‌باشد. زمانی که از کدی استفاده مجدد می‌کنید، اطمینان دارید نرم‌افزاری که می‌نویسید بر پایه محکمی بنا می‌شود. چرا که امتحان خود را پس داده است. در ضمن مطمئن هستید کدی که استفاده می‌کنید، مسأله را حل خواهد کرد. آنچه گفته شد برای کدهایی بود که قبلاً نوشته شده‌اند، اما کدهایی که نیاز به طراحی دارند، چطور؟ چگونه می‌توان فهمید که طراحی صورت گرفته خوب است؟ خوشبختانه الگوهای طراحی بسیاری از مشکلات، را به هنگام طراحی برطرف می‌کنند. با گذشت زمان بسیاری از برنامه‌نویسان متوجه شده‌اند که در طراحی، الگوهای مشترکی تکرار می‌شوند. از این رو این امر نکته قابل توجهی برای برنامه‌نویسان گشته است و نتیجه آن رشد پیوسته الگوهای طراحی است.

واژه جدید یک الگوی طراحی یک مفهوم طراحی با قابلیت استفاده مجدد است.

تاکنون متوجه شده‌اید که می‌توان در طراحی از الگوهای طراحی چندبار استفاده نمود مثلاً زمانی که از یک کلاس در برنامه دو یا چندبار استفاده می‌کنید. این موضوع سبب می‌گردد تا از خواص سودمند برنامه‌نویسی شیءگرا (OOP) که همانا قابلیت استفاده مجدد می‌باشد، در طراحی شیءگرا (OOD) نیز استفاده کنیم. زمانی که از الگوهای طراحی استفاده می‌کنید، می‌دانید که طراحی دارای اساس و پایه محکم و قابل اعتمادی است و آزمایش خود را در طول زمان پس داده است. این نکته نیز که دیگران نیز از الگوهای طراحی به کرات استفاده کرده‌اند خود نقطه قوت خوبی است!

الگوهای طراحی

کتاب «الگوهای طراحی: عناصر با قابلیت استفاده مجدد در نرم‌افزارهای شیءگرا» نوشته گاما، هلم، جانسون و ولیساید در اولین قدم، اقدام به معرفی مفهوم الگوهای طراحی نموده است. در این کتاب با ارزش تنها به معرفی تعدادی از الگوهای طراحی اکتفا نشده است. بلکه مفهومی نیز که در پشت سر آن نهفته است، به دقت موشکافی شده است.

با توجه به کتاب فوق، می‌توان گفت یک الگوی طراحی شامل چهار عنصر است:

- نام الگو
- مسأله
- حل مسأله
- نتایج حاصله

نام الگو

برای هر الگوی طراحی یک نام واحد به کار می‌رود. همانگونه که UML (زبان مدلسازی یکپارچه) یک زبان طراحی مشترک ارایه می‌دهد، نام الگو نیز واژگان مشترکی را برای توصیف عناصر طراحی ارایه می‌کند. با توجه به این نکته، برنامه‌نویسان می‌توانند به راحتی طراحی صورت گرفته را درک کرده و تجزیه و تحلیل کنند.

یک نام ساده می‌تواند کل یک مسأله، حل آن و نتایج حاصله را در یک عبارت خلاصه کند. همچنانکه اشیاء به برنامه‌نویس کمک می‌کنند تا در سطحی بالاتر و حالتی انتزاعی برنامه‌نویسی کنند، اینگونه عبارات نیز اجازه می‌دهند طراحی در سطحی بالاتر و حالتی انتزاعی‌تر انجام گیرد و برنامه‌نویس را از درگیری با جزییات دست و پاگیر و عموماً تکراری نجات می‌دهد.

مسأله

هر الگوی طراحی برای حل تعدادی از مسایل قابل استفاده است. درواقع هر الگوی طراحی مجموعه‌ای از مسایلی که باید راه حلی برای آن طراحی کرد را توصیف می‌کنند. از این طریق می‌توان فهمید با توجه به مسأله چه الگویی را باید به آن اعمال کرد.

راه حل

راه حل نحوه حل مسأله توسط الگوی طراحی را مشخص می‌کند و اشیاء مهمی که جهت حل مسأله باید آنها را به خدمت گرفت و همچنین وظایف و روابط آنها را تعیین می‌کند.

نکته

ذکر این نکته ضروری است که می‌توان یک الگوی طراحی را به مجموعه‌ای از مسایل مشابه هم اعمال کرد و راه حل ارایه شده حل کلی مسأله است و به معنای آن نیست که مسأله راه حل دیگری ندارد.

فرض کنید می‌خواهید بهترین راه مرور آیتمهای موجود در کارت خرید (فصل ۱۰، مقدمه‌ای بر طراحی شیءگرا) را بیابید. الگوی طراحی تکرار یک راه برای انجام این کار ارایه می‌دهد. در حل فوق جواب مسأله در قالب آیتمهای موجود و یا کارت خرید ارایه نشده است در عوض راه حل ارایه شده نحوه فرایند مرور لیست انتخابی را بیان می‌کند.

زمانی که اقدام به استفاده از یک الگوی طراحی می‌کنید باید به نحوی راه حلی کلی اعمال شده را با مسأله سازش دهید. گاهی اوقات این کار به سختی صورت می‌گیرد. با این حال الگوهای طراحی باید حالت کلی خود را حفظ کنند تا بتوان آنها را به مسایل متنوع اما مشابه اعمال کرد.

نتایج

دیگر چیزی به جز طراحی کامل وجود ندارد. هر طراحی خوب نیازمند یکسری ملاحظات است و هر ملاحظه‌ای که صورت گیرد شامل مجموعه‌ای از نتایج خاص خود است. هر زمانی که بخواهید میان دو چیز یکی را انتخاب کنید باید ملاحظاتی را مد نظر داشته باشید. برای مثال به تفاوت میان استفاده از یک آرایه و یک لیست پیوندی (Linked List) توجه کنید. آرایه قابلیت جستجوی سریع با استفاده از اندیس آرایه را فراهم می‌کند حال آنکه اگر بخواهید آرایه را با حذف تعدادی از عناصر کوچک کنید و یا آن را با اضافه کردن تعدادی عنصر گسترش دهید، کار وقتگیری خواهد بود. در حالی که لیست پیوندی به آسانی و با سرعت قابلیت اضافه‌سازی و حذف عناصر را فراهم می‌کند. اما در عوض نیازمند حافظه بیشتر و دچار سرعت کمتر هستید. نتیجه آنکه انتخاب هریک از این دو ملاحظاتی را می‌طلبد که برنامه‌نویس باید مدنظر داشته باشد. انتخاب شما بستگی به این دارد که در طراحی به چه چیزی بیشتر اهمیت می‌دهید. حافظه یا سرعت؟ آیا برنامه نیاز به حذف و اضافه‌های زیاد دارد و یا آنکه سرعت برای جستجوی عناصر از اهمیت بیشتری برخوردار است.

بهتر است هر زمان که در مورد موضوعی تصمیم می‌گیرید، آن را مکتوب کنید تا در صورتی که دیگران بخواهند از نتایج کار شما استفاده کنند علت انتخاب شما و منطق برنامه را بفهمند.

در صورتی که الگوی طراحی نتایجی داشته باشد که با هدفتان سازگار نیست، نباید از آن استفاده کنید، حتی اگر امکان حل مسأله را برایتان فراهم سازد.

واقعیتهای الگوها

قبل از آنکه مطالبی را در مورد الگوها فرا بگیرید، مهم است که بدانید الگو چه کاری را انجام می‌دهد و چه کاری را انجام نمی‌دهد. به عبارت دیگر کارکرد الگو چیست؟ فهرست زیر مسایل مهمی را در مورد الگوها بیان می‌کند.

الگوها

- طراحی‌های قابل استفاده مجددی هستند که امتحان خود را پس داده‌اند.
- راه حلی مجرد برای مسایل کلی هستند.
- راه حلی برای مسایل بازگشتی هستند.
- و راهی برای ثبت و ضبط تجربیات طراحی می‌باشند.

در ضمن الگوها

- راه حل یک مسأله خاص نمی‌باشند.
- راه حلی جادویی برای تمام قسمتهای مسأله‌تان نیستند.
- درکنار الگوها باید قسمتهایی ازکار را خودتان انجام دهید.
- کلاسهای مجزا، کتابخانه‌ها و راه حلهای از پیش آماده نیستند.

الگوها از دریچه مثال

کتابهای زیادی در مورد الگوها و روشهای بکارگیری آن مطلب نوشته‌اند و هیچ دلیلی نیست که بخواهیم آن مطالب را در این بخش تکرار کنیم. با این حال تعدادی از الگوهای طراحی را به صورت روزمره می‌بینید و با آنها سروکار دارید و هیچ مطلب و کتابی در مورد شیءگرایی نیست که بتواند همه مطالب را بدون معرفی این دسته از الگوها به پایان ببرد.

به جای آنکه این الگوها را مستقیماً معرفی کنیم، بهتر است از چند مثال کمک بگیریم. در باقی مطالب امروز با سه الگوی مهم آشنا می‌شویم:

- الگوی وفق دهنده (Adapter)
- الگوی واسط (Proxy)
- الگوی تکرارکننده (Iterator)

الگوی وفق دهنده

در فصل چهارم، مفهوم روابط جانشینی معرفی شد. فصل ششم نشان داد که چگونه می‌توان از روابط فوق برای اضافه کردن اشیاء جدید به سیستم استفاده کرد. زمانی که از وراثت برای تعریف جانشینی استفاده

می‌شود اشیاء مجبورند که به نحوی در سلسله مراتب وراثت جای گیرند. درواقع برای آنکه بتوان آنها را در برنامه وارد کرد باید هریک از اشیاء جزیی از سلسله مراتب وراثت باشند. حال اگر بخواهیم شیئی را وارد برنامه کنیم که نمی‌تواند به نحو درست و شایسته‌ای در سلسله مراتب وراثت قرار گیرد، چه کار باید کنیم؟

یک راه حل برای این منظور ویرایش کلاس است به نحوی که بتوان آن را در جایگاه درستی از سلسله مراتب وراثت جای داد. با این حال این راه حلی جامع و بهینه نیست.

در ضمن عموماً سرس کد کلاسهایی را که باید کلاس جدید را از آنها مشتق کرد وجود ندارد و یا آنکه گاهی اوقات زبان برنامه‌نویسی مورد استفاده از وراثت چندگانه پشتیبانی نمی‌کند.

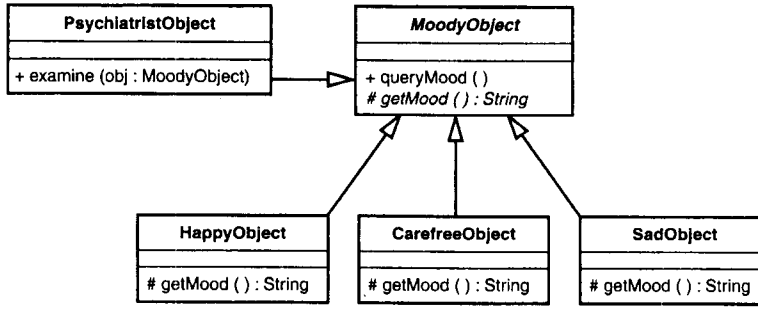
با داشتن سرس کد نیز منطقی نیست که هر زمان که بخواهیم کلاسی را در یک سلسله مراتب وراثت جای دهیم، آن را بازنویسی کنیم. دلیل آن هم روشن است: در هر برنامه، تعریف سلسله مراتب وراثت فرق می‌کند. بنابراین بازنویسی یک کلاس و تغییر آن برای هر برنامه کاری غیرمعقول است. تغییر و بازنویسی هر کلاس برای هر برنامه مغایر با اهداف OOP مبنی بر استفاده مجدد است. اگر به دنبال ایجاد نسخه‌های ویژه از یک کلاس برای هر برنامه باشیم، در مدت زمان کمی با حجم وسیعی از کلاسهای مازاد بر احتیاج روبرو خواهیم شد!

الگوی وفق دهنده راه حل دیگری را مطرح می‌کند که به حل مسأله ناسازگاری (از طریق تبدیل روابط ناسازگار به آنچه که مورد نیاز است) می‌پردازد. این الگو بر اساس قرارگیری شیء ناسازگار در درون شیء وفق دهنده مشکل را حل می‌کند. درواقع شیء وفق‌دهنده، نمونه‌ای از شیء ناسازگار را ایجاد کرده و از طریق رابطی که ارایه می‌دهد آن را به برنامه متصل می‌کند. از آنجایی که کلاس وفق دهنده، شیء ناسازگار را مخفی می‌کند (می‌پوشاند)، به الگوی پوشاننده نیز معروف است.

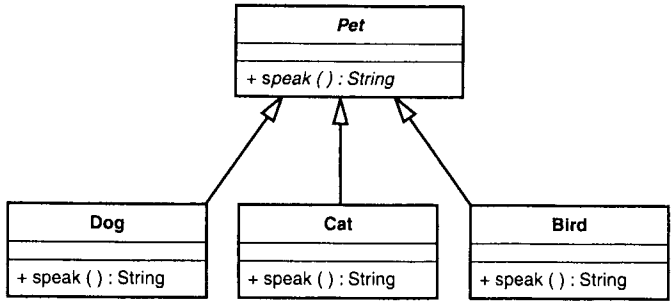
واژه جدید یک شیء وفق دهنده، تبدیل‌کننده یک رابط به یک شیء دیگر است.

پیاده‌سازی وفق‌دهنده

شکل ۱۱ ـ ۱ سلسله مراتب MoodyObject که در فصل هفتم نشان داده شد را نمایش می‌دهد. PsychiatristObject تنها می‌تواند با شیء از نوع MoodyObject (و کلاسهای مشتق شده از آن) کار کند. متد examine() از شیء PsychiatristObject متد queryMood() از شیء MoodyObject را فراخوانی می‌کند. دیگر انواع اشیاء نیازمند پیدا کردن نوع دیگری PsychiatristObject هستند.



شکل ۱۱ ـ ۱
سلسله مراتب MoodyObject



شکل ۱۱ ـ ۲
سلسله مراتب Pet

شکل ۱۱ ـ ۲ سلسله مراتب کلاس Pet را نمایش می‌دهد.

هر یک از اشیاء و کلاسهای از نوع Pet به شیوه خود کار می‌کنند و درواقع پیاده‌سازی خاصی را ارایه می‌دهند (متد speak()). خوب مشکل کجاست؟

اگر بخواهید دو شیء PsychiatristObject و Pet را با هم به کار ببرید، هیچ راهی وجود ندارد، چرا که دیگر متد queryMood() وجود ندارد و سلسله مراتب MoodyObject هم رعایت نشده است. برای حل مشکل نیازمند یک وفق دهنده هستیم. در این حالت وفق دهنده به عنوان پلی مابین این دو شیء ناسازگار عمل می‌کند. رابطی مابین Pet و MoodyObject.

لیست ۱۱ ـ ۱ یک راه پیاده‌سازی برای این وفق دهنده را نشان می‌دهد.

لیست ۱۱ـ۱ PetAdapter.java

```
public class PetAdapter extends MoodyObject {

    private Pet pet;

    public PetAdapter( Pet pet ) {
        this.pet = pet;
    }

    protected String getMood() {
        // only implementing because required to by
        // MoodyObject, since also override queryMood
        // we don't really need it
        return pet.speak();
    }

    public void queryMood() {
        System.out.println( getMood() );
    }
}
```

آنگونه که در کد فوق مشخص است کلاس PetAdapter شیء از نوع Pet را درون خود ایجاد کرده و با پیاده‌سازی متد getMood() رابطی بین دو شیء MoodyObject و Pet ایجاد کرده است.

لیست ۱۱ ـ ۲ کلاس وفق دهنده را در عمل نشان می‌دهد.

لیست ۱۱-۲ استفاده از کلاس وفق دهنده

```
PetAdapter dog=new PetAdapter(new Dog());
PetAdapter cat=new PetAdapter(new Cat());
PetAdapter bird=new PetAdapter(new Bird());
```

لیست ۱۱-۲ (ادامه)

```
PsychiatristObject psychiatrist= new PsychiatristObject();

psychiatrist.examine(dog);
psychiatrist.examine(cat);
psychiatrist.examine(bird);
```

پس از پنهان‌سازی شیء Pet در داخل کلاس وفق دهنده، از این شیء می‌توان به عنوان رابطی برای کلاس PsychiatristObject استفاده کرد. این راه حل به مراتب بهتر و بهینه‌تر از زمانی است که بخواهیم سلسله مراتب و کد کلاسهای جدید را تغییر دهیم. از طریق وفق دهنده فوق می‌توان هر نوع شیء از کلاس Pet را پشتیبانی نمود.

پیاده‌سازی ارایه شده وفق دهنده شیء (Object Adaptor) نامیده می‌شود چرا که کلاس وفق دهنده با استفاده از ایجاد شیء از کلاس ناسازگار مبادرت به تبدیل و ایجاد ارتباط نموده است. این در حالی است که می‌توان تطبیق را از راه وراثت نیز انجام داد. در این حالت وفق دهنده، وفق دهنده کلاس (Class Adaptor) نامیده می‌شود. چرا که خودِ تعریفِ کلاس را در این حالت با آنچه می‌خواهیم وفق داده است.

در صورتی که زبان برنامه‌نویسی که با آن کد می‌نویسید از وراثت چندگانه پشتیبانی کند، می‌توانید کلاسی ایجاد کنید که از کلاس مدنظر در این حالت (Pet) و کلاس موجود در سلسله مراتب وراثت در این حالت (MoodyObject) به صورت توأمان مشتق شده باشد.

اگر زبان برنامه‌نویسی مورد نظر شما از وراثت چندگانه پشتیبانی نمی‌کند (نظیر Java) باید از روش ترکیب (روش بالا) استفاده کرده و وفق دهنده شیء بسازید. البته استفاده از وراثت چندگانه در زبان‌هایی که از آن پشتیبانی می‌کنند نیز محدودیتهایی را به همراه دارد. چرا که ساخت زیرکلاس برای هر کلاس از نوع Pet خود تا حدودی غیرقبول است!

هر روش به هر حال محدودیتهایی را نیز با خود همراه دارد. وفق دهنده کلاس تنها برای همان کلاس کار می‌کند. بنابراین برای هر زیرکلاس نیز باید وفق دهنده‌ای خاص ایجاد کرد.

چه زمانی از الگوی Adapter باید استفاده کرد

الگوی وفق دهنده زمانی که بخواهیم شیء ناسازگاری را با دیگر اشیاء به کار ببریم، سودمند است. از این طریق می‌توان مستقیماً از کلاسها و کدهای موجود استفاده مجدد کرده و آنها را اگرچه با یکدیگر ناسازگار هستند باهم به خدمت گرفت.

از این الگو می‌توان در موارد دیگری نیز استفاده کرد. بسیاری از پروژه‌ها از کلاسها و توابع ارایه شده در

کتابخانه‌های مختلف (وابسته به سازندگان مختلف) تشکیل شده‌اند. با گذشت زمان کلاسها، توابع و اشیاء تغییر پیدا کرده و می‌تواند موجبات بروز مشکل را فراهم نمایند. با استفاده از این الگو می‌توان این توابع و کلاسها را از باقی برنامه جدا (ایزوله) کرد.

از الگوی وفق دهنده در موارد زیر استفاده نمایید:

- زمانی که بخواهید از اشیاء ناسازگار با باقی اشیاء در برنامه استفاده کنید.
- زمانی که بخواهید برنامه از کلاسها و توابع سازندگان مختلف ایزوله باشد.

جدول ۱۱ ـ ۱ مشخصات این الگو را نشان می‌دهد.

جدول ۱۱ـ۱ الگوی وفق دهنده

نام الگو	وفق دهنده، پوشاننده
مسأله	چگونه می‌توان از اشیاء ناسازگار در برنامه استفاده مجدد کرد.
راه حل	شی را به عنوان رابط (مبدل) بین کلاس ناسازگار و کلاسهای موجود ارایه می‌کند.
نتایج	اشیاء ناسازگار از این طریق سازگار می‌شوند. نتایج در قالب کلاسهای بیشتر، گاهی اوقات مشاهده می‌شود. این الگو را می‌توانید از طریق کلاسهای جدید (برای زبان‌هایی که از وراثت چندگانه پشتیبانی نمی‌کنند) و یا وراثت چندگانه از کلاسهای ناسازگار پیاده‌سازی نمود.

الگوی واسط

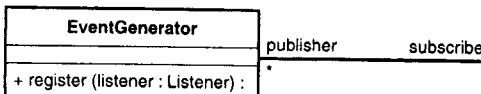
به طور طبیعی زمانی که شیء بخواهد با شیء دیگر در ارتباط و تعامل باشد، این کار را به صورت مستقیم انجام می‌دهد. در بسیاری از حالات، روش مستقیم فراخوانی بهترین روش است. با این حال مواقعی پیش می‌آید که بهتر است این روش به صورت غیرمستقیم انجام گیرد. الگوی واسط به این مقوله می‌پردازد.

انتشار/اشتراک و الگوی واسط

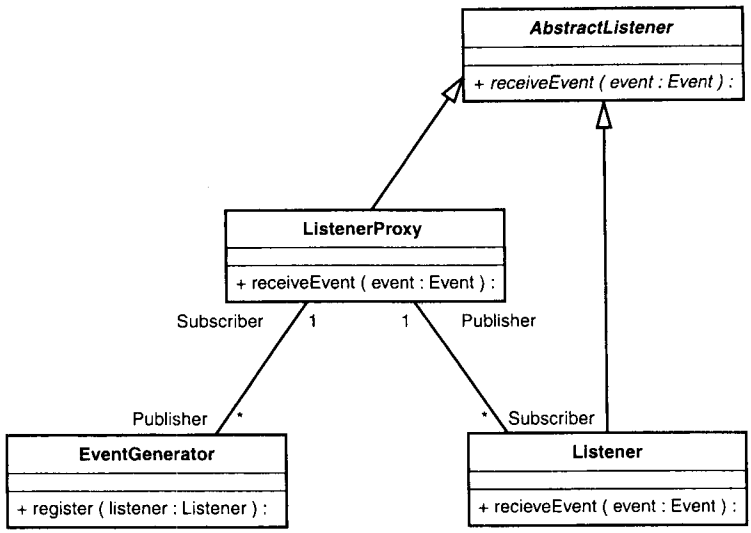
به مسأله انتشار (Publish)/اشتراک (Subscribe) دقت کنید. در این حالت یک شیء رخدادی را اعلام می‌کند و شیء دیگر در صورتی که آمادگی خود را برای دریافت رخداد اعلام کرده باشد (register)، رخداد را دریافت می‌کند. در این حالت یکی ناشر/فرستنده و دیگری گیرنده است. به عبارت دیگر ناشر، گیرنده را از وقوع رخدادی باخبر می‌کند.

شکل ۱۱ ـ ۳ یک رابطه ممکن بین انتشار/اشتراک را نشان می‌دهد.

در شکل ۱۱ ـ ۳ گیرندگان بسیاری می‌توانند برای دریافت رخدادها اعلام آمادگی کنند. در این صورت اگر رخدادی توسط مولد رخداد (Event Generator) تولید شود، رخداد صورت گرفته را برای تک‌تک گیرندگان ارسال می‌کند.



شکل ۱۱ـ۳
رابطه انتشار/اشتراک



شکل ۱۱ ـ ۴
راه حل مبتنی بر الگوی واسط

اگرچه این روش کار می‌کند ولی بار سنگینی را بر دوش EventGenerator تحمیل می‌کند. درواقع نه تنها مسئولیت تولید رخدادها بر دوش آن کلاس است بلکه مسئولیت ردیابی و ارسال رخداد به تمام گیرندگان نیز بر دوش آن تحمیل می‌گردد. الگوی واسط راه حل بسیار خوبی برای این مشکل است. شکل ۱۱ـ۴ را در نظر بگیرید. به جای آنکه مستقیماً با گیرندگان تماس حاصل شود، EventGenerator پیغام یا رخداد را به ListenerProxy تحویل می‌دهد. این کار یک بار صورت می‌گیرد. حال وظیفه ListenerProxy آن است که تمام گیرندگان را از رخداد جدید باخبر کند.

الگوی واسط کلی

سناریوی انتشار/اشتراک یک مورد از موارد استفاده از الگوی واسط را نشان می‌دهد. با این حال می‌توان از واسطها در بسیاری از جاها استفاده کرد.

اما یک واسط چیست؟

یک واسط جانشین یا نگهدارنده‌ای است که از طریق آن می‌توان به اشیاء دلخواه دسترسی داشت. از یک شیء از نوع واسط (Proxy) زمانی که بخواهیم از آن به عنوان جانشین و یا نگهدارنده‌ای برای یک شیء دیگر استفاده کنیم، سود می‌بریم. در این حالت، شیء واسط موظف به برقراری و نگهداری تمام جزییات مربوط به ارتباط میان شیء اصلی و شیئی که دسترسی به آن لازم است می‌باشد.

واژه جدید

یک واسط جانشین و یا نگهدارنده‌ای برای شیء دیگر است. دسترسی به شیء مورد نظر همواره از طریق این واسط صورت می‌گیرد و در نتیجه یک واسط را نمی‌توان از شیء مورد نظر جدا کرد.

دسترسی به شیء مورد نظر به صورت غیرمستقیم و از طریق واسط صورت می‌پذیرد. برای مثال شیء EventGenerator می‌پندارد که در آن واحد تنها با یک شیء در ارتباط است این در حالی است که ListenerProxy درواقع با گیرندگان مختلفی در ارتباط است. شیء واسط باعث می‌شود که تمام مسئولیتها به جای آنکه در یک کلاس جمع شود، مجزا گشته و مسئولیتهای مرتبط از غیرمرتبط جدا گردند. با این ترتیب تولید یک رخداد توسط یک شیء و انتشار و اعلام آن توسط شیء دیگر صورت می‌پذیرد.

چه زمانی از الگوی واسط باید استفاده کرد؟

از الگوی واسط در موارد زیر استفاده می‌شود:

- زمانی که بخواهیم اعمال وقتگیر و سنگین را در مواردی که نیاز است پوشش دهیم. فرض کنید شیء اطلاعاتی را از یک پایگاه داده‌ای گرفته و به برنامه برمی‌گرداند. اگرچه برنامه به داده‌های بازگشتی محتاج است، با این حال همیشه لازم نیست تمام اطلاعات برگردانده شوند. برای این منظور می‌توان از واسط استفاده کرد. واسط اطلاعات را گرفته و آنهایی را که برنامه احتیاج دارد برمی‌گرداند. در صورت درخواست برنامه برای دریافت اطلاعات بیشتر، باقی اطلاعات نیز ارسال می‌شوند.
  یک مثال دیگر استفاده از واسط فایل (File Proxy) است. در این حالت الگوی واسط تنها زمانی که کارتان با فایل تمام شد، عملیات IO را انجام می‌دهد. مثلاً به جای نوشتن‌های زیاد و کند بر روی فایل، آن را به یک باره و در انتها انجام می‌دهد، بدون آنکه اطلاعاتی در این میان گم شود.
- زمانی که بخواهید به طور غیرمحسوس نحوه استفاده از شیء را کنترل کنید. بسیاری از اشیاء قابل تغییرند. نظیر کلاسهای کلکسیون (Collection). از طریق یک رابط محافظ (Protective Proxy) می‌توان درخواستها (فراخوانی متدها) را گرفته و سپس آنها را بر روی کلکسیون مورد نظر اعمال کرد. درواقع با این کار دسترسی مستقیم به کلاسهای کلکسیون محدود شده است.
- زمانی که شیء مورد نظر از طریق یک شبکه و یا یک برنامه دیگر قابل دسترسی باشد از واسطها استفاده می‌کنیم. درواقع واسطها به هنگام نوشتن الگوریتمهایی که در محیطهای توزیع شده به کار می‌روند، بسیار مفید فایده هستند. از طریق یک واسط می‌توان از یک منبع توزیع شده (Distributed Resource) همانند یک منبع محلی (Local Resource) استفاده کرد. این کار از طریق ارسال درخواستها بر روی شبکه صورت می‌گیرد.
- زمانی که بخواهیم اعمال بیشتری را بر روی یک شیء به طور نامحسوس انجام دهیم از واسطها استفاده می‌کنیم. فرض کنید بخواهیم تعداد دفعاتی که یک متد فراخوانی می‌شود را بشماریم، بدون آنکه فراخواننده بفهمد. می‌توان از یک واسط شمارشگر برای این منظور استفاده کرد. جدول ۱۱ ـ ۲ مشخصات الگوی واسط را نشان می‌دهد.

جدول ۱۱-۲ الگوی واسط

نام الگو	واسط، جانشین (Surrogate)
مسأله	زمانی که نیاز به کنترل یک شیء است.
راه حل	شیئی را ارایه می‌کند که از طریق آن می‌توان به طور نامحسوس به شیء مورد نظر دسترسی داشت.
نتایج	سطحی از دسترسی غیرمستقیم به یک شیء را ارایه می‌دهد.

الگوی تکرار

الگوهای طراحی عموماً راه حلی برای مسایل مشترک ارایه می‌کنند. بسیار پیش می‌آید که نیاز است با را استفاده از حلقه‌ها کدهایی نوشت که اشیاء درون یک کلکسیون (Collection) شمارش کرد و به آنها دسترسی پیدا نمود.

لیست ۱۱-۳ حلقه‌ای برای شمارش و دسترسی به کارتهای موجود در میز کارتها

```
public String deckToString( Deck deck ) {
    String cards = "";
    for( int i = 0; i < deck.size(); i++ ) {
        Card card = deck.get( i );
        cards = cards + card.display();
    }
    return cards;
}
```

لیست ارایه شده یک روش عمومی برای دسترسی به آیتمهای موجود در کلکسیون را نشان می‌دهد. در این حالت، کارتهای موجود بر روی میز شمارش می‌شوند.

فرض کنید بخواهید این کار را بر روی آرایه‌ای از کارتها انجام دهید. لیست ۱۱ ـ ۴ کدهای مربوط به این حالت را در بر دارد.

لیست ۱۱-٤ حلقه‌ای برای دسترسی به عناصر آرایه

```
public String deckToString( Card [] deck ) {
    String cards = "";
    for( int i = 0; i < deck.length; i++) {
        cards = cards + deck[i].display();
    }
    return cards;
}
```

در آخر آنکه اگر بخواهید دسترسی به کارتها را به صورت معکوس انجام دهید، نیازمند متد دیگری هستید.لیست ۱۱ ـ ۵ این متد را نشان می‌دهد.

لیست ۱۱-۵ دسترسی به کارتهای موجود بر روی میز کارتها به صورت معکوس

```
public String reverseDeckToString( Deck deck ) {
    String cards = "";
    for ( int i = deck.size() - 1; i > -1; i--) {
        Card card = deck.get( i );
        cards = cards + card.display();
    }
    return cards;
}
```

هر زمان که بخواهید به عناصر موجود در یک آرایه و یا یک کلکسیون دسترسی داشته باشید، باید از یک حلقه استفاده کنید. بنابراین باید برای هر حلقه متدی بنویسید که نحوه دسترسی به انواع عناصر موجود در کلکسیونهای مختلف را بداند. در این حالت تفاوت موجود در متدهای مختلف در خواص و متدهای دسترسی به کلکسیون است. بنابراین منطق کدهای نوشته شده برای همه حلقه‌ها یکی است.

مسأله در اینجاست که متدهای نوشته شده در بالا تنها در پیاده‌سازی کلکسیون تفاوت دارند. بنابراین اگر نحوه پیاده‌سازی کلکسیون تغییر پیدا کند، باید تمام کدهای مربوط به دسترسی به کلکسیون را در برنامه تغییر دهید. این کار ممکن است تمام برنامه را تحت تأثیر قرار دهد.

نکته اگرچه یک شیء ممکن است تنها شامل یک حلقه باشد، برنامه ممکن است در مکانهای مختلف به کرات از اینگونه حلقه‌ها استفاده کرده باشد.

مکانیزمهای مختلف برای دسترسی به آیتمهای موجود در یک کلکسیون ایجاب می‌کند برای هریک از آنها متدی جداگانه (و در نتیجه حلقه‌ای متفاوت) نوشته شود.

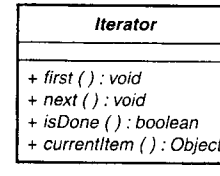
خوشبختانه الگوی تکرار کمکهای زیادی می‌تواند به ما کند و مشکلات زیادی را از دوش ما برمی‌دارد. شکل ۱۱ ـ ۵ رابط Iterator را نشان می‌دهد.

Iterator رابطی کلی برای گردش در یک کلکسیون را در اختیار می‌گذارد. به جای نوشتن تعدادی متد و حلقه برای یک کلکسیون خاص به راحتی می‌توان از این رابط استفاده کرد. رابط Iterator تمام پیاده‌سازیهای مرتبط با کلکسیون را در خود پنهان کرده است.

نکته Java رابط Iterator دیگری را تعریف کرده است: java.util.Iterator. تکرارکننده موجود در Java تنها شامل سه متد است: public Object next(), public void remove(), public boolean hasNext().

رابط Iterator موجود در Java تنها اجازه می‌دهد که گردش در کلکسیون تنها در یک جهت صورت گیرد و با رسیدن به آخرین آیتم موجود در کلکسیون امکان بازگشت به عنصر ابتدایی وجود ندارد. غیر از این مورد تفاوتی بین این دو تکرارکننده وجود ندارد.

به هنگام نوشتن کدهای Java، به خاطر داشته باشید که از تکرارکننده آن (java.util.Iterator) باید استفاده کنید تا دیگر برنامه‌های Java بتوانند از آن به راحتی استفاده کنند. برای اهداف این کتاب، درس امروز از تکرارکننده کلاسیکی که در شکل ۱۱ ـ ۵ نمایش داده شده است، استفاده می‌کند.



شکل ۱۱ ـ ۵
رابط Iterator

لیست ۱۱ ـ ۶ متد deckToString() را به روش دیگری ارایه می‌کند.

لیست ۱۱ـ۶ گردش در محتویات یک نمونه از تکرارکننده

```
public String deckToString( Iterator i ) {
    String cards = "";
    for ( i.first(); !i.isDone(); i.next() ) {
        Card card = (Card) i.currentItem();
        cards = cards + card.display();
    }
    return cards;
}
```

مقدار بازگشتی توسط تکرارکننده شیء از نوع Object است و البته این بدان معنا نیست که آیتمها داخل خود تکرارکننده ذخیره شده‌اند. تکرارکننده تنها دسترسی به محتویات خود شیء را فراهم می‌کنند.

سه مزیت مهم در دسترسی به اشیاء و آیتمهای موجود در یک کلکسیون با استفاده از یک تکرارکننده وجود دارد (نسبت به سایر موارد نظیر حلقه‌ها): اول آنکه تکرارکننده شما را محدود به یک کلکسیون خاص نمی‌کند. درواقع با استفاده از متدهای ارایه شده می‌توان همه انواع کلکسیون‌ها را پشتیبانی نمود.

دوم آنکه تکرارکننده‌ها می‌توانند عناصر را به هر ترتیب دلخواه (صعودی، نزولی، از اول به آخر و یا از آخر به اول) برگردانند. به عبارت دیگر محدودیتی در نحوه برگرداندن عناصر وجود ندارد.

در آخر آنکه با استفاده از تکرارکننده‌ها، تغییرات صورت گرفته در کلکسیونها باعث نمی‌شود که برنامه‌نویس مجبور شود دیگر اجزای برنامه را تغییر دهد. با این ترتیب برنامه‌نویس می‌تواند به محض آنکه اراده کند، کلکسیون را نیز عوض کند بدون آنکه به باقی برنامه لطمه‌ای وارد شود.

پیاده‌سازی و تکرارکننده

نکته

بسیاری از کلکسیونهای موجود در Java از تکرارکننده‌ها پشتیبانی می‌کند. برای مثال LinkedList استفاده شده در شی Deck دارای متد ()iterator است که شی از نوع java.util.Iterator را برمی‌گرداند.

لیست ۱۱ـ۷ نحوه پیاده‌سازی یک تکرارکننده که به کاربر اجازه می‌دهد عناصر موجود در یک کلکسیون از نوع LinkedList را مرور کند، را نشان می‌دهد.

لیست ۱۱ـ۷ پیاده‌سازی تکرارکننده

```
public class ForwardIterator implements Iterator {

    private Object [] items;
    private int index;

    public ForwardIterator( java.util.LinkedList items ) {
        this.items = items.toArray();
    }

    public boolean isDone() {
        if( index == items.length ) {
            return true;
        }
        return false;
    }

    public Object currentItem() {
        if( !isDone() ) {
            return items[index];
        }
```

لیست ۱۱ـ۷ (ادامه)

```
        return null;
    }

    public void next() {
        if( !isDone() ) {
            index++;
        }
    }

    public void first() {
        index = 0;
    }
}
```

برای مشاهده تکرارکننده معکوس خواهشمند است کد کامل را از اینترنت دریافت کرده و مشاهده کنید.

لیست ۱۱ ـ ۸ تغییراتی را که باید بر روی کلاس Deck اعمال نمود تا از یک تکرارکننده استفاده نماید، نشان می‌دهد.

لیست ۱۱ـ۸ کلاس Deck تغییریافته

```
public class Deck {
    private java.util.LinkedList deck;

    public Deck() {
        builCards();
    }

    public Iterator iterator() {
        return new ForwardIterator( deck );
    }
    // snipped for brevity
}
```

از یک نقطه نظر دیگر می‌توان گفت تکرار کننده خود به نوعی یک کلکسیون است که اجازه دسترسی به عناصر را فراهم می‌کند. با توجه به این موضوع انتخابهای دیگری را نیز به هنگام پیاده‌سازی می‌توان مدنظر داشت.

یک کلاس درونی کلاسی است که داخل کلاس دیگری تعریف می‌شود. با توجه به این موضوع می‌توان گفت که کلاس درونی امکان دسترسی به تمام اجزای کلاس بالاتر را دارد. این اجزا می‌توانند متغیرها و متدهای عمومی (public)، خصوصی (private) و محافظت شده (protected) باشند. در C++ این نوع کلاسها را کلاسهای دوست (friend) می‌گویند. از طریق یک کلاس دوست

می‌توان به اجزای یک شیء (یا به طور کلی یک کلاس) دیگر دسترسی داشت.

با این حال هر دو مفهوم کلاس درونی و کلاس دوست، قاعده کپسوله‌سازی را زیر پا می‌گذارند ولی در هر حال یک تکرارکننده را می‌توان جزیی از یک کلکسیون دانست. لیست ۱۱ـ۹ پیاده‌سازی یک تکرار کننده را به صورت یک کلاس درونی نشان می‌دهد.

لیست ۱۱ـ۹ پیاده‌سازی تکرارکننده به صورت یک کلاس درونی

```
public class Deck {

    private java.util.LinkedList deck;

    public Deck() {
        buildCards();
    }

    public Iterator iterator() {
        return new ForwardIterator();
    }
    //snipped for brevity

    private class ForwardIterator implements Iterator {
        int index;

        public boolean isDone() {
            // notice that the inner class has unfettered
            // access to Deck's internal variable deck
            if( index == deck.size() ) {
                return true;
            }
            return false;
        }

        public Object currentItem() {
            if( !isDone() ) {
                return deck.get( index );
            }
            return null;
        }

        public void next() {
            if( !isDone() ) {
                index++;
            }
        }
```

لیست ۱۱-۹ (ادامه)

```
        public void first() {
            index = 0;
        }
    }

}
```

چه زمانی باید از الگوی تکرارکننده استفاده کرد

دلایل چندی برای استفاده از این الگو وجود دارد:

- زمانی که بخواهید پیاده‌سازی کلکسیون را مخفی نمایید می‌توانید از این الگو استفاده نمایید.
- زمانی که بخواهید انواع مختلفی از حلقه‌ها را برای شمارش و دستیابی به عناصر یک کلکسیون به خدمت بگیرید، می‌توان از این الگو استفاده نمایید.
- از این الگو برای ساده نگهداشتن رابط یک کلکسیون می‌توان استفاده کرد. دیگر نیازی به متدهای اضافی جهت دسترسی به عناصر یک کلکسیون نیست.
- می‌توان کلاس پایه‌ای تعریف نمود که یک تکرارکننده را برمی‌گرداند. در صورتی که همه کلکسیونهای برنامه از این کلاس مشتق شوند، این تکرارکننده می‌تواند برای همه آنها استفاده شود. java.util.Collection نیز برای همین منظور تهیه شده است. این نحوه استفاده، فرم کلی الگوی تکرارکننده را نشان می‌دهد.
- تکرارکننده‌ها دسترسی بهینه به عناصر کلکسیونها را ارایه می‌دهند. دیگر ساختمانهای داده‌ای نظیر hashtable راهی بهینه برای دسترسی به عناصر یک کلکسیون به حساب نمی‌آیند. تنها عیب (نه چندان مهم) تکرارکننده‌ها اشغال حافظه به میزان بیشتری نسبت به سایر ساختمانهای داده‌ای است.

جدول ۱۱ ـ ۳ مشخصات الگوی تکرار کننده را نشان می‌دهد.

جدول ۱۱-۳ الگوی تکرارکننده

نام الگو	تکرارکننده، کرسر
مسأله	بدون وابستگی به پیاده‌سازی کلکسیون، می‌تواند دسترسی به اعضای آن را فراهم کند.
راه حل	از طریق ارایه شیئی که جزییات آن برای کاربر مخفی است، دسترسی به عناصر کلکسیون را فراهم می‌کند.
نتایج	رابطی ساده برای کلکسیون فراهم کرده و منطق حلقه دسترسی راکپسوله می‌نماید.

استفاده از بعضی از الگوها از دیگر الگوها سخت‌تر است. با این حال پیچیدگی استفاده از آنها مانع از استفاده از آنها نمی‌شود. قبل از آنکه الگویی را اعمال کنید و یا در مورد الگویی دیگر فکر کنید، باید الگوی مورد نظر را خوب فهمیده باشید. بنابراین بهتر است قبل از هر کاری مراحل زیر را طی کنید:

۱. الگو و کارکرد آن را خوب مطالعه کنید.
۲. دوباره الگو و کارکرد آن را مطالعه کرده و توجه کافی به نحوه استفاده از آن بنمایید. در صورت لزوم

نگاهی به کدهای نمونه و مثالهای آن نیز بیاندازید.

۳. پیاده‌سازی و نحوه استفاده از الگو را به صورت عملی تمرین کنید.

٤. در انتها الگو را به مسأله حقیقی خود اعمال نمایید.

پس از طی چهار مرحله فوق و فهم کامل، الگو حاصل شده است و می‌توان آن را به کار گرفت.

خلاصه

الگوهای طراحی یکی از مسایل بسیار مهم به هنگام طراحی راه حل مسأله می‌باشند. الگوها ماحصل سالها تجربه و تلاش در زمینه روش شیءگرا هستند و راهنماییهای باارزشی را برای برنامه‌نویس به ارمغان می‌آورند.

به هنگام استفاده از الگوهای طراحی باید محدودیتهای آن را نیز به خاطر داشته باشید. هر الگوی طراحی تنها مناسب یک مسأله و یک راه حل عمومی است. به خاطر داشته باشید که الگوهای طراحی جواب منحصر به فرد و کامل یک مسأله خاص نیستند. درواقع یک الگو یک راه حل انتزاعی را ارایه می‌کنند و این وظیفه شماست که آن را به نحوی که می‌خواهید به خدمت بگیرید.

عموماً نگاشت یک الگوی طراحی به راه حل مورد نظر مهمترین چالشی است که با آن روبرو می‌شوید. تنها با مطالعه بیشتر، تمرین کافی و گذاشتن وقت است که می‌توان به نحوه استفاده از الگوها مسلط شد.

پرسشها و پاسخها

آیا زبان Java از الگوها استفاده می‌کند؟

بله، بسیاری از توابع موجود در به Java از الگوها بهره می‌برند. در حقیقت همه الگوهایی راکه در این درس آموختید، در Java کار گرفته شده‌اند. در زیر فهرستی از الگوهای به کار رفته نشان داده شده است.

الگوی تکرار کننده: کلاسهای مرتبط با کلکسیونها

الگوی واسط: کلاسهای RMI

الگوی تطبیق دهنده: در گیرنده‌های رخدادها به کرات استفاده شده‌اند.

آیا همه الگوها به همه زبانها ترجمه شده‌اند (تبدیل شده‌اند)؟

خیر. زبانها با یکدیگر متفاوتند. بکارگیری برخی از الگوها در یکسری از زبانها امکانپذیر نیست و تعدادی از آنها نیز غیرضروری هستند. (به دلیل ویژگیهای درونی آن زبان).

کارگاه

سؤالات مطرح شده در قسمت زیر تنها برای فهم بیشتر شما از مطالب ارایه شده است.

پرسشها

۱. **یک کلاس تطبیق دهنده به چه منظوری استفاده می‌شود؟**

۲. **الگوی تکرارکننده برای حل چه مسأله‌ای مفید است؟**

۳. **چرا باید از الگوی تکرارکننده استفاده کرد؟**

٤. **الگوی تطبیق دهنده برای حل چه مسأله‌ای مفید است؟**

۵. چرا از الگوی تطبیق دهنده استفاده می‌کنیم؟
۶. الگوی واسط برای حل چه مسأله‌ای مفید است؟
۷. چرا باید از الگوی واسط استفاده کرد؟
۸. برای مسأله زیر از چه الگویی استفاده می‌کنید؟ و چرا؟

شرکتهای سان میکروسیستم، IBM و آپاچی برای تحلیل فایلهای XML هریک کتابخانه‌ای از توابع را ارایه می‌دهند. شما تصمیم می‌گیرید برای برنامه‌تان از کتابخانه شرکت آپاچی استفاده نمایید. در آینده تصمیم می‌گیرید از کتابخانه‌ای که شرکت دیگری برای تحلیل فایلهای XML ارایه داده است، استفاده کنید.

۹. مسأله زیر را در نظر بگیرید. چه الگویی را برای آن پیشنهاد می‌کنید؟ چرا؟

باید برنامه‌ای بنویسید که مقادیری داده را از فایلی بخواند. گاهی اوقات نیاز است داده‌ها از فایل محلی و گاهی اوقات از روی سرویس دهنده دوری (Remote Server) فراخوانده شوند.

۱۰. آیا الگوی واسط رابط یک شی (Object's interface) را تغییر می‌دهد؟

تمرین‌ها

۱. لیستهای ۱۱ ـ ۱۰ و ۱۱ ـ ۱۱ کلاسهای مربوط به کارت خرید و آیتمهای موجود در کارت خرید را نشان می‌دهند. لیست ۱۱ ـ ۱۲ رابط یک تکرارکننده را نمایش می‌دهد. با استفاده از کدهای ارایه شده، تکرارکننده‌ای طراحی کنید تا آیتمهای موجود در کارت خرید را شمارش کند.

لیست ۱۱-۱۰ Item.java

```
public class Item {

    private int     id;
    private int     quantity;
    private float   unit_price;
    private String  description;
    private float   discount;

    /**
    *   Create a new item with the given quantity, price,
    *   description, and unit discount.
    *   @param id the product id
    *   @param quantity the number of items selected
    *   @param unit_price the before discount price
    *   @param description the product description
    *   @param discount the dollar amount to subtract per item
    */
    public Item( int id, int quantity, float unit_price, float discount, String desc) {
        this.id = id;
        this.quantity = quantity;
        this.unit_price = unit_price;
        this.discount = discount;
        this.description =  description;
```

لیست ۱۱-۱۰ (ادامه)

```
    }

    /**
    *    @return int the item quantity
    */
    public int getQuantity() {
        return quantity;
    }

    /**
    *    @param quantity the new quantity
    */
    public void setQuantity( int quantity ) {
        this.quantity = quantity;
    }


    /**
    *    @return the item unit price
    */
    public float getUnitPrice() {
        return unit_price;
    }

    /**
    *    @return float the total price of the item minus any discounts
    */
    public float getTotalPrice() {
        return ( unit_price * quantity ) - ( discount * quantity );
    }

    /**
    *    @return String the product description
    */
    public String getDescription() {
        return description;
    }

    /**
    *    @return int the product id
    */
    public int getID() {
        return id;
    }

}
```

لیست ۱۱-۱۱ ShoppingCart.java

```
public class ShoppingCart {

    java.util.LinkedList items = new java.util.LinkedList();

    /**
    * adds an item to the cart
    * @param item the item to add
    */
    public void addItem( Item item ) {
        items.add( item );
    }

    /**
    * removes the given item from the cart
    * @param item the item to remove
    */
    public void removeItem( Item item ) {
        items.remove( item );
    }

    /**
    * @return int the number of items in the cart
    */
    public int getNumberItems() {
        return items.size();
    }

    /**
    * retrieves the indexed item
    * @param index the item's idex
    * @return Item the item at index
    */
    public Item getItem( int index ) {
        return (Item) item.get( index );
    }
}
```

لیست ۱۱-۱۲ Iterator.java

```
public interface Iterator {
    public void first();
    public void next();
    public boolean isDone();
    public Object currentItem();
}
```

۲. کلاس PetAdapterارایه شده در ابتدای فصل تنها یک نمونه از کلاس Petرا در خود نگهداشته است. تطبیق دهنده را به نحوی تغییر دهید تا بتوان در هر لحظه شیء موجود در آن را تغییر داد.

به نظر شما چرا باید از یک تطبیق دهنده قابل تغییر (mutable adapter) استفاده کرد؟

روز ۱۲

Object Orinted Programming

الگوهای پیشرفته طراحی

در فصل روز گذشته دیدیم که چگونه الگوهای طراحی استفاده مجدد از کد طرح‌های آزمایش شده را ممکن می‌کنند. امروز، بحث الگوها را با بررسی سه الگوی دیگر ادامه می‌دهیم.

آنچه امروز خواهید آموخت

- سه الگوی مهم طراحی
- چگونه از منفرد ماندن شیء مطمئن شویم.
- چگونه یک مثال قبلی را بهبود دهیم
- دامها

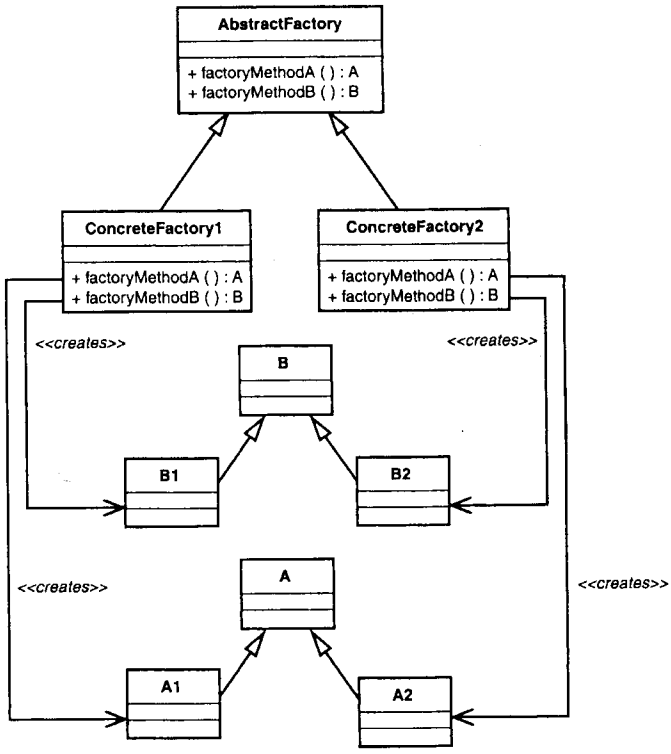
مثالهای دیگر از الگوها

بهتر است بحث الگوها را با در نظر گرفتن سه الگوی مهم ادامه دهیم:

- کلاس مجرد عامل
- تک‌برگ
- شمارشی با نوع محافظت شده

هریک از این الگوها راه خود را در اکثر طراحی‌ها پیدا می‌کنند.

شکل ۱۲ـ۱
الگوی عامل مجرد



الگوی مجرد عامل

قبلاً در فصل هفتم دیدیم که می‌توان با ترکیب وراثت و چندشکلی نرم‌افزاری آینده‌نگر نوشت. روابط اتصال‌پذیری حاصل از وراثت در ترکیب با چندشکلی بودن اجازه اتصال اشیاء تازه به برنامه را در هر زمانی ممکن می‌کند، با این حال در این مورد جنبه‌ای منفی هم وجود دارد.

برای اینکه متغیرهایی از نوع اشیاء جدید تعریف کنید، باید به کد بازگردید و آن را دستکاری کنید. آیا بهتر نبود که برای این کار راه ساده‌تری یافت می‌شد؟

الگوی جاری، این مسأله را از راه واگذاری حل می‌کند. به جای تعریف متغیر از نوع شیء مزبور، وظیفه مورد نظر را به شیئی به نام عامل (Factory) واگذار می‌کنیم. هرگاه شیء نیاز داشت که شیئی دیگر ایجاد کند، آن را از عامل درخواست می‌کند. با استفاده از عامل، می‌توان تمام فرایندهای ایجاد شیء را در یک جا جمع کرد. بنابراین هرگاه نیاز به ایجاد شیء جدید وجود داشت، کافی است فقط عامل را طوری به روزآوری کرد که توانایی ایجاد شیء جدید را داشته باشد. اشیایی که از عامل استفاده می‌کنند، هرگز متوجه تفاوتی نخواهند شد.

شکل ۱۲ ـ ۱ طرح عام الگوی عامل مجرد را نشان می‌دهد.

این عامل مجرد از وراثت و اتصال‌پذیری تواماً بهره می‌گیرد. کلاس عامل پایه تمام روالهای ایجاد اشیاء را تعریف می‌کند و هر زیرکلاس آن روش ایجاد اشیاء مورد نظر خود را با جایگزینی روالهای مطلوب تعریف می‌کند.

پیاده‌سازی یک عامل مجرد

گاهی وقتی از کتابخانه‌های دیگران استفاده می‌کنید، می‌بینید که در نسخه‌های جدید رابط عمومی API آنها

تغییر کرده. خوشبختانه الگوی عامل مجرد راه حلی برای این معضل ارایه می‌دهد.

فرض کنید که بر روی یک پروژهٔ XML کار می‌کنید. این پروژه، یک سند XML را به عنوان String می‌گیرد و شیء که نمایانگر سند است را تحت نام Document بازمی‌گرداند. اگر بدانیم که کتابخانه در آینده تغییر می‌کند و API فعلی آن دیگر معتبر نخواهد بود، می‌توانیم با انجام یک سلسله اعمال از برنامه خود حفاظت کنیم.

واژه جدید

استخراج کننده XML، یک سند XML را می‌گیرد و نمایش شیء آن را باز می‌گرداند.
بهتر است استخراج کننده را توسط یک کلاس بپوشانیم (wrap).

نکته

پوشاننده (wrapper) مانند یک مبدل عمل می‌کند. پوشاننده رابط عمومی یک شیء را به صورت دیگری تبدیل می‌کند. عموماً پوشاننده وقتی به کار می‌رود که بخواهیم یک رابط مطابق استاندارد مطلوب برنامه خودمان عمل کند.

از آنجایی که می‌توانیم API پوشاننده را کنترل کنیم، لذا استفاده از کتابخانه مزبور تضمین شده خواهد بود. لیست ۱۲ ـ ۱ یک پوشاننده نمونه پیشنهادی را نشان می‌دهد.

لیست ۱۲-۱ Parser.java

```
public interface Parser {
    public org.w3c.dom.Document parser( String document )
```

نکته

org.w3c.dom.Document رابطی است که توسط w3c برای نمایش اسناد XML به صورت ساختاری از اشیاء تدوین شده است. برای اطلاعات بیشتر به http://www.w3c.org مراجعه کنید.

لیست‌های ۱۲ ـ ۲ و ۱۲ ـ ۳ دو پیاده‌سازی پیشنهادی را نشان می‌دهند، که هریک نسخه متفاوتی را به کار می‌گیرند.

لیست ۱۲-۲ VersionOneParser.java

```
public class VersionOneParser implements Parser {

    public org.w3c.dom.Document parse( String document ) {
        // instantiate the version 1 parser
        // XMLParser p = new XMLParser();
        // pass the document to the parser and return the result
        // return p.parseXML( document );
    }
}
```

لیست ۱۲-۳ VersionTwoParser.java

```
public class VersionTwoParser implements Parser {
    public org.w3c.dom.Document parse( String document ) {
        // instantiate the version 2 parser
```

لیست ۱۲-۳ VersionTwoParser.java

```
        // DOMParser parser = new DOMParser();
        // pass the document to the parser and return the result
        // return parser.parse( document );
    }
}
```

برنامه می‌تواند از الگوی مجرد عامل برای ایجاد نسخه مناسب استخراج کننده استفاده کند. لیست ۱۲ ـ ۴ رابط عامل پایه را نشان می‌دهد.

لیست ۱۲-٤ ParserFactory.java

```
public class TwParserFactory {
    public Parser createParser();
}
```

ما به دو پیاده‌سازی عامل نیاز داریم، زیرا دو پیاده‌سازی مختلف استخراج کننده موجود است. لیست‌های ۱۲ـ۵ و ۱۲ ـ ۶ این پیاده‌سازی‌ها را نشان می‌دهند.

لیست ۱۲-۵ VersionOneParserFactory.java

```
public class VersionOneParserFactory implements ParserFactory {
    public Parser createParser() {
        return new VersionOneParser();
    }
}
```

لیست ۱۲-۶ VersionTwoParserFactory.java

```
public class VersionTwoParserFactory implements ParserFactory {
    public Parser createParser() {
        return new VersionTwoParser();
    }
}
```

حال به جای استفاده مستقیم از متغیرهای استخراج کننده، برنامه می‌تواند عامل مناسب را مورد استفاده قرار دهد. لذا در برنامه فقط احتیاج به استفاده از متغیر عامل دارد.

الگوی روال عامل وابستگی زیادی به الگوی مجرد عامل دارد. در واقع یک عامل مجرد می‌تواند روال عامل را برای اشیایی که برمی‌گرداند مورد استفاده قرار دهد.

روال عامل، روالی است که شیء ایجاد می‌کند، مانند createParser() و نمونه‌هایی که در Java API یافت می‌شوند. مانند Class.newInstance().

چنانکه می‌بینید روال عامل می‌تواند هم در کلاسهای معمولی و هم در عامل مجرد ظاهر شود.

الگوی مجرد عامل را چه زمانی به‌کار ببریم؟

وقتی که می‌خواهیم

- چگونگی ایجاد یک شیء را مخفی کنیم
- کلاس اصلی یک شیء را مخفی کنیم
- گروهی از اشیاء را با هم مورد استفاده قرار دهیم. در این صورت در برابر استفاده نامناسب مصونیت داریم.
- بتوانیم از نسخه‌های مختلف پیاده‌سازی استفاده کنیم.

جدول ۱۲ ـ ۱ کار الگوی عامل مجرد را شرح می‌دهد.

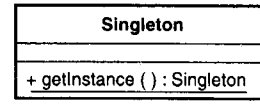
جدول ۱۲-۱ الگوی عامل مجرد

نام الگو	عامل مجرد
مسأله	نیاز به راهی برای جابجایی اشیاء اتصال‌پذیر داریم.
راه حل	ایجاد رابط مجرد حاوی روالهایی برای تعریف اشیاء جدید.
نتایج	می‌توان کلاسهای جدید را به سیستم وارد و خارج کرد.

الگوی تک‌برگ

گاهی به کلاسهایی برمی‌خوریم که لزوماً باید یک و فقط یک متغیر از آنها تعریف شود. مثلاً یک عامل، یا شیء که به یک منبع غیرقابل اشتراک دسترسی دارد. اما هیچ چیزی نمی‌تواند شیء را از تعریف متغیر دیگری از آن نوع بازدارد. پس چه می‌شود کرد؟

الگوی تک‌برگ (Singletone) پاسخی به این پرسش است. الگوی تک‌برگ، با گرفتن وظیفهٔ ایجاد و قطع دسترسی به متغیر در خود شیء، طرح رامحدود می‌کند. چنین کاری، تضمین می‌کند که تنها یک متغیر ایجاد شود و نیز دسترسی به آن را نیز منفرد نگاه می‌دارد. شکل ۱۲ ـ ۲ بخشی از یک کلاس تک‌برگ را نشان می‌دهد.



شکل ۱۲-۲
الگوی تک‌برگ

لیست ۱۲ ـ ۷ یک کلاس تک‌برگ نمونه را نشان می‌دهد.

لیست ۱۲-۷ پیاده‌سازی الگوی تک‌برگ

```
public class Singleton {

    // a class reference to the singleton instance
    private static Singleton instance;

    // the constructor must be hidden so that objects cannot instantiate
    // protected allows other classes to inherit from Singleton
    protected Singleton() {}

    // a class method used to retrieve the singleton instance
    public static Singleton getInstance() {
        if( instance == null ) {
            instance = new Singleton();
```

لیست ۱۲-۷ (ادامه)

```
        }
        return instance;
    }
}
```

کلاس Singleton، یک متغیر static از نوع Singleton دارد، که دسترسی به آن را فقط به روال getInstance() محدود کرده است.

پیاده‌سازی یک تک‌برگ

در فصل ۷، کلاس Payroll را معرفی کردیم. یک سیستم پرداخت حقوق واقعی، احتمالاً دارای یک پایگاه داده از کارمندان می‌باشد. شاید فکر خوبی باشد که بتوان فقط یک متغیر Payroll داشت تا از تداخل منابع جلوگیری شود.

لیست ۱۲ ـ ۸ کلاس تک‌برگ Payroll را نشان می‌دهد.

لیست ۱۲-۸ تک‌برگ Payroll

```
public class Payroll {
    // a class reference to the single singleton instance
    private static Payroll instance;

    private int    total_hours;
    private int    total_sales;
    private double total_pay;

    // hide the constructor so that other objects cannot instantiate
    protected Payroll() {}

    // note the use of static, you don't have an instance when you go to retrieve
    // an instance, so the method must be a class method, thus static
    public static Payroll getInstance() {
        if( instance == null ) {
            instance = new Payroll();
        }
        return instance;
    }
    public void payEmployees( Employee [] emps ) {
        for( int i = 0; i < emps.length; i ++ ) {
            Employee emp = emps[i];
            total_pay += emp.calculatePay();
            emp.printPaycheck();
        }
    }

    public void calculateBonus( Employee [] emps ) {
        for( int i = 0; i < emps.length; i ++ ) {
            Employee emp = emps[i];
System.out.println("Pay bonus to " + emp.getLastName() + ", " + emp.getFirstName() + " $"
+ emp.calculateBonus() );
```

لیست ۱۲-۸ (ادامه)

```
        }
    }

    public void recordEmployeeInfo( CommissionedEmployee emp ) {
        total_sales += emp.getSales();
    }

    public void recordEmployeeInfo( HourlyEmployee emp ) {
        total_hours += emp.getHours();
    }

    public void printReport() {
        System.out.println( "Payroll Report:" );
        System.out.println( "Total Hours: " + total_hours );
        System.out.println( "Total Sales: " + total_sales );
        System.out.println( "Total Paid: $" + total_pay );
    }
}
```

یک روال getInstacne() دارد. این روال مسئول ایجاد و دسترسی به متغیر Singleton است. توجه کنید که تابع سازنده حفاظت شده است، لذا اشیاء دیگر نمی‌توانند متغیر Payroll تعریف کنند. اما از Payroll جدیدی از اشیاء دیگر می‌توانند آن را ارث ببرند.

لیست ۱۲ ـ ۹ مثالی از کاربرد تک‌برگ را نشان می‌دهد.

لیست ۱۲-۹ کاربرد تک‌برگ Payroll

```
//retrieve the payroll singleton
Payroll payroll = Payroll.getInstance();

//create and update some employees
CommissionedEmployee emp1 = new CommissionedEmployee( "Mr.", "Sales", 25000.00, 1000.00);
CommissionedEmployee emp2 = new CommissionedEmployee( "Ms.", "Sales", 25000.00, 1000.00);
emp1.addSales( 7 );
emp2.addSales( 5 );

HourlyEmployee emp3 = new HourlyEmployee( "Mr.", "Minimum Wage", 6.50 );
HourlyEmployee emp4 = new HourlyEmployee( "Ms.", "Minimum Wage", 6.50 );
emp3.addHours( 40 );
emp4.addHours( 46 );

//use the overloaded methods
payroll.recordEmployeeInfo( emp2 );
payroll.recordEmployeeInfo( emp1 );
payroll.recordEmployeeInfo( emp3 );
payroll.recordEmployeeInfo( emp4 );
```

به محض اینکه متغیر تک‌برگ تعریف شد، دقیقاً مانند هر متغیر دیگری قابل استفاده خواهد بود. در لیست ۱۲ ـ ۹ نکتهٔ جالبی وجود دارد:

```
Payroll peyroll = Payroll.getInstacne();
```

در حالی که قبلاً داشتیم:

```
Payroll payroll = new Payroll();
```

وراثت و الگوی تک‌برگ

الگوی تک‌برگ دشواریهایی در وراثت پدید می‌آورد. مخصوصاً اینکه متغیر تک‌برگ را کدام شیء کنترل می‌کند. والد یا فرزند؟

راه اول ایجاد زیرکلاس و به روزآوری والد برای تعریف مورد از فرزند است. لیست‌های ۱۲ ـ ۱۰ و ۱۲ـ۱۱ این راه حل را نشان می‌دهند.

لیست ۱۲-۱۰ تک‌برگ ChildSingletone

```
public class ChildSingleton extends Singleton {

    protected ChildSingleton() { }

    public String toString() {
        return "I am the child singleton";
    }

}
```

لیست ۱۲-۱۱ Singletone به روز شده

```
public class Singlton {

    // a class reference to the singleton instance
    private static Singleton instance;

    // the constructor must be hidden so that objects cannot instantiate
    // protected allows other classes to inherit from Singleton
    protected ParentSingleton() { }

   // a class method used to retrieve the singleton instance
   public static Singleton getInstance() {
       if( instance == null ) {
           instance = new ChildSingleton();
       }
       return instance;
   }


   public String toString() {
       return "I am the singleton";
   }
}
```

مشکل این راه حل، احتیاج به ایجاد تغییر در کلاس والد است. راه حل جانشین عبارت است از اینکه کلاس

تک‌برگ اولین باری که getInstance() فراخوانی می‌شود، یک متغیر حاوی اطلاعات پیکربندی را بخواند. کلاس تک‌برگ می‌تواند از شیء که با این مقدار مشخص می‌شود، متغیر تعریف کند.

همچنین می‌توانید اجازه دهید هر کلاسی پیاده‌سازی خاص خود از getInstance را مورد استفاده قرار دهد. با این راه حل دیگر نیازی به تغییر والد نخواهد بود.

الگوی تک‌برگ چه وقت استفاده می‌شود؟

وقتی که بخواهیم در برنامه از یک کلاس خاص، تنها یک متغیر داشته باشیم. جدول ۱۲ ـ ۲ کاربرد الگوی تک‌برگ را بیان می‌کند.

جدول ۱۲-۲ الگوی تک برگ

نام الگو	تک برگ
مسأله	تنها یک متغیر از کلاس بتواند در برنامه مورد استفاده قرار گیرد.
راه حل	قادر کردن شی به مدیریت ایجاد و دسترسی از طریق یک روال در کلاس
نتایج	دسترسی کنترل شده به متغیر از شی. امکان دسترسی به گروهی از اشیاء هم موجود است. مشتق کردن از هر کلاس تک‌برگ کمی مشکل‌تر است.

الگوی شمارش با نوع محافظت شده

لیست ۱۲ ـ ۱۲ حاوی بخشیء از کلاس Card از فصل ۳ است.

لیست ۱۲-۱۲ بخشی از Card.java

```
public class Card {

    private int rank;
    private int suit;
    private boolean face_up;

    // constants used to instantiate
    // suits
    public static final int DIAMONDS = 4;
    public static final int HEARTS   = 3;
    public static final int SPADES   = 6;
    public static final int CLUBS    = 5;
    // values
    public static final int TWO   = 2;
    public static final int THREE = 3;
    public static final int FOUR  = 4;
    public static final int FIVE  = 5;
    public static final int SIX   = 6;
    public static final int SEVEN = 7;
    public static final int EIGHT = 8;
    public static final int NINE  = 9;
    public static final int TEN   = 10;
    public static final int JACK  = 74;
```

لیست ۱۲-۱۲ (ادامه)

```
    public static final int QUEEN = 81;
    public static final int KING  = 75;
    public static final int ACE   = 65;

    // creates a new card - only use the constants to initialize
    public Card( int suit, int rank ) {
        // In a real program you would need to do validation on the arguments.

        this.suit = suit;
        this.rank  = rank;
    }
}
```

می‌توان دید که هنگام تعریف متغیر از نوع Card باید ثوابت شماره و خال ورق را به تابع سازنده پاس کرد.استفاده از ثوابت به این صورت می‌تواند باعث بروز مشکلات فراوانی شود. درواقع به راحتی می‌توان هر مقدار صحیحی را به سازنده پاس کرد و از آنجایی که باید فقط به نام ثوابت ارجاع داده شود، امکان لو رفتن پیاده‌سازی وجود دارد. مثلاً برای پیدا کردن خال ورق باید مقدار صحیح را بگیرید و آن را با ثوابت مقایسه کنید. این راه حل با وجود صحیح بودن، اصلاً مناسب نیست.

اما مشکل از آنجایی آغاز می‌شود که خال و شماره در ذات خود شیء هستند. عدد صحیح ذات خود را به خاطر اینکه شما می‌خواهید از آن تعبیر خاصی کنید، تغییر نمی‌دهد. در این صورت وظایف روالها پیچیده می‌شود، زیرا در هر موردی که پای خال و شماره به میان بیاید شما باید معنی دلخواه خود را تحمیل کنید.

برخی زبان‌ها مانند C++ دارای ساختار داده‌ای به نام نوع شمارشی (Enumeration) هستند که از این پس آنها را شمارشی خواهیم خواند. شمارشیها درواقع فهرستی از ثوابت صحیح تعریف می‌کنند. اما این ثوابت محدود به شرایطی هستند. مثلا نمی‌توانند رفتار خاصی در پیش بگیرند، یا افزودن ثوابت جدید به مجموعه آنها دشوار است.

با تمام این اوصاف، الگوی شمارشی راهی شیءگرا برای اعلام و تعریف ثوابت فراهم می‌کند. به جای تعریف ثوابت معمولی صحیح، برای هر نوع ثابت کلاسی تعریف می‌کنیم.

پیاده‌سازی الگوی شمارشی نوع‌دار

بگذارید ابتدا نگاهی به پیاده‌سازی Rank (شماره ورق) و Suit (خال ورق) در لیستهای ۱۲-۱۳ و ۱۲-۱۴ بیاندازیم.

لیست ۱۲-۱۳ Suit.java

```
public final class Suit {

    // statically define all valid values of Suit
    public static final Suit DIAMONDS = new Suit( (char)4 );
    public static final Suit HEARTS   = new Suit( (char)3 );
    public static final Suit SPADES   = new Suit( (char)6 );
    public static final Suit CLUBS    = new Suit( (char)5 );

    // helps to iterate over enum values
```

لیست ۱۲-۱۳ (ادامه)

```
    public static final Suit [] SUIT = { DIAMONDS, HEARTS, SPADES, CLUBS };

    // instance variable for holding onto display value
    private final char display;

    // do not allow instantiation by outside objects
    private Suit( char display ) {
        this.display = display;
    }

    // return the Suit's value
    public String toString() {
        return String.valueOf( display );
    }
}
```

Suit ساده است. تابع سازنده یک کاراکتر می‌گیرد و خال را نمایش می‌دهد. از آنجایی که Suit خود یک شیء کامل است، می‌تواند دارای روال هم باشد. در اینجا روال toString() تعریف شده. به شمارشی ایمن می‌توانید هر روالی را که صلاح بدانید بیافزایید.

دیده می‌شود که ثابت هم اختصاصی تعریف شده. با این کار نمی‌توان مستقیماً از Suit متغیر تعریف کرد. در عوض دسترسی تنها به ثوابت تعریف شده کلاس ممکن خواهد بود. علاوه بر این کلاس نهایی تعریف شده، لذا نمی‌توان از آن کلاسی مشتق کرد.

کلاس Suit گروهی از ثوابت تعریف می‌کند. یعنی برای هر خال معتبر یک ثابت. وقتی بخواهیم از ثوابت استفاده کنیم تنها کافی است بنویسیم، مثلاً

```
Suit.DIAMONDS
```

لیست ۱۲-۱٤ Rank.java

```
    public static final Rank TWO   = new Rank( 2, "2" );
    public static final Rank THREE = new Rank( 3, "3" );
    public static final Rank FOUR  = new Rank( 4, "4" );
    public static final Rank FIVE  = new Rank( 5, "5" );
    public static final Rank SIX   = new Rank( 6, "6" );
    public static final Rank SEVEN = new Rank( 7, "7" );
    public static final Rank EIGHT = new Rank( 8, "8" );
    public static final Rank NINE  = new Rank( 9, "9" );
    public static final Rank TEN   = new Rank( 10, "10" );
    public static final Rank JACK  = new Rank( 11, "J" );
    public static final Rank QUEEN = new Rank( 12, "Q" );
    public static final Rank KING  = new Rank( 13, "K" );
    public static final Rank ACE   = new Rank( 14, "A" );

    public static final Rank [] RANK =
          { TWO, THREE, FOUR, FIVE, SIX, SEVEN,
            EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE };
```

لیست ۱۲-۱۴ (ادامه)

```
    private final int    rank;
    private final String display;

    private Rank( int rank, String display ) {
        this.rank = rank;
        this.display = display;
    }

    public int getRank() {
        return rank;
    }
    public String toString() {
        return display;
    }
}
```

کلاس را Rank در لیست ۱۲ ـ ۱۴ مشابه Suit عمل می‌کند، با این تفاوت که روالهای بیشتری دارد. روال getRank() مقدار Rank برمی‌گرداند. این مقدار می‌تواند برای محاسبه ارزش دست ورق مفید باشد. برخلاف ثوابت قبلی، دیگر احتیاجی به مراقبت در اعمال تفسیر خاص خودتان از ثوابت ندارید. بلکه از هر آنجایی که Suit و Rank دو شیء هستند، خود دارای معنی خاص می‌باشند. مثلاً دیگر نیازی نیست هر بار مراقب باشید که معنی عدد صحیح ۴ همان DIAMONDS است. هرگاه هم نیاز به مقایسه مقادیر ثوابت داشتید، می‌توانید از روال مقایسه اشیاء equals() استفاده کنید.

لیست ۱۲ ـ ۱۵ تغییرات لازم در Card برای ایجاد امکان استفاده از ثوابت جدید را نشان می‌دهد.

لیست ۱۲-۱۵ Card.java به روز شده

```
public class Card {

    private Rank rank;
    private Suit suit;
    private boolean face_up;

    // creates a new card - only use the constants to initialize
    public Card( Suit suit, Rank rank ) {
        // In a real program you would need to do validation on the arguments.

        this.suit = suit;
        this.rank = rank;
    }

    public Suit getSuit() {
        return suit;
    }

    public Rank getRank() {
        return rank;
    }
```

لیست ۱۲-۱۵ (ادامه)

```
    public void faceUp() {
        face_up = true;
    }

    public void faceDown() {
        face_up = false;
    }

    public boolean isFaceUp() {
        return face_up;
    }

    public String display() {
        return rank.toString() + suit.toString();
    }
}
```

شاید توجه کرده باشید که Suit و Rank هر دو آرایه‌هایی از ثوابت تعریف می‌کنند. این کار استفاده از حلقه‌ها را آسانتر می‌کند. لیست ۱۲ـ۱۶ را مشاهده کنید. استفاده از حلقه کد روال buidCards() را بسیار آسانتر کرده.

لیست ۱۲-۱۶ buidCards.java به روز شده

```
private void buildCards() {
    deck = new java.util.LinkedList();

    for( int i = 0; i < Suit.SUIT.length; i ++ ) {
        for( int j = 0; j < Rank.RANK.length; j ++ ) {
            deck.add( new Card( Suit.SUIT[i], Rank.RANK[j] ) );
        }
    }
}
```

کاربرد الگوی شمارشی با نوع محافظت شده

وقتی از این الگو استفاده کنید که متوجه شوید مجبور هستید:

- ثوابت متعددی تعریف کنید.
- معانی خاصی برای مقادیر متغیرها متصور شوید.

جدول ۱۲-۳ الگوی شمارشی با نوع محافظت شده

نام الگو	شمارشی با نوع محافظت شده
مسأله	ثوابت صحیح دچار محدودیت هستند.
راه حل	برای هر نوع ثوابت کلاسی ایجاد کنید و مقادیر را از آن بگیرید.
نتایج	ثوابت شیءگرای بسط‌پذیر. ثوابتی کارآمد که دارای رفتار هستند. برای افزودن ثوابت جدید

دامهای الگوها

الگوها اگر درست به کار گرفته نشوند، می‌توانند مانند هر ابزار دیگری گمراه کننده باشند. الگوها طراحی خوب را تضمین نمی‌کنند. درواقع الگوها اگر در غیر جای خود به کار گرفته شوند، می‌توانند مخرب باشند. لذا باید در استفاده از آنها بسیار دقت کنید.

تذکر

چند راهکار برای جلوگیری از افتادن به چنین دامهایی وجود دارد:

۱. فرو کردن مهره‌های استوانه‌ای در سوراخهای مکعب شکل! اگر متوجه شدید که مشغول جستجو برای یافتن جایگاهی برای استفاده از الگوی خاصی هستید، مطمئن باشید که دچار مشکل شده‌اید! به جای آن بهتر است فکر کنید: «من قبلاً هم همین طرح را دیده‌ام، شاید الگوی خاصی بتواند مشکل را حل کند.» آن وقت به دنبال الگوی مناسب بگردید. همواره از دید صورت مسأله به قضیه نگاه کنید، به دنبال یافتن مسأله برای راه حل (الگو) نباشید.

۲. حملهٔ فراموشی! اگر متوجه شدید نمی‌توانید حداکثر در دو جمله بگویید چرا از الگوی خاصی استفاده کرده‌اید، بدانید که دچار دردسر شده‌اید!

اخیراً الگوهای طراحی دچار رگبار فرایندهای از انتقادات شده‌اند. بدبختانه تمایل غریبی، به خصوص بین تازه کارها، وجود دارد که در هر جایی که ممکن باشد از الگوها استفاده کنند.

توجه داشته باشید مسابقه در تعداد الگوهای به کار رفته نیست. مسابقهٔ واقعی در طراحی هماهنگ و صحیح است و چنین طرحی می‌تواند اصلاً از الگوها استفاده نکرده باشد.

در رابطه با الگوها دام عمیق‌تری هم وجود دارد. یاوه‌گویی! هیچگاه برای اینکه باهوش یا زرنگ به نظر برسید از الگوها استفاده نکنید. حتی در گفتگو هم الگویی که مطالعه نکرده‌اید را پیشنهاد نکنید، چون رسوا خواهید شد. اصولی که ما را به الگوها کشانده‌اند را فراموش نکنید و از آنها تخطی نکنید.

خلاصه

الگوها در طرح راه حل‌ها ابزار بسیار مفیدی هستند. به یک نگاه خاص به مسأله، الگوها عصارهٔ آگاهی جامعه شیءگرا و حاصل سالها تجربهٔ طراحی هستند.

در هنگام استفاده از الگوها، همواره محدودیتهای آنها را به یاد داشته باشید. یک الگوی طراحی، یک و تنها یک مسأله مجرد خاص را حل می‌کند. الگوی طراحی راه حل یک مسأله خاص نیست، بلکه راه حلی مجرد برای یک مسأله عام است. استفاده صحیح از این الگو در مسأله خاص، هنر خود شماست. مهارت در این کار تنها در طی زمان و با مطالعه و تمرین حاصل خواهد شد.

پرسشها و پاسخها

چگونه یک الگوی طراحی را برمی‌گزینید؟

هر الگوی طراحی وابسته به یک مسأله و الگوهای وابسته آن است. اگر شرح مسأله با مسأله مزبور شباهت دارد به الگو بپردازید. مطالعه الگوهای وابسته هم مفید خواهد بود. قبل از استفاده از الگو در مسأله مورد نظر، حتماً به نتایج این کار

توجه داشته باشید. اگر نتایج با نیازمندیهای شما تضاد داشت، احتمالاً باید از الگو صرفنظر کنید.

چه وقت یک الگوی طراحی را مورد استفاده قرار می‌دهیم؟

این پرسش پاسخ ساده‌ای ندارد. اگر الگویی را نشناسید، نمی‌توانید از آن استفاده کنید. هیچ کس تمام الگوهای موجود را نمی‌شناسد. در زمان طراحی سعی کنید تا جای ممکن اطلاعات جمع‌آوری کنید. از دیگران بپرسید. مطالعه کنید. دربارهٔ الگوهای موجود پرس‌وجو کنید. هرچه الگوهای بیشتری را بشناسید، فرصت استفاده بیشتری خواهید داشت.

کارگاه

پرسشها

۱. **کلاس پوشاننده چیست؟**
۲. **الگوی عامل مجرد چه مسأله‌ای را حل می‌کند؟ چرا از آن استفاده می‌کنید؟**
۳. **الگوی تک‌برگ چه مسأله‌ای را حل می‌کند؟ چرا از آن استفاده می‌کنید؟**
٤. **الگوی شمارشی با نوع محافظت شده چه مسأله‌ای را حل می‌کند؟ چرا از آن استفاده می‌کنید؟**
۵. **آیا استفاده از الگو صحت و کامل بودن طراحی را تضمین می‌کند؟ چرا؟**

تمرین‌ها

۱. در لیست ۱۲ ـ ۱۷ کد Bank از فصل ۷ آمده. آن را تبدیل به تک‌برگ کنید.

لیست ۱۲-۱۷ Bank.java

```
public class Bank {

    private java.util.Hashtable accounts = new java.util.Hashtable();

    public void addAccount( String name, BankAccount account ) {
        accounts.put( name, account );
    }

    public double totalHoldings() {
        double total = 0.0;

        java.util.Enumeration enum = accounts.elements();
        while( enum.hasMoreElements() ) {
            BankAccount account = (BankAccount) enum.nextElement();
            total += account.getBalance();
        }
        return total;
    }

    public int totalAccounts() {
        return accounts.size();
    }

    public void deposit( String name, double ammount ) {
```

لیست ۱۲-۱۷ (ادامه)

```
        BankAccount account = retrieveAccount( name );
        if( account != null ) {
            account.depositFunds( ammount );
        }
    }

    public double balance( String name ) {
        BankAccount account = retrieveAccount( name );
        if( account != null ) {
            return account.getBalance();
        }
        return 0.0;
    }

    private BankAccount retrieveAccount( String name ) {
        return (BankAccount) accounts.get( name );
    }
}
```

۲. کلاس Error لیست ۱۲ ـ ۱۸ را در نظر بگیرید. آن را با استفاده از شمارشی نوع‌دار تغییر دهید.

لیست ۱۲-۱۸ Error.java

```
public class Error {

    // error levels
    public final static int NOISE   = 0;
    public final static int INFO    = 1;
    public final static int WARNING = 2;
    public final static int ERROR   = 3;

    private int level;

    public Error( int level ) {
        this.level = level;
    }

    public int getLevel() {
        return level;
    }

    public String toString() {
        switch (level) {
            case 0: return "NOISE";
            case 1: return "INFO";
            case 2: return "WARNING";
            default: return "ERROR";
        }
    }
}
```

۳. یک عامل مجرد برای سلسله مراتب BankAccount طرح کرده و ایجاد کنید. (فصل ۷، آزمایشگاه ۳)

روز ۱۳

Object Oriented Programming

شیءگرایی و برنامه‌نویسی رابط کاربر

رابط کاربر (UI) دریچه‌ای است برای ورود و تعامل کاربر با سیستم. همه انواع سیستمهای مدرن به نحوی رابط کاربر را ارایه می‌کنند، به صورت گرافیکی، خط فرمان و یا حتی بر اساس گفتار (تشخیص گفتار) و بعضی از آنها همه انواع ذکر شده را با هم به کار می‌برند! در هر صورت، در طراحی و پیاده‌سازی رابط کاربر باید دقت به خرج دهید. خوشبختانه OOP مزایای بسیاری را برای UI در اختیار شما قرار می‌دهد.

آنچه امروز خواهید آموخت

- چگونه OOP و UI با یکدیگر پیوند می‌خورند.
- درباره اهمیت رابطهای منقطع شده از هم مطالبی را فراخواهید گرفت.
- چه الگوهایی به شما در منقطع کردن اجزای رابط کاربر کمک می‌کنند.

شیءگرایی و رابط کاربر

فرایند طراحی و پیاده‌سازی رابط کاربر چیزی فرای طراحی و برنامه‌نویسی دیگر اجزای سیستم نیست. تنها تفاوت آن است که با مجموعه‌ای از دستورات، توابع و APIهای جدید برای نوشتن عناصر مرتبط با رابط کاربر آشنا شوید، با این حال در انتها باز هم به تکنیکهای شیءگرایی محتاج خواهید بود.

نکته

خواهید آموخت که توسعه و نوشتن رابطهای کاربر را با دید یک برنامه‌نویس انجام دهید. به عنوان یک برنامه‌نویس کلاسهایی را برای پیاده‌سازی و پشتیبانی رابط کاربر طراحی و پیاده‌سازی خواهید کرد.

درس امروز مبحث کلی طراحی رابط کاربر را دنبال نمی‌کند. طراحی رابط کاربر خود تکنیکی برای ارایه توانمندیهای برنامه به کاربر است. بحث کلی طراحی کاربر که چیزی کاملاً جدای از برنامه‌نویسی است، ریشه در هنرهای ترسیمی و گرافیکی دارد و روانشناسی هم در آن دخالت دارد. گروه ویژه ACM در مورد تعاملات بین انسان و کامپیوتر (SIGCHI) مرجع بسیار خوبی برای دستیابی به اطلاعات مرتبط با طراحی UI است.

به هنگام طراحی و پیاده‌سازی رابط کاربر، باید آنچه را که تاکنون در مورد OO آموخته‌اید به خوبی به کار ببندید. عموماً اجزای مختلف مرتبط با رابط کاربر به خوبی در کنار یکدیگر جمع شده و کار می‌کنند.

کدهای مرتبط با رابط کاربر دقیقاً همانند دیگر کدهای موجود در سیستم باید شیءگرا باشند. پیاده‌سازی رابط کاربر هم از تکنیکهای کپسوله‌سازی، وراثت و پلی مورفیسم (چندشکلی) استفاده می‌کند.

همچنین به هنگام طراحی و تحلیل شیءگرای مسأله (OOD و OOA) باید رابط کاربر را هم مدنظر داشته باشید. بدون یک تحلیل و طراحی درست، نیازمندیهای کاربر را به خوبی تشخیص نداده و رابط کاربری ناپایداری خواهید نوشت.

اهمیت رابطهای کاربری منقطع از هم

همانگونه که ممکن است تاکنون دیده باشید عموماً یک سیستم شامل رابطهای کاربری مختلف و در عین حال غیرمرتبط با یکدیگر است. برای مثال یک سیستم تدارکات مبتنی بر وب به کاربران مختلف اجازه می‌دهد از طریق وب با استفاده از کامپیوترهای دستی قابل حمل (PDA) و یا کامپیوترهای شخصی (PC) سفارشهای خود را ارسال کنند. همانگونه که واضح است، رابط کاربر بر روی این دو سیستم با یکدیگر بسیار متفاوت است. در هر کدام نمایش اطلاعات به نحو خاصی خواهد بود.

همچنین نیازمندیهای مرتبط با رابطهای کاربری عموماً در حال تغییر است. با گذشت زمان و اضافه شدن ویژگیهای جدید به سیستم و پیدا کردن ضعفهای سیستم توسط کاربران، رابطهای کاربری نیز باید بهبود یابند. بنابراین به طور پیوسته رابطهای کاربری را باید تغییر داد. این واقعیت این نکته را گوشزد می‌کند که رابطهای کاربر را باید فوق‌العاده پایدار و قابل اطمینان ساخت.

بهترین کار برای دستیابی به پایداری و اطمینان بالا جداسازی کامل سیستم از رابط کاربری آن است. یک طراحی جدا شده از UI این امکان را می‌دهد تا هرگونه رابط کاربری را به سیستم بتوان اضافه کرد و بتوان تغییرات درخواستی را به سرعت و با حداقل زمان ممکن بر روی سیستم اعمال نمود. همچنین از این طریق می‌توان قابلیتها و کارایی سیستم را آسانتر آزمایش کرد و در نتیجه خطاهای سیستم را راحتتر پیدا کرد.

خوشبختانه OOP جامعترین راه حل برای این مسأله است. با ایزوله کردن صحیح مسئولیتها خواهید توانست حداقل تغییرات را بر روی قسمتهای نامرتبط اعمال کنید. درواقع از این طریق اضافه کردن هر نوع رابطی به سیستم حداقل زمان ممکن را از شما می‌گیرد و البته این تغییرات، تغییرات بنیادی در سیستم را به دنبال نخواهد داشت. نکته کلیدی در اینجا آن است که کدهای مربوط به رابط کاربری داخل کدهای مربوط به سیستم قرار نگیرد. این دو باید از یکدیگر جدا باشند.

اجازه دهید به عنوان نمونه مثال زیر را که به صورتی نادرست اجزای UI را جدا کرده است، نگاهی بیاندازیم. لیست ۱۳ ـ ۱ یک پیاده‌سازی نادرست از رابط کاربری را نشان می‌دهد.

لیست ۱۳-۱ VisualBankAccount.java

```
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.JLabel;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.WindowListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JTextField;
import javax.swing.JButton;

public class VisualBankAccount extends JPanel implements ActionListener {

    public static void main( String [] args ) {
        JFrame frame = new JFrame();

        WindowAdapter wa = new WindowAdapter() {
            public void windowClosing( WindowEvent e ) {
                System.exit( 0 );
            }
        };

        frame.addWindowListener( wa );

        frame.getContentPane().add( new VisualBankAccount( 10000.00 ) );
        frame.pack();
        frame.show();
    }

    // private data
    private double balance;

    // UI elements
    private JLabel     balanceLabel   = new JLabel();
    private JTextField amountField    = new JTextField( 10 );
    private JButton    depositButton  = new JButton( "Deposit" );
    private JButton    withdrawButton = new JButton( "Withdraw" );

    public VisualBankAccount( double initDeposit ) {
      setBalance( initDeposit );
```

لیست ۱۳-۱ (ادامه)

```
    buildUI();
}

public void actionPerformed( ActionEvent e ) {
    if( e.getSource() == depositButton ) {
        double amount = Double.parseDouble( amountField.getText() );
        depositFunds( amount );
    } else if( e.getSource() == withdrawButton ) {
        double amount = Double.parseDouble( amountField.getText() );
        if( amount > getBalance() ) {
            amount = getBalance();
        }
        withdrawFunds( amount );
    }
}

private void buildUI() {

    setLayout( new BorderLayout() );

    // build the display
    JPanel buttons = new JPanel( new BorderLayout() );
    JPanel balance = new JPanel( new BorderLayout() );
    buttons.add( depositButton, BorderLayout.WEST );
    buttons.add( withdrawButton, BorderLayout.EAST );
    balance.add( balanceLabel, BorderLayout.NORTH );
    balance.add( amountField, BorderLayout.SOUTH );
    add( balance, BorderLayout.NORTH );
    add( buttons, BorderLayout.SOUTH );

    // set up the callbacks so that the buttons do something
    // the deposit button should call depositFunds()
     depositButton.addActionListener( this );
    // the withdraw button should call withdrawFunds
     withdrawButton.addActionListener( this );
}

public void depositFunds( double amount ) {
    setBalance( getBalance() + amount );
}

public double getBalance() {
    return balance;
}
protected void setBalance( double newBalance ) {
    balance = newBalance;
     balanceLabel.setText( "Balance: " + balance);
}

public double withdrawFunds( double amount ) {
```

لیست ۱۳-۱ (ادامه)

```
        setBalance( getBalance() - amount );
        return amount;
    }
}
```

VisualBankAccount از کتابخانه Java Swing برای نمایش عناصر گرافیکی استفاده کرده است. هر زبان OO کتابخانه‌ای برای ایجاد و نمایش رابطهای گرافیکی کاربر (GUI) دارد. اگر چیزی از این مثال متوجه نشده‌اید، نگران نباشید! تنها کافی است یک دید کلی از مثال را درک کرده باشید.

کتابخانه Swing برای هر یک از عناصر GUI کلاسی را ارایه می‌دهد. برای مثال JButton کلاسی برای نمایش و کار با یک دکمه (Button) است. JLabel و JTextField به ترتیب برای نمایش برچسب (Label) و متن (Text) می‌باشند. می‌توان عناصر فوق را داخل یک پانل (Panel) قرار داد تا ارتباط آنها با یکدیگر روشنتر شود.

هریک از عناصر GUI شامل متد addActionListener() هستند. این متد اجازه می‌دهد که رخدادهای مربوط به هریک از عناصر از قبیل Callback پاسخ داده شوند. با رخدادن یک اتفاق (مثلاً کلیک) شیء مربوطه پاسخ مناسب را خواهد داد.

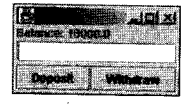
Visual Account همه قابلیتهایی که در کلاس BankAccount در دروس گذشته ارایه شده است را در خود دارد. همچنین نحوه نمایش و پاسخ به رخدادهای ایجاد شده توسط کاربر را هم می‌داند. شکل ۱۳ ـ ۱، این کلاس را به هنگام اجرا نشان می‌دهد. شکل ۱۳ ـ ۱ VisualBankAccount که در داخل یک قاب (Frame) قرار داده شده است.

در صورتی که در قسمت amount تایپ کرده و سپس یکی از دکمه‌های Deposit (واریز) و یا Withdraw (برداشت) را فشار دهید، میزان وجه از فیلد amount استخراج گشته و برحسب مورد یکی از توابع withdrawfunds() و یا depositFunds() فراخوانی می‌گردد.

از آنجا که VisualBankAccount از نوع JPanel است، آن را می‌توان در هر نوع Java GUI، جاسازی کرد. متأسفانه رابط کاربری از کلاس حساب بانکی (BankAccount) جدا نشده است. به همین خاطر نمی‌توان رابط کاربری را عوض کرده و فرم دیگری را برای این منظور به کار برد. برای تغییر دادن رابط کاربری باید مستقیماً کدهای مربوط به حساب بانکی را دستکاری کرد. به عبارت دیگر باید نسخهٔ دیگری از VisualBankAccount را برای ساخت یک UI جدید نوشت.

چگونه با استفاده از الگوی کنترل کننده نمایش، اجزای UI را جدا کنیم

الگوی طراحی کنترل کننده نمایش (MVC) یک روش طراحی رابط کاربری ارایه می‌دهد که کاملاً اجزای UI را از باقی سیستم جدا می‌کند.



شکل ۱۳ـ۱
کلاس VisualBankAccount درون یک فریم

نکته

MVC تنها یک روش طراحی شیءگرای رابط کاربری است و روشهای معتبر دیگری نیز برای طراحی رابط کاربری وجود دارند. با این حال MVC روشی است که در صنعت نرم‌افزار امتحان خود را پس داده است. اگر با طراحی رابط کاربری مخصوص در وب و با استفاده از ابزارهای J2EE شرکت Sun کار کرده باشید، بارها با این روش روبرو شده‌اید.

از طریق الگوی MVC می‌توان رابط کاربری را از باقی سیستم جدا کرد. برای این منظور طراحی رابط کاربری در سه قسمت باید صورت گیرد:

- مدل (Model)، که نمایشی از سیستم است.
- نما (View)، که نمایش مدل است.
- کنترل کننده (Controller) که ورودیهای کاربر را پردازش می‌کند

هریک از اجزای MVC، مسئولیتهای منحصر به خود را دارند.

مدل

مدل موظف است اعمال زیر را فراهم کند:

- دسترسی به قابلیتها و کاربردهای پایه و اساسی سیستم
- دسترسی به اطلاعات وضعیت سیستم
- مکانیزم اطلاع دادن تغییر وضعیت سیستم

درواقع مدل لایه‌ای از MVC است که رفتارهای پایه و وضعیت سیستم را مدیریت می‌کند. مدل باید به پرسشهایی که در مورد وضعیت سیستم هستند و از طریق نما (View) و یا کنترل کننده (Controller) ارایه می‌شوند پاسخگو باشد.

نکته

یک سیستم می‌تواند شامل مدلهای مختلفی باشد. به عنوان مثال یک سیستم بانکی می‌تواند شامل مدلهایی برای حساب بانکی و مشتری باشد. تعداد زیادی مدل کوچک بهتر از یک مدل بزرگ وظایف را بر عهده می‌گیرند. نگذارید عبارت مدل شما راگیج کند. مدل چیزی جز شیئی که سیستم را نشان می‌دهد، نیست.

کنترل کننده لایه‌ای از MVC است که ورودیهای کاربر را تفسیر می‌کند. در جواب ورودی کاربر، کنترل کننده ممکن است به مدل و یا نما دستوراتی بدهد و یا از آنها درخواست کند، عملی را انجام دهند.

نما لایه‌ای از MVC است که نمایش گرافیکی و یا متنی از مدل را ارایه می‌کند. درواقع نما تمام اطلاعات مربوط به وضعیت سیستم را از مدل دریافت می‌کند.

در این صورت، مدل هیچ آگاهی از فراخوانی متدها توسط کنترل کننده و یا نما را ندارد. مدل تنها از این نکته آگاه است که شیء یکی از متدهایش را فراخوانی کرده است. تنها راه ارتباطی مدل با رابط کاربر (UI) از طریق آگاه‌سازی اجزای UI از ایجاد تغییرات در وضعیت سیستم است.

در صورتی که کنترل کننده و یا نما علاقمند به دریافت اطلاعات مربوط به تغییر وضعیت سیستم باشند، باید برای این امر سراغ مدل رفته و رخدادهای درخواستی را اعلام نمایند. زمانی که مدل وضعیت خود را

تغییر می‌دهد، به سراغ فهرست اشیاء ثبت شده (که شنونده‌ها و بیننده‌ها نامیده می‌شوند) رفته و هریک از آنها را از تغییر رخ داده باخبر سازد. عموماً برای ساخت این الگو، مدلها به سراغ الگوی Observer (بیننده و یا رصدکننده) می‌روند.

الگوی رصدکننده

الگوی رصدکننده مکانیزمی برای انتشار/دریافت در اختیار اشیاء می‌گذارد. درواقع این الگو اجازه می‌دهد که شیء (رصدکننده) علاقمندیهایش نسبت به رخدادهای یک شیء دیگر (رصدشونده) را ثبت کند. زمانی که شیء رصدشونده بخواهد رصدکننده را به واسطه تغییری در وضعیتش باخبر کند، متد ()update از آن شیء را فراخوانی می‌کند.

لیست ۱۳ ـ ۲ رابط Observer را ارایه می‌کند. تمام رصدکننده‌هایی که بخواهند خود را برای رخدادهای یک شیء رصدکننده آماده کنند، باید این رابط را پیاده‌سازی نمایند.

لیست ۱۳-۲ Observe.java

```
public interface observer {
    public void update();
}
```

اشیاء رصدشونده متدی را در اختیار رصدکننده‌ها می‌گذارند تا از آن طریق رصدکننده‌ها بتوانند رخدادهای مورد علاقه برای رصد (پیگیری) را اعلام نمایند. لیست ۱۳ ـ ۳ کلاس را که الگوی Observer را پیاده‌سازی کرده است، نشان می‌دهد.

پیاده‌سازی مدل

با اعمال الگوی MVC به VisualBankAccount، کلاس فوق پایداری و اطمینان بیشتری به دست می‌آورد. اجازه دهید با جداکردن قابلیتها و کاربردهای اصلی سیستم از کدهای نمایش، مدلی را ایجاد کنیم.

لیست ۱۳ ـ ۳ قابلیتهای اساسی حساب بانکی ـ مدل ـ را نشان می‌دهد.

لیست ۱۳-۳ BankAccountModel.java

```
import java.util.ArrayList;
import java.util.Iterator;

public class BankAccountModel {
    // private data
    private double    balance;
    private ArrayList listeners = new ArrayList();

    public BankAccountModel( double initDeposit ) {
    setBalance( initDeposit );
    }

    public void depositFunds( double amount ) {
      setBalance( getBalance() + amount );
```

لیست ۱۳-۳ (ادامه)

```
    }

    public double getBalance() {
        return balance;
    }

    protected void setBalance( double newBalance ) {
        balance = newBalance;
        updateObservers();
    }

    public double withdrawFunds( double amount ) {
        if( amount > getBalance() ) {
            amount = getBalance();
        }
        setBalance( getBalance() - amount );
        return amount;
    }

    public void register( Observer o ) {
        listeners.add( o );
        o.update();
    }

    public void deregister( Observer o ) {
        listeners.remove( o );
    }

    private void updateObservers() {
        Iterator i = listeners.iterator();
        while( i.hasNext() ) {
            Observer o = (Observer) i.next();
            o.update();
        }
    }
}
```

کلاس BankAccountModel بسیار شبیه کلاس BankAccount که در دروس گذشته با آن آشنا شدیم، است.تفاوت تنها در استفاده از الگوی Observer برای اضافه کردن قابلیت ثبت درخواستها و به‌روز کردن اشیاء به هنگام ایجاد تغییرات است.

این نکته نیز قابل ذکر است که تمام منطق سیستم در این کلاس نهفته است. withdrawFunds() میزان وجهی که برداشت می‌شود را چک می‌کند تا از میزان موجودی بیشتر نباشد. رعایت یکسری از قوانین حوزه مسأله (Domain Rules) بسیار اهمیت دارد. اگر این قوانین در نما و یا کنترل کننده رخنه کنند، هرکدام

مجبورند که خود اقدام به حفظ و رعایت قوانین کنند که در این صورت از لحاظ مسئولیت‌پذیری و پاسخگویی مشکلاتی برای سیستم به وجود خواهد آمد. همچنین این امر باعث می‌شود که تغییر قوانین بسیار مشکل گردد چراکه در این حالت باید سراغ کدهای مربوط به نما و یا کنترل کننده رفت.

قرار دادن قوانین در داخل نما خطرات دیگری را هم در پی خواهد داشت. از آنجا که نما برای تمامی زیرکلاسها نیز کار خواهد کرد، اگر قانون مربوط به برداشت پول را داخل نما قرار دهیم در این صورت این نما برای کلاس OverdraftAccount که در آن می‌توان بیش از موجودی، از حساب برداشت کرد، کار نخواهد کرد. توجه داشته باشید که طرای نما باید به گونه‌ای باشد که برای همه انواع مدلها کارساز باشد!

نما

نما مسئولیتهای زیر را بر عهده دارد:

- نمایش مدل به کاربر
- معرفی خود به مدل برای دریافت تغییرات رخ داده در وضعیت سیستم
- دریافت اطلاعات وضعیت از مدل

نما لایه‌ای از MVC است که وظیفه‌اش نمایش اطلاعات به کاربر است. نما اطلاعات را با استفاده از رابطهای عمومی مدل دریافت کرده و آنها را نمایش می‌دهد و برای آنکه بتواند تغییرات در وضعیت سیستم را دنبال کرده و آنها را نیز نمایش دهد، باید خود را به مدل معرفی کند.

نکته یک مدل می‌تواند چندین نمای مختلف داشته باشد.

پیاده‌سازی نما

با وجود یک مدل زمان آن رسیده است که نمای مربوط به کلاس حساب بانکی را نیز ایجاد کنیم. لیست ۱۳ـ۴ نمای کلاس BankAccountModel است.

لیست ۱۳ـ۴ BankAccounView.java

```
import javax.swing.JPanel;
import javax.swing.JLabel;
import java.awt.BorderLayout;
import javax.swing.JTextField;
import javax.swing.JButton;

public class BankAccountView extends JPanel implements Observer {

    public final static String DEPOSIT  = "Deposit";
    public final static String WITHDRAW = "Withdraw";
    private BankAccountModel model;
    private BankAccountController controller;

    // GUI Elements, pre-allocate all to avoid null values
```

لیست ۱۳-٤ (ادامه)

```
private JButton depositButton  = new JButton( DEPOSIT );
private JButton withdrawButton = new JButton( WITHDRAW );
private JTextField amountField = new JTextField();
private JLabel balanceLabel    = new JLabel();

public BankAccountView( BankAccountModel model ) {
    this.model = model;
    this.model.register( this );
    attachController( makeController() );
    buildUI();
}

// called by model when the model changes
public void update() {
    balanceLabel.setText( "Balance: " + model.getBalance() );
}

// provides access to the amount entered into the field
public double getAmount() {
    // assume that the user entered a valid number
    return Double.parseDouble( amountField.getText() );
}

// wires the given controller to the view, allows outside object to set controller
public void attachController( BankAccountController controller ) {
    // each view can only have one controller, so remove the old one first
    if( this.controller != null ) { // remove the old controller
        depositButton.removeActionListener( controller );
        withdrawButton.removeActionListener( controller );
    }

    this.controller = controller;
    depositButton.addActionListener( controller );
    withdrawButton.addActionListener( controller );
}

protected BankAccountController makeController() {
    return new BankAccountController( this, model );
}

private void buildUI() {

    setLayout( new BorderLayout() );
```

لیست ۱۳-٤ (ادامه)

```
        // associate each button with a commend string
        depositButton.setActionCommand( DEPOSIT );
        withdrawButton.setActionCommand( WITHDRAW );

        // build the display
        JPanel buttons = new JPanel( new BorderLayout() );
        JPanel balance = new JPanel( new BorderLayout() );
        buttons.add( depositButton, BorderLayout.WEST );
        buttons.add( withdrawButton, BorderLayout.EAST );
        balance.add( balanceLabel, BorderLayout.NORTH );
        balance.add( amountField, BorderLayout.SOUTH );
        add( balance, BorderLayout.NORTH );
        add( buttons, BorderLayout.SOUTH );
    }
}
```

تابع سازنده BankAccountView یک ارجاع از BankAccountModel را به عنوان پارامتر ورودی دریافت می‌کند. به هنگام ایجاد، کلاس BankAccountView رخدادهای مورد نظر را به مدل اعلام می‌کند و سپس خودش را به کنترل کننده متصل کرده و رابط کاربر را می‌سازد. نما از مدل برای دریافت اطلاعاتی که جهت نمایش بدان نیازمند است، استفاده می‌کند. به هنگام تغییری در موجودی، مدل تابع update() از نما را فراخوانی می‌کند. با فراخوانی این تابع، نما موجودی نمایش داده شده را اصلاح می‌کند. به طور کلی نما خودش اقدام به ایجاد کنترل کننده می‌کند، همانگونه که BankAccountView در داخل متد makeController() اقدام به این کار کرده است. زیرکلاسها با جایگزینی این متد می‌توانند کنترل کننده خاص خود را ایجاد نمایند. با فراخوانی تابع attachController() نما کنترل کننده را برای کنترل و بالتبع دریافت پیغام از دکمه‌های deposit و withdraw آماده می‌کند.

این نکته را مدنظر داشته باشید که نما ابتدا کنترل کننده‌های از پیش موجود را حذف می‌کند تا در هر لحظه تنها یک کنترل کننده وجود داشته باشد. همینطور توجه داشته باشید که attachController() یک متد عمومی (public) است. با استفاده از این متد می‌توان کنترل کننده را بدون ایجاد یک زیرکلاس از نما، تغییر داد. کنترل کننده‌ای که کد آن را ملاحظه خواهید کرد به همان شیوه‌ای عمل می‌کند که VisualBankAccount عمل می‌کرد (تنها تفاوت اندکی وجود دارد).

برخلاف نما که می‌تواند در هر لحظه یک کنترل کننده داشته باشد، یک مدل می‌تواند تعداد زیادی نمای مختلف داشته باشد. لیست ۱۳ - ۵ نمای دومی را برای حساب بانکی نشان می‌دهد.

لیست ۱۳-۵ BankAccountCLV.java

```
public class BankAccountCLV implements Observer {

    private BankAccountModel model;

    public BankAccountCLV( BankAccountModel model ) {
```

```
        this.model = model;
        this.model.register( this );
    }

    public void update() {
        System.out.println( "Current Balance: $" + model.getBalance() );
    }
}
```

کلاس BankAccountCLV تنها اقدام به چاپ موجودی در خط فرمان می‌کند. اگرچه این کار رفتار ساده‌ای را نشان می‌دهد ولی نمایانگر نمای دیگری برای BankAccountModel است. آنگونه که مشاهده می‌کنید نمای فوق نیازی به کنترل کننده ندارد، چرا که رخدادی از طرف کاربر را نمی‌پذیرد. بنابراین در همه مواقع نیازی به کنترل کننده برای یک نما نیست.

نکته

یک نما همیشه نیاز به نمایش بر روی صفحه نمایشگر ندارد. برای مثال یک واژه‌پرداز را در نظر بگیرید. مدل موجود در واژه‌پرداز باید متن وارد شده را دنبال کند، آن را قالب‌بندی کرده و پاورقی و... را به متن بیافزاید. یک نما جهت نمایش متن بر روی صفحه نمایشگر به کار می‌رود. حال آن که می‌توان از نماهای دیگر برای تبدیل متن به فرمتهای دیگر نظیر PDF و یا HTML و یا حتی Post Script نیز استفاده کرد. درواقع نمایی که جهت نوشتن اطلاعات درون یک فایل به کار می‌رود نیازی به نمایش بر روی صفحه نمایشگر ندارد. دیگر برنامه‌ها می‌توانند فایل ایجاد شده را باز کرده و در صورت نیاز داده‌های موجود در آن را نمایش دهند.

کنترل کننده

کنترل کننده وظایف زیر را بر عهده دارد:

- جداکردن رخدادها (events) از نما
- تفسیر رخدادها و فراخوانی متدهای مناسب از مدل و یا نما
- همبستگی با مدل برای تشخیص تغییر وضعیت در مدل در صورت نیاز

کنترل کننده را می‌توان واسطی بین مدل و نما در نظر گرفت. درواقع کنترل کننده رخدادها را دریافت کرده و آنها را به درخواستهایی که به نماد یا مدل ارسال می‌شوند، تبدیل می‌کند.

نکته

یک نما تنها دارای یک کنترل کننده و یک کنترل کننده هم تنها یک نما دارد. بعضی از نماها اجازه می‌دهند کنترل کنندهٔ آنها را مستقیماً انتخاب کنید.

هر نما تنها یک کنترل کننده دارد و تمام تعاملات کاربر به سمت کنترل کننده روانه می‌شود. در صورتی که کنترل کننده به وضعیت جاری مدل وابسته باشد، باید خود را به مدل معرفی کرده باشد تا در این صورت مدل تغییرات صورت گرفته را به کنترل کننده اعلام کند.

پیاده‌سازی کنترل کننده

با مدل و نمای ایجاد شده تنها ایجاد کنترل کننده BankAccountView باقی مانده است. لیست ۱۳ ـ ۶ کنترل کننده این نما را نشان می‌دهد.

لیست ۱۳-۶ avaj.rellortnoCtnuoccAknaB

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class BankAccountController implements ActionListener {

    private BankAccountView  view;
    private BankAccountModel model;

    public BankAccountController( BankAccountView view, BankAccountModel model ) {
        this.view  = view;
        this.model = model;
    }

    public void actionPerformed( ActionEvent e ) {

        String command = e.getActionCommand();
        double amount = view.getAmount();

        if( command.equals( view.WITHDRAW ) ) {
            model.withdrawFunds( amount );
        } else if( command.equals( view.DEPOSIT ) ) {
            model.depositFunds( amount );
        }

    }
}
```

از طریق تابع سازنده، ارجاعی از مدل و نما به عنوان پارامترهای ورودی دریافت می‌گردد. کنترل کننده از نما جهت دریافت میزان موجودی استفاده می‌کند. همچنین از مدل برای اعمال واریز یا برداشت از حساب استفاده می‌شود.

خود BankAccountController منطق ساده‌ای دارد. درواقع با پیاده‌سازی رابط ActionListener رخدادهای کاربر بر روی نما، دریافت می‌گردد. نما تنها از کنترل کننده برای تفسیر رخدادهای صورت گرفته استفاده می‌کند. با دریافت یک رخداد توسط کنترل کننده، آن را بررسی می‌کند تا متوجه شود منبع رخداد کدامیک از دو حالت برداشت (withdraw) و یا واریز (deposti) بوده است. در هر صورت، متدهای متناظر در مدل فراخوانی می‌گردند. برخلاف کد نوشته شده در VisualBankAccount، کنترل کننده تنها نیاز به دو تابع depostiFunds() و withdrawFunds() دارد. در ضمن هیچ نیازی نیست که کنترل کننده مقدار موجودی را بررسی کند که آیا بیشتر و یا کمتر از میزان برداشتی هست یا خیر، چراکه این کار در مدل صورت می‌گیرد.

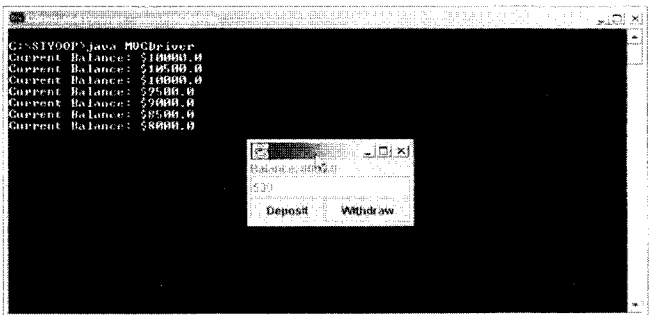
کنار هم قرار دادن مدل، نما و کنترل کننده

لیست ۱۳ ـ ۷ برنامه‌ای را نشان می‌دهد که یک مدل و دو نما را به یکدیگر متصل کرده است.

لیست ۱۳-۷ مدل، نما و کمترل کننده در کنار هم

```
import java.awt.event.WindowListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;

public class MVCDriver {

    public static void main( String [] args ) {

        BankAccountModel model = new BankAccountModel( 10000.00 );
        BankAccountView view   = new BankAccountView( model );
        BankAccountCLV clv = new BankAccountCLV( model );

        JFrame frame = new JFrame();

        WindowAdapter wa = new WindowAdapter() {
            public void windowClosing( WindowEvent e ) {
                System.exit( 0 );
            }
        };

        frame.addWindowListener( wa );

        frame.getContentPane().add( view );
        frame.pack();
        frame.show();
    }
}
```

در ابتدا نمونه‌ای از کلاس مربوط به مدل ایجاد می‌شود. با ایجاد مدل نوبت به ایجاد نماها می‌رسد. در مورد کلاس BankAccountView، باید نما را داخل یک قابل (frame) قرار داد تا امکان نمایش نما فراهم شود. شکل ۱۳ ـ ۲ نتیجه حاصل شده را نشان می‌دهد.



شکل ۱۳ ـ ۲
مدل حساب بانکی به همراه‌چند نوع مختلف نما

با اجرای کلاس MVCDriver خواهید دید که دو نمای جدا از هم برای یک مدل کار می‌کنند. با استفاده از الگوی MVC می‌توان نماهای بسیاری را برای مدل مورد نیاز خود تهیه کرد.

مشکلات مرتبط با الگوی MVC

مثل همه طراحیها، الگوی MVC شامل مزایا و معایبی است. مشکلات مرتبط با این الگو عبارتند از:

- تأکید بر روی داده‌ها
- اتصال محکم بین نماکنترل کننده با مدل
- امکان ناکارآمدی سیستم

ایجاد مشکلات فوق بستگی به مسأله‌ای دارد که با آن روبرو هستید.

تأکید بر روی داده‌ها

با توجه به میزان خلوص OO، الگوی MVC به دلیل تأکید بسیار بر روی داده‌ها جایگاه ممتازی ندارد. به جای آنکه نما از شیء بخواهد تا کاری را انجام دهد، از مدل در مورد داده‌ها سؤال کرده و سپس آن را نمایش می‌دهد.

برای کم کردن این مشکل بهتر است تنها به نمایش داده‌های موجود در مدل بسنده کنید. به عبارت دیگر پس از دریافت داده‌ها از مدل بر روی آنها پردازشی انجام ندهید. در صورتی که فراخوانی داده‌ها از مدل نیازمند پردازش‌های زیاد قبل و یا بعد از فراخوانی باشد، بهتر است اجازه دهید پردازش‌ها را مدل برای شما انجام دهد.

نکته در صورتی که برای هر نما مجبورید مجموعهٔ کدهای تکراری را به کار ببندید، توصیه می‌شود این قبیل کد را به مدل منتقل کنید.

اتصال محکم

نما و کنترل کننده، هر دو باید با متدهای مدل پیوند محکمی داشته باشند. با تغییر در رابط مدل باید نما و کنترل کننده را تغییر داد. زمانی که از الگوی MVC استفاده می‌کنید، ابتدا باید از پایداری مدل مطمئن شده، سپس به سراغ نماد کنترل کننده بروید. در صورتی که این کار مورد پسندتان نیست، بهتر است از الگوی طراحی دیگری استفاده کنید.

نما و کنترل کننده نیز خود رابطه محکمی دارند. هر کنترل کننده همیشه با یک نمای مشخص به کار برده می‌شود. با استفاده از طراحی مناسب می‌توانید از قابلیت استفاده مجدد (reuse) بهره‌های کافی ببرید.

ناکارآمدی

به هنگام طراحی رابطهای کاربری (UI) مبتنی بر MVC باید مراقب ناکارآمدی آن باشید. یک مدل می‌تواند از انتشار تغییرات غیرضروری به گیرندگانش خودداری کند. درواقع با صف کردن تغییرات مرتبط به نحوی که با یکدیگر یک تغییر عمده را به وجود آورند، می‌توان از ناکارآمدی سیستم جلوگیری کرد. مدل رخدادها در AWT از این روش برای رسم بر روی صفحه نمایش استفاده می‌کند.

به هنگام طراحی نما و کنترل می‌توان در صورتی که انتقال داده از مدل به کندی صورت می‌گیرد آنها را به صورت موقت ذخیره کرد و به هنگام تغییر وضعیت در مدل تنها نسبت به دریافت تغییرات مبادرت کرد.

خلاصه

رابط کاربری یکی از اجزای مهم هر سیستم است. می‌توان همان روشی را که برای تحلیل، طراحی و پیاده‌سازی باقی اجزای سیستم به کار می‌بریم برای رابط کاربری هم به کار ببندیم. رابط کاربری نباید آخرین چیزی باشد که به آن فکر می‌شود و در دقیقه آخر هم پیاده‌سازی شود. اگرچه روشهای بسیاری برای طراحی رابط کاربری وجود دارد ولی MVC روشی برای طراحی ارایه می‌کند که با جداسازی UI از باقی سیستم پایداری خوبی به سیستم می‌بخشد. با این حال همچون دیگر طراحیها باید قبل از استفاده از آن جنبه‌های مختلف آن را بررسی کنید: MVC شما را از واقعیتهای سیستم جدا نمی‌کند!

پرسشها و پاسخها

کلاس BankAccountModel شامل تمامی منطق سیستم است. آیا مدل همیشه باید شامل منطق کار سیستم باشد و یا آنکه می‌تواند به عنوان دروازه‌ای برای سیستم واقعی باشد؟

بستگی دارد. گاهی اوقات مدل می‌تواند به عنوان دروازه ورودی به سیستم عمل کند ولی گاهی هم پیش می‌آید که مدل باید شامل خود سیستم باشد. به هر حال این امر کاملاً به طراحی برمی‌گردد. در مورد رابط کاربری مهم نیست که مدل شامل خود سیستم هست یا خیر.

در لیست ۱۳ ـ ۶ ارایه شده، داشتیم که

```
if (Command.equals (view.WITHDRAW)) {
    model.withdrawFunds (amount);
} else if (command.equals.(View.DEPOSIT)) {
    model.depositfunds(amount);
}
```

به نظر شما، این خطوط منطق شرطی را به یاد نمی‌آورند؟ آنگونه که بخاطر داریم منطق شرطی از لحاظ شیءگرایی بد بود؟

بله، مثال ارایه شده شامل منطق شرطی است. برای آنکه بتوانم مثال را ساده بگیرم، تصمیم گرفتم که کنترل کننده را به نحوی طراحی کنم که بتواند پاسخگوی هر دو رخداد باشد. در پیاده‌سازی واقعی، از منطق شرطی باید چشم‌پوشی کنید. برای این کار بهتر است که رخدادها را نما دریافت کرده وبر اساس هر رخداد، خود رخداد جداگانه را ایجاد نماید. برای مثال، نما در نمونهٔ ارایه شده دو رخداد واریز و برداشت را می‌تواند تولید کند. در این حالت کنترل کننده باید به این دو رخداد پاسخ گوید. بنابراین نما در این حالت متدهای لازم را از کنترل کننده صدا خواهد زد و دیگر دستورات همراه با منطق شرطی را نخواهیم داشت.

خب، با پاسخ ارایه شده در سؤال قبل، اندکی قانع شدم. ولی اگر کنترل کننده شکل که در بالا گفته شد، پیاده‌سازی شود، خود نما نباید از منطق شرطی استفاده کند تا بفهمد کدام دکمه فشار داده شده است؟

خیر، نما می‌تواند با ارایه گیرنده‌های مختلف برای هریک از دکمه‌ها از منطق شرطی جلوگیری کند. زمانی که رابطه‌ای یک به یک برای گیرنده‌ها و عناصر موجود است، نیازی به منطق شرطی نیست.

کارگاه

سؤالات ارایه شده در این بخش تنها برای درک بیشتر از مفاهیم ارایه شده در طول درس آورده شده‌اند.

پرسش‌ها

۱. چگونه می‌توان تحلیل، طراحی و پیاده‌سازی رابط کاربری را به صورتی جداگانه از سیستم انجام داد؟
۲. چرا رابط کاربری را از خود سیستم جدا می‌کنیم؟
۳. سه جزء موجود در الگوی MVC کدام‌ها هستند؟
٤. دو انتخاب دیگر برای الگوی MVC را نام ببرید.
۵. وظایف مدل را برشمارید.
۶. وظایف نما را برشمارید.
۷. وظایف کنترل کننده را برشمارید.
۸. یک سیستم چه تعداد مدل می‌تواند داشته باشد؟ هر مدل چه تعداد نما می‌تواند داشته باشد؟ و بالاخره هر نما چند کنترل کننده می‌تواند داشته باشد؟
۹. به هنگام استفاده از الگوی MVC چه ملاحظاتی را در مورد ناکارآمدی باید مدنظر داشته باشید؟
۱۰. چه ملاحظاتی را به هنگام استفاده از الگوی MVC باید در نظر گرفت؟
۱۱. تاریخچه الگوی MVC چیست؟ (برای پاسخ به این پرسش نیازمند جستجو در اینترنت هستید).

تمرین‌ها

۱. لیست ۱۳ ـ ۸ یک کلاس Employee را نشان می‌دهد. کلاس Employee را به نحوی تغییر دهید تا بتواند گیرنده‌هایش را اعلام کرده و یا حذف نماید تا از آن طریق وضعیت خود را به اطلاع آنها برساند. لیست ۱۳ ـ ۲ یک رابط برای Observer را نشان می‌دهد که می‌توانید از آن برای این مثال استفاده کنید.

لیست ۱۳-۸ EmployeeModel.java

```
public abstract class Employee {

   private String first_name;
   private String last_name;
   private double wage;

   public Employee(String first_name,String last_name,double wage) {
      this.first_name = first_name;
      this.last_name  = last_name;
      this.wage = wage;
   }

   public double getWage() {
       return wage;
   }

   public void setWage( double wage ) {
      this.wage = wage;
   }

   public String getFirstName() {
      return first_name;
   }
```

لیست ۱۳-۸ (ادامه)

```
    public String getLastName() {
        return last_name;
    }

    public abstract double calculatePay();

    public abstract double calculateBonus();

    public void printPaycheck() {
        String full_name = last_name + ", " + first_name;
        System.out.println( "Pay: " + full_name + " $" + calculatePay() );
    }

}
```

۲. لیستهای ۱۳ ـ ۹ و ۱۳ ـ ۱۰ را به عنوان نقطه آغازین در نظر گرفته و کنترل کننده جدیدی بنویسید که رابط BankActivityListener جدیدی را پیاده‌سازی کرده و بدون استفاده از منطق شرطی به رخدادها پاسخ گوید. لیست ۱۳ ـ ۹ کلاس‌های BankActivityEvent و BankActivityListerner مرتبط با آن را نشان می‌دهد.

لیست ۱۳-۹ BankActivityEvent.java و BankActivityListener.java

```
public interface BankActivityListener {

    public void withdrawPerformed( BankActivityEvent e );

    public void depositPerformed( BankActivityEvent e );

}

public class BankActivityEvent {

    private double amount;

    public BankActivityEvent( double amount ) {
        this.amount = amount;
    }

    public double getAmount() {
        return amount;
    }

}
```

لیست ۱۳ ـ ۱۰ نسخه به‌روز شده‌ای از BankAccountView را نشان می‌دهد. این کلاس رخداد مربوط به دکمه را گرفته و متناظر با آن رخداد جدیدی را به کنترل کننده ارسال می‌کند.

لیست ۱۳-۱۰ BankAccountView.java

```
import javax.swing.JPanel;
import javax.swing.JLabel;
import java.awt.BorderLayout;
import javax.swing.JTextField;
import javax.swing.JButton;
import java.util.ArrayList;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class BankAccountView extends JPanel implements Observer {

    public final static String DEPOSIT  = "Deposit";
    public final static String WITHDRAW = "Withdraw";

    private BankAccountModel model;
    private BankAccountController controller;

    // GUI Elements, pre-allocate all to avoid null values
    private JButton depositButton  = new JButton( DEPOSIT );
    private JButton withdrawButton = new JButton( WITHDRAW );
    private JTextField amountField = new JTextField();
    private JLabel balanceLabel    = new JLabel();

    public BankAccountView( BankAccountModel model ) {
        this.model = model;
        this.model.register( this );
        attachController( makeController() );
        buildUI();
    }

    // called by model when the model changes
    public void update() {
        balanceLabel.setText( "Balance: " + model.getBalance() );
    }

    // wires the given controller to the view, allows outside object to set controller
    public void attachController( BankAccountController controller ) {
        this.controller = controller;
    }

    protected BankAccountController makeController() {
        return new BankAccountController( this, model );
    }

    // provides access to the amount entered into the field
    private double getAmount() {
```

لیست ۱۳-۱۰ (ادامه)

```
        // assume that the user entered a valid number
        return Double.parseDouble( amountField.getText() );
    }

    private void fireDepositEvent() {
        BankActivityEvent e = new BankActivityEvent( getAmount() );
        controller.depositPerformed( e );
    }

    private void fireWithdrawEvent() {
        BankActivityEvent e = new BankActivityEvent( getAmount() );
        controller.withdrawPerformed( e );
    }

    private void buildUI() {

        setLayout( new BorderLayout() );

        // associate each button with a commend string
        depositButton.setActionCommand( DEPOSIT );
        withdrawButton.setActionCommand( WITHDRAW );

        // build the display
        JPanel buttons = new JPanel( new BorderLayout() );
        JPanel balance = new JPanel( new BorderLayout() );
        buttons.add( depositButton, BorderLayout.WEST );
        buttons.add( withdrawButton, BorderLayout.EAST );
        balance.add( balanceLabel, BorderLayout.NORTH );
        balance.add( amountField, BorderLayout.SOUTH );
        add( balance, BorderLayout.NORTH );
        add( buttons, BorderLayout.SOUTH );

        depositButton.addActionListener(
            new ActionListener() {
                public void actionPerformed( ActionEvent e ) {
                    fireDepositEvent();
                }
            }
        );

        withdrawButton.addActionListener(
            new ActionListener() {
                public void actionPerformed( ActionEvent e ) {
                    fireWithdrawEvent();
                }
            }
    }
}
```

روز ۱۴

آزمون: راه اعتماد به نرم‌افزار

Object Orinted Programming

در برنامه‌نویسی شیءگرا همواره تلاش بر این است که نرم‌افزاری نوشته شود طبیعی، قابل اعتماد، قابل استفاده و مکرر و توسعه‌پذیر در مدت زمانی قابل قبول. برای رسیدن به این اهداف، باید متوجه باشید که هیچ یک از آنها تصادفی حاصل نمی‌شوند. باید با تحلیل و طراحی دقیق به مسأله حمله کنید، در عین حال هرگز نباید از اصول شیءگرا غفلت کنید. فقط در این شرایط است که برنامه‌نویسی شیءگرا قدرتهای خود را به نمایش می‌گذارد. حتی با تحلیل و طراحی دقیق هم، برنامه‌نویسی شیءگرا فرمولی جادویی نخواهد شد. یعنی نمی‌تواند برنامه‌نویس را از نتایج اشتباهات خودش و دیگران در امان دارد. به هر صورت اشتباه رخ خواهد داد! بنابراین باید نرم‌افزار را آزمایش کنید.

آنچه امروز خواهید آموخت:

- آزمون در فرایند تکرار
- انواع مختلف آزمون
- چگونه کلاس‌ها را امتحان کنیم
- چگونه نرم‌افزار ناتمام را آزمایش کنیم
- چه کنیم تا کد قابل اعتماد بیشتری بنویسیم
- چگونه آزمایش را مؤثرتر کنیم

آزمون نرم‌افزار شیءگرا

روش شیءگرا جلوی اشتباه کردن را نمی‌گیرد. حتی بهترین‌ها هم اشتباه می‌کنند. خطاها از اشتباه در طراحی، منطق یا حتی نوشتن کد ایجاد می‌شوند. با وجودی که اغلب خطاها در پیاده‌سازی رخ می‌دهند، اما برخی هم به صورتهای دیگر و در جاهای دیگر ظاهر می‌شوند.

وقتی شیئی از شیء دیگر استفاده نادرست می‌کند، ایجاد خطا بالقوه ممکن است. طبیعت سیستم شیءگرا دقیقاً مانند یک شیء متعامل است. این تعامل‌ها منبع ایجاد انواع خطاها هستند.

خوشبختانه، با استفاده از آزمون نرم‌افزار می‌توان از آسیب خطاها اجتناب کرد.

توجه

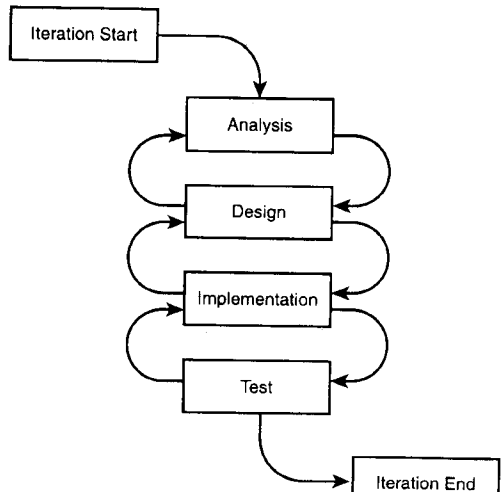
دقیقاً مانند خود روش شیءگرا، آزمون هم ابزاری جادویی نیست. آزمودن کامل نرم‌افزار، بسیار مشکل است. راههای مختلفی که می‌توان برنامه را به آنها کشاند به قدری زیاد و وقت‌گیرند، که آزمودن آنها را بسیار مشکل و زمانبر می‌کند. بنابراین حتی کد آزموده شده هم می‌تواند حامل خطاهای پنهان باشد.

بهترین کار این است که نرم‌افزار را به حدی تست کنید که تا جای ممکن خطاها یافت شده و رفع شوند، ضمناً از زمانبندی پروژه عقب نیافتید و نیز از لحاظ هزینه‌ها به صرفه باشد. در این صورت میزان دقت تست، توسط شرایط پروژه و مشتری و خود شما تعیین خواهد شد.

آزمون و فرایند توسعه نرم‌افزار تکراری

شکل ۱۴ ـ ۱ روند تکرار مطرح شده در فصل ۹ را نشان می‌دهد. آزمون، آخرین مرحلهٔ فرایند تکرار است. قبل از اینکه روند تکرار را ترک کنید، آزمون گام مهمی است که باید الزاماً طی شود. مرحلهٔ تست برای اطمینان از این است که تغییرات اعمال شده در روند تکرار کارایی‌های موجود را از بین نمی‌برند. علاوه بر این، در مرحله تست صحت کارکرد کارایی‌های قبلی را مورد بررسی قرار می‌دهیم. به همین دلایل آزمونهایی که قبل از ترک تکرار انجام می‌شوند را آزمونهای عملیاتی یا کارکردی یا قبولی می‌نامند.

اگر خطایی یافت شد، باید برگردیم و آن را تصحیح کنیم. بیشتر اوقات بازگشت به پیاده‌سازی کفایت می‌کند، اما گاهی هم باید به طرح یا تحلیل اولیه بازگشت کنید.



شکل ۱۴ ـ ۱
یک تکرار

نکته

آزمون را جزو اهداف خود و آنچه در طی فرایند توسعه نرم‌افزار انجام می‌دهید قرار دهید. اگر تست را به انتهای کار واگذار کنید، شاید هرگز موفق به انجام آن نشوید. بنابراین آزمون را جزو ساختاری روش کار خود کنید.

توجه

بعد از رفع یک خطا، نباید تنها به بررسی آن قسمت کد بپردازید. بلکه تمام تست‌ها باید از اول انجام شوند. زیرا در حین رفع یک خطا، به راحتی می‌توان به اشتباه اشکال دیگری ایجاد کرد.

اشتباه در تحلیل اولیه یعنی اینکه نرم‌افزار مطابق آنچه مشتری انتظار دارد کار نخواهد کرد. بنابراین نه تنها باید کد را برای یافتن اشتباهات پیاده‌سازی مورد بررسی قرار داد، بلکه باید قابل قبول بودن کارایی آن از نظر مشتری هم مورد بررسی قرار گیرد.

برای اینکه قادر باشید یک سیستم را تست کنید، باید حالتها و شرایط مختلف آزمون را نوشته و اجرا کنید. هر حالت یا مرحلهٔ تست، یکی از جنبه‌ها یا خواص سیستم را مورد بررسی قرار خواهد داد.

واژه جدید

هر مرحله آزمون، یک بلوک ساختمانی فرایند آزمون است. فرایند آزمون مجموعه‌ای از این مراحل است که با انجام آنها می‌توان یک سیستم را معتبر تشخیص داد. هر مرحلهٔ تست از گروهی ورودی و مجموعه‌ای از خروجی‌های مطلوب تشکیل می‌شود. آزمون ممکن است از فرایندی ویژه درون سیستم (جعبهٔ سفید) یا رفتار تعریف شدهٔ خاصی (جعبهٔ سیاه) صورت پذیرد.

هر مرحلهٔ تست، بخش خاصی از کارایی سیستم را برای ارزشیابی عملکرد آن طبق تعریف، مورد بررسی قرار می‌دهد. اگر سیستم مطابق انتظار عمل نکند، آزمون رد می‌شود. در این صورت می‌دانیم که خطایی در سیستم وجود دارد. همواره سعی کنید سیستم ۱۰۰٪ آزمون‌ها را پاس کند. یک آزمون رد شده را حتی با وجود صد آزمون مثبت نادیده نگیرید.

واژه جدید

آزمون جعبهٔ سیاه، بررسی می‌کند که آیا سیستم تمام کارایی‌های مورد انتظار را دارد یا خیر. با داشتن یک ورودی معین، آزمون جعبهٔ سیاه وجود خروجی یا رفتار نتیجهٔ مناسب تعریف شده توسط مشخصات سیستم را بررسی می‌کند.

واژه جدید

در آزمون جعبهٔ سفید، آزمون‌ها بر پیاده‌سازی روالها متمرکز می‌شوند. در این آزمونها سعی می‌شود ۱۰۰٪ کد مورد بررسی قرار گیرد.

وقتی کلاسی مورد آزمون باشد، آزمون جعبهٔ سیاه بر اساس نیازمندیهای کارکردی کلاس صورت می‌گیرند. اما در آزمون تمام سیستم، تست جعبه سیاه بر اساس مراحل تست انجام می‌شود. در هر دو صورت، این آزمون مطابق انتظار بودن رفتار سیستم یا شیء را ارزیابی می‌کند. مثلاً اگر قرار باشد روالی دو عدد را با هم جمع کند، در آزمون جعبه سیاه دو عدد به روال پاس می‌شود و سپس حاصل جمع بودن خروجی مورد بررسی قرار می‌گیرد.

از سوی دیگر، آزمون جعبهٔ سفید روی پیاده‌سازی روال تمرکز دارد. هدف اصلی این آزمون، حصول اطمینان از اجرا شدن تمام بخش‌های کد است. در حالی که آزمون جعبه سیاه تقریباً یک سوم تا یک دوم کد را بررسی می‌کند.

تذکر

برای حصول اطمینان از کارایی آزمون جعبه سفید دو کار را باید انجام دهید:
- طوری برنامه بنویسید که حاوی حداقل مسیرهای ممکن باشد.
- مسیرهای بحرانی را پیدا کنید و حتماً آنها را آزمایش کنید.

فرضاً در صورتی که روالی دو عدد را بر هم تقسیم می‌کند و بخشی برای کنترل خطا وجود دارد که از تقسیم بر صفر جلوگیری می‌کند، حتماً باید عملکرد اصلی و شرط خطا را مورد بررسی قرار دهید. از آنجایی که به برخی بخشهای کد به هیچ وجه در مستندسازی اشاره نمی‌شود، لذا آزمون جعبه سفید حتماً باید بر اساس خود کد طرح شود.

به هر صورت آزمونهای جعبه سیاه و جعبه سفید چگونگی آزمودن و ایجاد مراحل تست را شکل می‌دهند. هر کدام نقش مهمی در تعیین شکل آزمون خواهند داشت.

اشکال آزمون

به طور کلی، چهار شکل مختلف آزمون وجود دارد. این آزمونها از سطوح پایین، در حد آزمایش تک تک اشیاء تا سطوح بالا، آزمایش کل سیستم متغیرند. انجام هریک کیفیت کلی نرم‌افزار را تضمین می‌کند.

آزمون واحد

این آزمون در پایین‌ترین سطح آزمایشات قرار دارد. یک آزمون واحد، در یک زمان تنها یک ویژگی را مورد آزمایش قرار می‌دهد.

واژه جدید آزمون واحد یک پیام به شیء ارسال می‌کند و دریافت نتیجه مورد انتظار را ارزیابی می‌کند. در اصطلاح روش شیءگرا، آزمون واحد یک کلاس واحد شیء را آزمایش می‌کند. در آزمونهای واحد می‌توان هم از روش جعبه سیاه و هم از روش جعبه سفید استفاده کرد. درواقع برای حصول اطمینان از عملکرد صحیح شیء، باید هر دو را توأماً انجام داد. بهتر است مرحله آزمون مربوط به هر کلاس را در زمان نوشتن آن کلاس تدوین کرد.

آزمون مجتمع

سیستمهای شیءگرا از اجتماع اشیاء متعامل با یکدیگر ایجاد می‌شوند. آزمون مجتمع کیفیت تعامل این اشیاء با یکدیگر را مورد بررسی قرار می‌دهد. آنچه به تنهایی درست کار می‌کند، ممکن است در ترکیب با بقیهٔ اشیاء سیستمی ناپایدار تشکیل دهد. منابع عمدهٔ خطاهای اجتماع از خطا یا عدم درک صحیح قالبهای ورودی/خروجی، تداخل منابع و ترتیب ناصحیح فراخوانی روالها هستند.

واژه جدید آزمون اجتماع، عملکرد مشترک دو یا چند کلاس را بررسی می‌کند.

دقیقاً مانند آزمون واحد، این آزمون هم می‌تواند به صورت جعبه سفید و هم به صورت جعبه سیاه صورت پذیرد. توجه داشته باشید که هر تعامل مهمی در سیستم باید مورد آزمایش مجتمع قرار گیرد.

آزمون سیستم

آزمون سیستم عملکرد کل سیستم نسبت به انتظارات را در چهارچوب مراحل تست بررسی می‌کند. علاوه

بر این سیستم باید در شرایطی که در شرح وظایف سیستم نیست هم مورد آزمایش قرار داد. در این صورت سیستم می‌تواند در مقابل شرایط پیش‌بینی نشده هم مقاوم باشد.

تذکر

آزمونهای زیر هم مفید خواهند بود:

آزمون اعمال تصادفی

عبارت است از کار با سیستم به صورت تصادفی

آزمون پایگاه داده خالی

این اطمینان را ایجاد می‌کند که سیستم در شرایطی که پایگاه داده دچار مشکل شود هم کار خواهد کرد.

آزمون جهش یافته

با تغییر مراحل تست به صورت پیش‌بینی نشده، می‌توان سیستم را در مقابل برخی تعامل‌ها بیمه کرد.

اگر بدانید کاربر می‌تواند چه بلاهایی بر سر سیستم بیاورد، شگفت‌زده خواهید شد! کاربر می‌تواند سیستم را در شرایط غیرعادی قرار دهد. بنابراین بهتر است خود را برای بدترین شرایط آماده کنید.

واژه جدید

آزمون سیستم کل سیستم را مورد بررسی قرار می‌دهد. آزمون سیستم عملکرد سیستم را هم در شرایط مورد انتظار و هم در شرایط پیش‌بینی نشده بررسی و ارزیابی می‌کند.

آزمونهای تنش و کارایی هم در چهارچوب آزمون سیستم قرار می‌گیرند. در این صورت می‌توان اطمینان داشت که سیستم تحت هر فشاری کارایی مورد نظر را دارا خواهد بود. در صورت امکان، آزمودن سیستم در شرایط واقعاً عملی بسیار مفید خواهد بود.

آزمون سیستم بخش مهمی در فرایند قبول سیستم توسط مشتری است. زیرا تمام کارایی‌ها و ویژگیها را یکجا و همزمان آزمایش می‌کند. در این آزمون بسیاری از اشیاء و زیرسیستم‌ها مورد بررسی قرار می‌گیرند. سیستم در صورتی می‌تواند روند تکرار را ترک کند که از این آزمونها با موفقیت بیرون بیاید.

آزمون واپسگرد

این آزمون تنها در صورتی معتبر است که بعد از آن سیستم تغییر نکرده باشد. وقتی یک جنبه تغییر می‌کند، آن قسمت و تمام بخش‌های وابسته باید دوباره آزمایش شوند. درواقع این آزمون تکرار آزمونهای واحد، مجتمع و سیستم بعد از انجام تغییرات می‌باشد.

واژه جدید

آزمون واپسگرد، تغییرات در اجزایی از سیستم را بررسی می‌کند که قبلاً ارزیابی شده‌اند. بسیار مهم است که حتی بعد از یک تغییر کوچک هم آزمون را تکرار کرد. یک تغییر کوچک می‌تواند خطایی ایجاد کند که بالقوه قادر به تخریب کل سیستم باشد. خوشبختانه آزمون واپسگرد، ساده است.

راهنمای نوشتن کد قابل اعتماد

از آنجایی که تمام اشکال آزمایش در کیفیت کلی نرم‌افزار مؤثرند، امروز روی آنچه می‌توان در کار روزمره برای اطمینان از کیفیت سیستم‌هایی که می‌نویسید مورد استفاده قرار گیرد، تمرکز خواهیم کرد. برای اینکه

بتوانید کد مطمئن بنویسید، باید آزمون واحد را انجام دهید، بیاموزید بین شرایط خطا تفکیک قایل شوید و مستندات قابل استفاده ایجاد کنید.

ترکیب توسعه و آزمون

حقیقتی که در شکل ۱۴ ـ ۱ نادیده گرفته شده این است که آزمون باید فرایندی مستمر باشد. آزمون نباید کنار گذاشته شود، به انتهای کار موکول شود، توسط کس دیگری انجام شود یا اصلاً فراموش شود. درواقع، غیرممکن است که وقتی ساخت سیستم به پایان رسید، به طور ناگهانی شروع به آزمایش آن کرد. باید آزمایش را در فرایند توسعه ترکیب کنید. در این صورت باید برای هر کلاسی که می‌نویسید، آزمون واحد تدوین کنید.

یک آزمون واحد نمونه

کلاس SavingAccount معرفی شده در فصل ۵ را در نظر بگیرید. می‌خواهیم برای آن آزمون واحد تدوین کنیم.

لیست ۱۴-۱ SavingAccountTest.java

```
public class SavingAccountTest {
    public ststic void main( String [] args ) {
        SavingAccountTest sat = new SavingAccountTest();
        sat.test_applyingInterest();
    }

    public void test_applyingInterest() {

        SavingAccount acct = new SavingAccount( 10000.00, 0.05 );

        acct.addInterest();

        print_getBalanceResult( acct.getBalance(), 10500.00, 1 );

    }

    private void print_getBalanceResult( double actual, double expected, int test ) {
        if( actual == expected ) { //passed
            System.out.println( "PASS: test #" + test + "interest applyied properly" );
            System.out.println();
        } else { //failed
            System.out.println( "Value returned:" + actual );
            System.out.println( "Expected value:" + expected );
            system.out.println();
        }
    }
}
```

در اینجا با SavingAccountest که همان آزمون واحد است تنها یک حالت آزمون دارد: test_applyingInterest در این آزمون اطمینان حاصل می‌شود که متغیر SavingAccount سود را به درستی در قیمت اعمال می‌کند. آزمون واحد می‌تواند شامل چندین حالت آزمون مختلف باشد، ولی هریک از حالتها تنها باید یک ویژگی را بررسی کند.

SavingAccountTest یک آزمون واحد است، چون سطح پایین‌ترین بلوک‌های ساختمانی یا واحدها را آزمایش می‌کند. واحد دنیای شیءگرا همان شیء است! هر آزمون واحد باید در هر زمان تنها یک شیء را آزمایش کند.

وقتی آزمون واحد را می‌نویسید، باید تا جای ممکن از ارزیابی دستی خروجی پرهیز کنید. ارزیابی غیر خودکار اکثر اوقات زمانبر و خطاساز است.

چرا باید آزمون واحد نوشت

آزمونهای واحد به یافتن خطاها کمک می‌کنند. اگر چیزی را در کلاس تخریب کنید، سریع متوجه خواهید شد، چون آزمون واحد آن را فاش خواهد کرد. آزمون واحد اولین سنگر دفاعی برنامه‌نویس در مقابل خطا است.

علاوه بر این آزمون‌های واحد کامل بودن کلاس را هم مشخص می‌کنند. زیرا کلاسی کامل است که تمام آزمون‌های واحد را با موفقیت بگذراند. در غیر این صورت، باور کنید فهمیدن این موضوع مشکل است! دانستن این موضوع از افزودن کارایی بیش از حد به کلاس جلوگیری می‌کند.

نوشتن آزمون واحد باعث می‌شود به طراحی کلاسها دوباره فکر کنید، به خصوص اگر حالت تست را قبل از نوشتن کلاس بنویسید. چنین کاری این آزادی را می‌دهد که بهترین رابط را برای کلاس خود طرح کنید. وقتی نوشتن آزمون را تمام کردید، به کلاس بپردازید. در این صورت همه چیز درست انجام خواهد شد.

داشتن آزمون واحد، ایجاد تغییر در کد را ساده‌تر می‌کند. زیرا با هر تغییری، آزمون واحد می‌تواند اثر آن را روی کد کاملاً نشان دهد، در این صورت همواره این پرسش که: «آیا چیزی را خراب کردم؟» آزارتان نمی‌دهد و می‌توانید با آرامش تغییرات لازم را ایجاد کنید.

و در پایان شاید خود شما همواره برای آزمون سیستم حاضر نباشید. آزمون‌های واحد اجازه می‌دهند که دیگران هم بتوانند شیء را آزمایش کنند.

نوشتن آزمون واحد

SavingAccountTest و test_applyingInterest() مثالهای ساده‌ای از آزمون واحد و حالتهای تست هستند. با این حال نوشتن این آزمون‌ها از صفر، بسیار وقت‌گیر خواهد بود. سیستمی را در نظر بگیرید که در آن مجبور باشید صدها حالت آزمون را در نظر بگیرید. اگر بخواهید آزمونهای واحد را برای چنین سیستمی از صفر بنویسید، بدا به حالتان! بهتر است که از یک چهارچوب آزمون (Testing Framework) استفاده کنید.

واژه جدید

چهارچوب عبارت است از مدل دامنه قابل استفاده مکرر. چهارچوب حاوی تمام کلاسهای مشترک در یک دامنهٔ مسایل است و به عنوان اساس یک برنامه کاربردی در دامنه عمل می‌کند.

کلاسهای چهارچوب طرح عمومی برنامه کاربردی را تعریف می‌کنند. توسعه دهنده این کلاسها را برای مسأله مفروض بسط می‌دهد، بنابراین می‌تواند کلاسهایی برای همان مسأله خاص داشته باشد و برنامه را با

آنها ایجاد کند.

اما چهارچوب آزمون، اسکلتی را تعریف می‌کند که می‌توان از آن برای نوشتن و اجرای آزمون واحد استفاده کرد. چهارچوب تست امکان نوشتن آزمون واحد را به سرعت و بااطمینان فراهم می‌کند. این کار با کم کردن میزان عملیات اضافی و خطاپذیر صورت می‌پذیرد. به یاد داشته باشید که هر چیزی در برنامه می‌تواند حاوی خطا باشد، حتی کد آزمون. بنابراین در اختیار داشتن یک چهارچوب آزمون مطمئن و آزموده می‌تواند جلوی بسیاری از خطاهای آزمایش را بگیرد. بدون استفاده از چهارچوب آزمون، تصور حجم عظیم کد لازم برای پیاده‌سازی کافی است که برنامه‌نویس را از نوشتن آن منصرف کند.

چهارچوب آزمون کامل حاوی کلاسهای پایه‌ای برای نوشتن آزمونهای واحد خواهد بود. همچنین پشتیبانی ذاتی از آزمونهای خودکار و داشتن برنامه‌های کمکی برای تشخیص و نمایش خروجی جزو خواص چهارچوب آزمون کامل هستند. در درس امروز به یادگیری JUnit که یک چهارچوب تست برای آزمایش کلاسهای Java است، خواهیم پرداخت. JUnit را می‌توان از http://www.junit.org تهیه کرد.

نکته

مجموعه JUnit دارای مقدار معتنابهی کد منبع می‌باشد. JUnit طرحی عالی به همراه مجموعه‌ای کد و مستندات آموزشی در اختیار شما قرار می‌دهد. به خصوص الگوی طراحی مورد استفاده خود JUnit خیلی خوب مستند شده است.

JUnit

JUnit دارای کلاسهایی برای نوشتن آزمونهای واحد، ارزیابی خروجی و اجرای حالتهای تست تحت یک GUI یا خط فرمان می‌باشد. junit.framework.testcase کلاس عام تعریف آزمونهای واحد است. بنابراین برای نوشتن یک آزمون کافی است کلاسی از آن مشتق کنید، برخی روالها را جایگزین کنید و روالهای خاص مربوط به آن را اضافه کنید.

نکته

وقتی از JUnit برای نوشتن آزمون واحد استفاده می‌کنید، همواره باید نام آزمون را با test آغاز کنید. زیرا JUnit ساختاری در اختیار دارد که با استفاده از آن هر روالی را که نام آن با عبارت test آغاز شده باشد را بارگذاری و اجرا خواهد کرد.

لیست ۱۴ ـ ۲ نسخه JUnit آزمون SavingAccountTest را نشان می‌دهد.

لیست ۱۴-۲ SavingAccountTest.java

```
import junit.framework.TestCase;
import junit.framework.Assert;

public class SavingsAccountTest extends TestCase {

    public void test_applyingInterest() {
        SavingsAccount acct = new SavingAccount( 10000.00, 0.05 );
```

لیست ۱۴-۲ (ادامه)

```
        acct.addInterest();

        Assert.assertTrue( "interest applied incorrectly", acct.getBalance() == 10500.00 );

    }

    public SavingsAccountTest( String name ) {
        super( name );
    }

}
```

نسخه JUnit آزمون از نسخه اصلی آن بسیار ساده‌تر است. زیرا نیازی به نوشتن کد نمایش یا آزمون منطق نیست. فقط کافی است از کلاس Assert فراهم شده توسط JUnit استفاده کنید. کلاس Assert گروهی روال در اختیار دارد که یک متغیر منطقی را به عنوان ورودی دریافت می‌کنند. اگر متغیر مزبور نادرست باشد، یک خطا ثبت می‌شود. مثلاً در اینجا متغیر منطقی برگشتی از acct.getBalance == 10500.00 را به Assert ارسال می‌کند. اگر نتیجه مقایسه، نادرست باشد، JUnit یک خطا ثبت می‌کند.

این آزمونها را چگونه اجرا کنیم؟

JUnit برای اجرای آزمونها چندین راه در پیش رو قرار می‌دهد. این راهها به دو دسته تقسیم می‌شوند: پویا و ایستا. اگر بخواهید از روش ایستا استفاده کنید، باید runTest() را جایگزین کنید تا آزمون مورد نظر خودتان را انجام دهد.

در Java بهترین راه، نوشتن یک کلاس ناشناس خاص است تا برای هر آزمون لزومی به ایجاد یک کلاس جداگانه نباشد. لیست ۱۴ ـ ۳ را ببینید.

لیست ۱۴-۳ SavingAccountTest

```
SavingAccountTest test =
    new SavingAccountTest( "test_applyingInterest" ) {
        public void runTest() {
            test_applyingInterest();
        }
    };
    test.run();
```

با این ترفند می‌توان بدون نیاز به ایجاد کلاس در فایل جدید، هنگام تعریف متغیر از شیء روالی را جایگزین کرد. در اینجا main() متغیری از SavingAccountTest تعریف می‌کند، اما runTest() را طوری جایگزین می‌کند که حالت آزمون test_applyingInterest() را اجرا کند.

واژه جدید

کلاس ناشناس (Anonymous)، کلاسی است که نامی ندارد. این نوع کلاس همان وقتی تعریف می‌شود که متغیر از آن تعریف شود. این کلاسها در فایل جداگانه یا در کلاس دیگری تعریف نمی‌شوند.

کلاسهای ناشناس برای استفاده به عنوان کلاسهای یکبار مصرف انتخابی ایده‌آل است. البته در صورتی که کلاس کوچک باشد. با استفاده از کلاس‌های ناشناس، می‌توان از دردسر تعریف کردن کلاسهای جداگانه گریخت.

البته این کلاس‌ها معایبی هم دارند. مثلاً برای هر حالت تست باید یک کلاس جداگانه تعریف نمود. در این صورت وقتی تعداد حالتهای تست زیاد شود، تعداد تعاریف کلاس هم افزایش می‌یابد.

برای غلبه بر این نقاط ضعف، JUnit مکانیزمی پویا برای جستجوی روالهایی که نامشان با _test شروع می‌شود، معرفی می‌کند. برای منظور درس امروز، از این مکانیزم پویا کمک خواهیم گرفت.

نکته

JUnit مکانیزمی تحت عنوان سویت آزمون (Test Suite) برای اجزای آزمون‌های چندگانه فراهم می‌کند. مکانیزمی ایستا برای تعریف گروهی از آزمایش‌ها به صورت یک سویت وجود دارد و در کنار آن مکانیزم پویا روالهای _test را خواهد یافت و آنها را اجرا خواهد کرد.

JUnit خواص جالبی دارد. نسخه به‌روز شده SavingAccountTest در لیست ۱۴ ـ ۴ را مشاهده می‌کنید که روال withdrawFunds() را نیز مورد آزمون قرار می‌دهد.

لیست ٤-١٤ SavingAccountTest.java

```
import junit.framework.TestCase;
import junit.framework.Assert;

public class SavingsAccountTest extends TestCase {

    private SavingsAccount acct;

    public void test_applyingInterest() {

        acct.addInterest();

        Assert.assertTrue( "interest applied incorrectly", acct.getBalance() == 10500.00 );

    }

    public void test_withdrawFunds() {

        acct.withdrawFunds( 500.00 );

        Assert.assertTrue( "incorrect amount withdrawn", acct.getBalance() == 9500.00 );

    }

    protected void setUp() {
        acct = new SavingsAccount( 10000.00, 0.05 );
```

لیست ۳-۲ (ادامه)

```
    }

    public SavingsAccountTest( String name ) {
        super( name );
    }
}
```

در این نسخه، آزمون دو روال تازه دارد. ()test_withdrawFunds و ()setup. ()setup روالی در کلاس پایه TestCase را جایگزین می‌کند. هر بار که JUnit یک روال تست را فراخوانی کند، ابتدا ()setup را برای بنا نهادن چیدمان ثابت (fixture) آزمون، فرا می‌خواند.

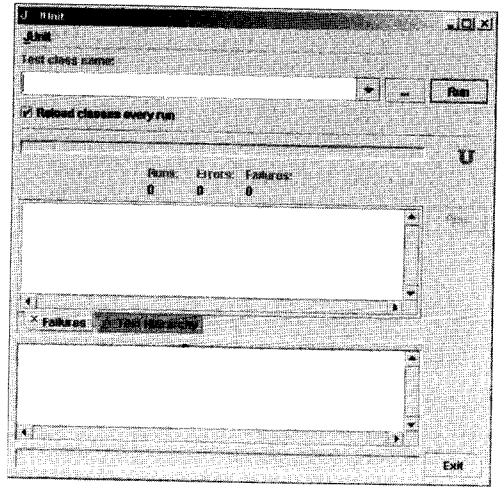
چیدمان آزمون مجموعه‌ای از اشیاء تعریف می‌کند که آزمون بر روی آنها اجرا خواهد شد. تعریف و بنا کردن چیدمان عموماً وقت‌گیرترین پروسهٔ نوشتن آزمون است. اهمیت دیگر چیدمان امکان استفاده چیدمان ثابتی برای مجموعه‌ای از حالت‌های آزمون، بدون نیاز به کدنویسی مجدد برای هریک می‌باشد.

JUnit تضمین می‌کند که اشیاء چیدمان در حالتی معلوم خواهند بود. این کار با فراخوانی ()setup قبل از اجرای دو آزمون انجام می‌شود. علاوه بر این JUnit روال مرتبط دیگری به نام ()tearDown برای پاکسازیهای چیدمان بعد از اجرای آزمون دارد.

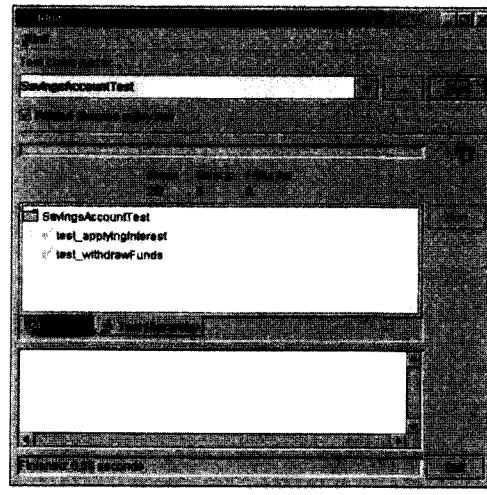
JUnit اجراکننده‌های آزمونی در اختیار دارد که برای بررسی و جمع‌آوری نتایج آزمایشها به کار می‌روند. این ابزار به دو صورت گرافیکی و خط فرمان موجود هستند. برای اجرای گرافیکی SavingAccountTest کافیست تایپ کنید:

```
java junit.swingui.TestRunner
```

شکل ۱۴ ـ ۲ پنجره اصلی JUnit را نشان می‌دهد.



شکل ۱۴ـ۲
روابط کاربر اصلی JUnit



شکل ۱۴ـ۳
آزمون توسط JUnit

با استفاده از UI می‌توان به دنبال SavingAccountTest گشت و آن را بارگذاری نمود. بعد از آن فقط کافی است روی دکمهٔ Run کلیک کنید.

همانطور که از شکل ۱۴ ـ ۳ واضح است، رابط ابزار JUnit تعداد آزمونها و نتایج هریک را نمایش می‌دهد. JUnit ابزار کارآمدی است چون اجازه می‌دهد به سرعت به نتیجه آزمون دست پیدا کنید. یعنی می‌توانید به سرعت از پرسش آزاردهنده «چه چیزی را خراب کردم؟» رها شوید.

نوشتن آزمون واحد پیشرفته

بیاید یک کلاس پیچیده‌تر را در نظر بگیریم. تمرین ۱ از فصل ۱۱ یک پیاده‌سازی برای Item پیشنهاد کرد. در حال حاضر برای اینکه بتوان Item را نمایش داد، باید گروهی روالهای get را فراخوانی کرده و خروجی آنها را برای نمایش پردازش نمود. بدبختانه، خواستن داده از خود شیء روشی شیءگرا نیست. باید از شیء بخواهیم که کاری با داده انجام دهد. لیست ۱۴ ـ ۵ را در نظر بگیرید. یک پیشنهاد دیگر برای پیاده‌سازی Item.

لیست ۱۴-۵ Item.java

```
public class Item {

    private int    id;
    private int    quantity;
    private float  unitPrice;
    private String description;
    private float  discount;

    public Item( int id, int quantity, float unitPrice, float discount, String desc ) {
        this.id = id;
        this.quantity = quantity;
        this.unitPrice = unitPrice;
        this.discount = discount;
```

لیست ۱۴-۵ (ادامه)

```
        this.description = desc;
    }

    public void display( ItemDisplayFormatter format ) {

        format.quantity( quantity );
        format.id( id );
        format.unitPrice( unitPrice );
        format.discount( discount );
        format.description( description );
        format.adjustedPrice( getTotalPrice() );

    }

    public float getTotalPrice() {
        return ( unitPrice * quantity ) - ( discount * quantity );
    }

}
```

در اینجا می‌توان از Item خواست که خود را با استفاده از یک روال قالب‌بندی داده نمایش دهد. استفاده از یک شیء قالب‌بندی کننده جداگانه روش بهتری نسبت به استفاده از توابع get یا نمایش داده توسط خود شیء است. وقتی نیازهای نمایش تغییر کرد، فقط کافی است پیاده‌سازی ItemDisplayFormatter را تغییر دهیم.

لیست ۱۴ ـ ۶ رابط ItemDisplayFormatter را نمایش می‌دهد. لیست ۱۴ ـ ۷ هم یک پیاده‌سازی ممکن را نشان می‌دهد.

لیست ۱۴-۶ ItemDisplayFormatter.java

```
public interface ItemDisplayFormatter {
    public void quantity( int quantity );
    public void id( int id );
    public void unitPrice( float unitPrice );
    public void discount( float discount );
    public void description( String description );
    public void adjustedPrice( float total );
    public String format();
}
```

لیست ۱۴-۷ ItemTableRow.java

```
public class ItemTableRow implements ItemDisplayFormatter {

    private int    quantity;
    private int    id;
```

لیست ۱۴-۷ (ادامه)

```
    private float  unitPrice;
    private float  discount;
    private String description;
    private float  adjPrice;

    public void quantity( int quantity ) {
        this.quantity = quantity;
    }

    public void id( int id ) {
        this.id = id;
    }

    public void unitPrice( float unitPrice ) {
        this.unitPrice = unitPrice;
    }

    public void discount( float discount ) {
        this.discount = discount;
    }

    public void description( String description ) {
        this.description = description;
    }

    public void adjustedPrice( float total ) {
        this.adjPrice = total;
    }

    public String format() {
        String row = "<tr>";
        row = row + "<td>" + id          + "</td>";
        row = row + "<td>" + quantity    + "</td>";
        row = row + "<td>" + description + "</td>";
        row = row + "<td>" + unitPrice   + "</td>";
        row = row + "<td>" + adjPrice    + "</td>";
        row = row + "</tr>";
        return row;
    }
}
```

قالب‌بندی کننده ItemTableRow یک جدول HTML برای نمایش Item ایجاد می‌کند. قالب‌بندی کننده‌های دیگر می‌توانند داده را به صورتهای دیگر نمایش دهند.

این مثال تعدادی از معضلات سر راه آزمایش همچنین برخی امکانات جالب را نشان می‌دهند. برای این

کلاسها، تنها فراخوانی روال و بررسی پاسخ کافی نیست.

آزمون روال display()کلاس Item مورد خاصی است. زیرا آزمون واحد باید هر کلاس را جداگانه و ایزوله مورد آزمایش قرار دهد. با این حال برای اینکه بتوان display()را آزمایش کرد، باید ItemDisplayFormatterرا نیز به آن پاس کرد.

خوشبختانه اشیاء کاذب در این راه کمک بزرگی هستند.

واژه جدید

اشیاء کاذب (Mock Objects) جایگزین ساده شده‌ای برای اشیاء واقعی هستند. این شیء برای اهداف تست جعل می‌شود. با اینکه شیء کاذب پیاده‌سازی ساده شده‌ای دارد، کاربردی بیشتر برای کمک به آزمایش‌ها دارد.

گاهی اشیاء کاذب را شبیه‌ساز می‌نامند (Simulator). این اشیاء در سیستم اصلی ظاهر نخواهند شد و فقط در کد آزمون مورد استفاده قرار گیرند.

هدف شیء کاذب فراهم کردن کارایی کامل خود شیء اصلی نیست، بلکه، پیاده‌سازی‌ای فراهم می‌شود که برای آزمون مناسبتر است.

توجه

اشیاء کاذب را تا جای ممکن ساده نگه دارید. معمولاً این اشیاء باید مستقل باشند و به هیچ شی کاذب دیگری وابسته نباشند. زیرا در این صورت بیش از حد پیچیده خواهد بود.

به عنوان مثال، یک سیستم پایگاه داده را در نظر بگیرید که شیء Item را برمی‌گرداند. باید طوری شیء کاذب را برنامه‌ریزی کنید که همان Itemرا بارها و بارها بازگرداند. با این حال اگر شیء بازیافت کننده Item را تحت آزمون واحد دارید، کد آزمون تفاوتی را احساس نخواهد کرد. چنین کاری شیء مورد آزمون را از اثرات اشیاء دیگر مصون نگاه می‌دارد. می‌توان از یک شیء کاذب برای بررسی اینکه آزمون display()درست از ItemDisplayFormatter استفاده می‌کند بهره گرفت. لیست ۱۴ ـ ۸ یک ItemDisplayFormatter کاذب را نشان می‌دهد.

لیست ۱۴-۸ MockDisplayFormatter.java

```
import junit.framework.Assert;

public class MockDisplayFormatter implements ItemDisplayFormatter {

    private int    test_quantity;
    private int    test_id;
    private float  test_unitPrice;
    private float  test_discount;
    private String test_description;
    private float  test_adjPrice;

    private int    quantity;
    private int    id;
    private float  unitPrice;
```

لیست ۱۴-۸ (ادامه)

```
private float  discount;
private String description;
private float  adjPrice;

public void verify() {
    Assert.assertTrue( "quantity set incorrectly", test_quantity == quantity );
    Assert.assertTrue( "id set incorrectly", test_id == id );
    Assert.assertTrue( "unitPrice set incorrectly", test_unitPrice == unitPrice );
    Assert.assertTrue( "discount set incorrectly", test_discount == discount );
    Assert.assertTrue( "description set incorrectly", test_description == description );
    Assert.assertTrue( "adjPrice set incorrectly", test_adjPrice == adjPrice );
}

public void test_quantity( int quantity ) {
    test_quantity = quantity;
}

public void test_id( int id ) {
    test_id = id;
}

public void test_unitPrice( float unitPrice ) {
    test_unitPrice = unitPrice;
}

public void test_discount( float discount ) {
    test_discount = discount;
}

public void test_description( String description ) {
    test_description = description;
}

public void test_adjustedPrice( float total ) {
    test_adjPrice = total;
}

public void quantity( int quantity ) {
    this.quantity = quantity;
}

public void id( int id ) {
    this.id = id;
}
```

لیست ۱۴-۸ (ادامه)

```
    public void unitPrice( float unitPrice ) {
        this.unitPrice = unitPrice;
    }

    public void discount( float discount ) {
        this.discount = discount;
    }

    public void description( String description ) {
        this.description = description;
    }

    public void adjustedPrice( float total ) {
        this.adjPrice = total;
    }

    public String format() { // we're not testing formatter functionality
        return "NOT IMPLEMENTED";
    }
}
```

MockDisplayFormatter بسیار مشابه پیاده‌سازی اصلی است. با این حال که format() به صورت کامل پیاده‌سازی نمی‌شود، می‌توان دید که کلاس روالهایی برای تنظیم مقادیر مورد انتظار و نیز برای مقایسهٔ ورودیهای Item با این مقادیر فراهم می‌کند.

لیست ۱۴ ـ ۹ نشان می‌دهد که چگونه می‌توان از نمایش کاذب برای آزمایش کلاس Item استفاده کرد.

لیست ۱۴-۹ ItemTest.java

```
import junit.framework.TestCase;
import junit.framework.Assert;

public class ItemTest extends TestCase {

    private Item item;

    // constants for constructor values
    private final static int ID          = 1;
    private final static int QUANTITY    = 10;
    private final static float UNIT_PRICE  = 100.00f;
    private final static float DISCOUNT    = 5.00f;
    private final static String DESCRIPTION = "ITEM_TEST";

    protected void setUp() {
        item = new Item( ID, QUANTITY, UNIT_PRICE, DISCOUNT, DESCRIPTION );
```

لیست ۱۴-۹ (ادامه)

```
    }

    public void test_displayValues() {
        MockDisplayFormatter formatter = new MockDisplayFormatter();
        formatter.test_id( ID );
        formatter.test_quantity( QUANTITY );
        formatter.test_unitPrice( UNIT_PRICE );
        formatter.test_discount( DISCOUNT );
        formatter.test_description( DESCRIPTION );

        float adj_total = ( UNIT_PRICE * QUANTITY ) - ( DISCOUNT * QUANTITY );
        formatter.test_adjustedPrice( adj_total );

        item.display( formatter );

        formatter.verify();
    }

    public ItemTest( String name ) {
        super( name );
    }
}
```

روال test_displayValues() شیء ItemTest یک MockDisplayFormatter ایجاد می‌کند، ورودیهای مورد انتظار را تنظیم می‌کند، آن را به Itemارسال می‌کند و از قالب‌بندی برای ارزیابی ورودی استفاده می‌کند. روال verify() قالب‌بندی از کلاس Assert برای ارزیابی ورودی استفاده می‌کند.

اشیاء کاذب، مفهوم عمیقی دارند و ابزار قدرتمندی هستند. زیرا می‌توان آنها را برای انجام هر کاری برنامه‌ریزی کرد. می‌توان شیء کاذبی داشت که تعداد فراخوانی‌های یک روال را می‌شمارد، یا حجم ترافیک شبکه را مونیتور کند. اشیاء مجازی مراقبت‌ها و آزمونهایی را ممکن می‌کنند که در صورتی که شیء تمام اشیاء مورد نیاز خود را بسازد، ممکن نیستند.

مثلاً یک راه این است که این اشیاء، شیء مجازی برای متغیرها تعریف کنند. راه حل بهتر نوشتن کدی است که راحت‌تر قابل آزمون باشند.

شاید چنین کاری عقب‌نشینی به نظر برسد. معمولاً کسی برای این کد نمی‌نویسد، که آن را بتواند تست کند! مثال Itemدرس خوبی به ما می‌دهد! ساده‌تر نوشتن کلاسها به صورتی که قابل تست باشند، آنها را بیشتر شیءگرا می‌کند. در این مورد تغییر اشیاء وابسته، شیء را از کلاسهای دیگر مستقل‌تر کرده، یعنی اتصال‌پذیرترند!

تذکر

کلاسها را طوری بنویسید که به آسانی قابل آزمایش باشند. در زمان طراحی اشیاء کاذب را در ذهن داشته باشید. در این صورت کد شما شیءگراتر می‌شود.

تذکر

طوری کد بنویسید که اشیاء وابسته به آن ارسال شوند، نه اینکه از آنها در خود شیء متغیر تعریف شود. این تمرین ما را به کد خودبسنده رهنمون می‌کند.

با اینکه ()display به رابط ItemDisplayFormatter وابسته است، اما این وابستگی به پیاده‌سازی توسعه پیدا نمی‌کند زیرا خودش آن را ایجاد نمی‌کند.

تذکر

تذکراتی برای آزمون مؤثر:

- باید آزمونها را برای سرعت بیشتر تنظیم کرد. آزمونهای سریع باعث بازخورد مؤثرتر می‌شوند، لذا انگیزه استفاده از آنها بیشتر می‌شود.
- حالت‌ها و مراحل تست را همراه خود کلاس ترجمه و کامپایل کنید. در این صورت مراحل تست با کد هماهنگ خواهند بود.
- از ارزیابی دستی خودداری کنید، چون خطاساز است. در عوض از مکانیزم‌های خودکار استفاده کنید.
- طوری آزمون‌ها را بنویسید که به آنها نیاز دارید.

نوشتن کد استثناء

آزمون تنها کاری نیست که باید برای حصول اطمینان از کیفیت کدی که می‌نویسید انجام دهید. باید بیاموزید که فرق بین باگ و شرایط خطا را دریابید. چون این دو یکی نیستند!

مطمئناً می‌دانید که باگ چیست. شرایط خطا کمی متفاوت است. یک حالت خطا، یا استثناء، نقصی قابل پیش‌بینی است که تحت شرایط خاصی رخ می‌دهد. برای مثال فروشگاه اینترنتی را در نظر بگیرید. مشکلات شبکه خطایی است که همواره رخ می‌دهد. به جز در صورتی که خود برنامه باعث مشکلات شبکه شده باشد، این مشکلات باگ نیستند. به جای تلقی شرایط خطا به عنوان باگ، باید برای آن کد نوشت.

مثلاً اگر اتصال پایگاه داده قطع شود، باید سعی در اتصال مجدد کرد. اگر تلاش موفقیت‌آمیز نبود، باید مثلاً به کاربر اطلاع داد.

هر زبانی روش خاص خود را برای گزارش خطا دارد. Java و C++ از مکانیزمی به نام استثناها برای نشان دادن رخداد خطا استفاده می‌کنند. زبان‌هایی مانند C به کد برگشتی تکیه می‌کنند. هر زبانی که مورد استفاده قرار گیرد، باید برای کشف و رفع خطاها کدنویسی کنید.

استثناها در C++ و Java به صورتی مشابه کار می‌کنند. استثناها نوع دیگری از اشیاء هستند. کامپایلر Java شما را مجبور خواهد کرد که به آنها بپردازید و برای رخداد خطا فکری کنید.

یکی از روالهای کلاس URL به صورت زیر است:

```
public URLConnection openConnection () throws IOException
```

می‌بینید که روال اطلاعات خاصی دارد. روال بیان می‌کند که ممکن است IOException تولید کند، یعنی اینکه تحت شرایط عادی روال URLConnection برمی‌گرداند. اما در صورتی که خطایی رخ دهد، روال IOException برمی‌گرداند.

در Java، به این استثناها در بلوک‌های try-catch رسیدگی می‌شود. لیست ۱۴ ـ ۱۰ نشان می‌دهد که چگونه می‌توان با openConnection کار کرد.

لیست ۱۴-۱۰ پاسخگویی به یک استثناء

```
java.net.URL url = new java.net.URL("http://www.samspublishing.com/");
java.net.URLConnection conn;
try {
    conn = url.openConnection();
} catch ( java.io.IOException e ) { // an error has occured
    // log an error, write something out to the screen
    // do something to handle the error
}
```

وقتی که openConnection را فراخوانی می‌کنید، این کار را به صورت عادی انجام می‌دهید. اما باید این کار را درون بلوک try-catch انجام دهید. یا اینکه صریحاً بیان کنید که روال ممکن است IOException برگرداند.

اگر فراخوانی openConnection باعث بازگشت استثناء شود، conn مقداردهی نخواهد شد. اجرا ادامه خواهد یافت و سعی می‌شود اتصال برقرار شود. یک گزارش (Log) تولید می‌شود، یک پیام روی صفحه به نمایش در می‌آید یا اینکه استثنای دیگری برگردانده می‌شود.

اگر به استثناها رسیدگی نشود یا استثناء جدیدی برگردانده شود، پشتهٔ فراخوانی به آرامی پر می‌شود تا وقتی کامل شود یا کسی به استثناء رسیدگی کند.

آنچه اهمیت دارد برنامه‌ریزی برای این استثناها با استفاده از مکانیزم فراهم شده توسط زبان برنامه‌نویسی است. یعنی در زمان طراحی کلاس، باید به این خطاهای ممکن هم فکر کنید، آنها را با اشیاء استثناء مدل کنید و بگذارید روالها آنها را برگردانند.

نوشتن مستندات مؤثر

یک قدم دیگر در راه بهتر کردن کیفیت کار وجود دارد: آن را مستند کنید. اشکال زیادی از مستندسازی وجود دارد و هریک خواص خود را دارند.

کد منبع به عنوان مستندات

کد منبع و یا حتی آزمون واحد، نوعی از مستندسازی است. وقتی دیگران بخواهند کد شما را دستکاری کنند، خوانا بودن و واضح بودن آن مهم است، در غیر این صورت کسی نمی‌تواند از آن سر در آورد.

سرس کد مهمترین نوع مستندسازی است، زیرا تنها مستنداتی است که باید حتماً ایجاد و تنظیم شود.

رسم‌الخط کدنویسی

اولین گامی که باید در راه تبدیل کد خود به مستندات خوب بردارید، انتخاب یک رسم‌الخط کدنویسی و مداومت در استفاده از آن است. رسم‌الخط تعیین کننده شکل همه چیز از جمله نامگذاری متغیرها و روالها، طرز گذاشتن کروشه‌ها و قلابها و... است. خود انتخاب رسم‌الخط اهمیتی ندارد، آنچه مهم است این است که تیم کاری و حتی کل شرکت شما به آن رسم‌الخط وفادار بمانند. در این صورت، هر کسی می‌تواند هر قطعه کد را بردارد و از آن سر در آورد و حداقل با رسم‌الخط آن گیج نشود.

لیست ۱۴ ـ ۱۱ روشی برای معرفی کلاسهای Java ارایه می‌کند.

لیست ۱۴-۱۱ یک کلاس نمونه

```
public class <ClassName>
    extends <ParentClassName>
    implements <LIST OF INTERFACES>
{
    // public variables
    // protected variables
    // private variables
    // constants

    // public methods
    // protected methods
    // private methods
}
```

تذکر

نام کلاس همواره باید با حروف بزرگ شروع شود. نام روالها باید همواره با حرف کوچک آغاز شود. همچنین نام متغیرها.

هر نامی که از چند کلمه تشکیل شده باشد، باید با حروف بزرگ جدا شود، مثلاً ()someMethod و یا HappyObject. ثوابت همواره باید کلاً با حروف بزرگ نوشته شوند. متغیرهای عادی باید با حروف کوچک باشند.

توجه داشته باشید که این رسم‌الخط مختص Java است.

یک روش برای معرفی روال و عبارت if/else آشیانه‌ای:

```
public void method() {
    if (Conditional) {

    }else{

  }
}
```

درج توضیحات

هیچ چیز بیشتر از درج توضیحات (Comment) مناسب، به درک طرز کار کد کمک نمی‌کند. البته باید در آن هم حد نگاه داشت. زیاده‌روی در توضیح آن را بی‌معنی می‌کند.

مثلاً توضیح زیر، بی‌فایده است:

```
public void id (int id){
    this.id = id; //set id
}
```

درج توضیحات برای توضیح عملکرد کدهای پیچیده مفید است. توجه داشته باشید که قرار دادن این توضیحات جای نامگذاری مناسب روال و متغیرها را نمی‌گیرد.

نامها

نام کلاس‌ها، روالها و متغیرها همه باید معنی‌دار و گویا باشند. همچنین باید به صورت واضحی نوشته شوند. مثلاً در نامهای چندکلمه‌ای کلمه دوم را با حرف بزرگ آغاز کنید، یا بین کلمات از _(کاراکتر خط زیر یا under line) استفاده کنید. شیوه نامگذاری خود را استاندارد و یک شکل کنید و آن را جزو رسم‌الخط خود بیاورید.

سرفایل‌ها

همواره از اضافه کردن سرفایلهای (Header) روالها و کلاسها اطمینان حاصل کنید. سرفایل روال، شرح و فهرست آرگومان‌ها و توضیحات مقدار برگشتی و استثناها را در خود دارد. سرفایل می‌تواند حاوی پیش‌شرط‌ها هم باشد. سرفایل کلاس حاوی توضیحات، اطلاعات نگارش، لیست نویسندگان و تاریخچهٔ تجدیدنظرهای اعمال شده می‌باشد.

وقتی در Java برنامه می‌نویسید سعی کنید از امکانات Javadoc استفاده کنید. Javadoc کمک بزرگی برای نوشتن سرفایل است. زیرا به صورت خودکار مستندات مبتنی بر وب (web-based) برای کلاس تولید می‌کند.

نکته

اگر برای کد مستندات ایجاد کردید، یعنی متعهد شده‌اید که آن را با کد هماهنگ و به‌روز نگاه دارید. مستندات به هر صورتی که تهیه شده باشند، اگر به روز نباشند بی‌فایده‌اند و حتی می‌توانند گمراه کننده هم باشند.

خلاصه

امروز درباره آزمون کد و اهمیت آن در کیفیت کار بحث کردیم. کلاً چهار نوع آزمون وجود دارد:

- آزمون واحد
- آزمون مجتمع
- آزمون سیستم
- آزمون واپسگرد

در کارهای روزمره آزمون واحد اولین خط دفاع در برابر اشکالات برنامه‌نویسی است. علاوه بر این آزمون واحد، برنامه‌نویس را وا می‌دارد که طراحی خود را از منظر آزمون انجام دهد و مکانیزمی ایجاد کند که تصحیح و تغییر کد را آسانتر کند.

امروز همچنین رسیدگی به استثناها را بررسی کردیم که اهمیت زیادی دارد و در پایان به مستندسازی پرداختیم. تک تک این موارد کیفیت کدنویسی را بهبود می‌بخشند.

پرسشها و پاسخها

چرا برنامه‌نویس‌ها از آزمون نفرت دارند؟

فرهنگ نفرت از آزمون بین برنامه‌نویس‌ها وجود دارد! در بیشتر شرکت‌ها بخش تضمین کیفیت (QA) بخشی جدا از برنامه‌نویسان است. آنها وارد می‌شوند، آزمایش می‌کنند و خطاها را گزارش می‌کنند. این فرایند برنامه‌نویس را در موضع دفاعی قرار می‌دهد. خصوصاً اگر مدیر پروژه حجم عظیمی کار ترمیمی را

به او تحمیل کند. در این صورت آزمون به صورت تنبیه در می‌آید.

تلقی کار اضافه از آزمون هم معضل دیگری است. اکثر گروهها آزمون را تا انتهای کار به تعویق می‌اندازند، بنابراین آزمون به صورت کاری در می‌آید که در پس کار اصلی قرار می‌گیرد.

چرا اکثراً آزمونها توسط گروه تضمین کیفیت جداگانه‌ای انجام می‌شوند؟

برای جلوگیری از تست نکردن برخی از بخش‌های کار که احتمال وجود خطا در آنها داده می‌شود. خطایی که هر برنامه‌نویسی ممکن است مرتکب شود.

کارگاه

پرسشها

۱. **چگونه در برنامه اشکال ایجاد می‌شود؟**
۲. **یک حالت یا مرحله آزمون چیست؟**
۳. **دو راهی که می‌توان آزمون را بر پایه آنها انجام داد کدامند؟**
٤. **آزمونهای جعبه سفید و جعبه سیاه را تعریف کنید.**
۵. **چهار نوع آزمون کدامند؟**
۶. **آزمون واحد را تعریف کنید.**
۷. **نکته‌ای که در پس آزمون مجتمع و آزمون سیستم وجود دارد کدام است؟**
۸. **چرا نباید آزمون را تا انتهای کار به تأخیر انداخت؟**
۹. **چرا باید از ارزیابی دستی یا چشمی خودداری کرد؟ انتخاب دیگر چیست؟**
۱۰ **چهارچوب یعنی چه؟**
۱۱. **شیء کاذب چیست؟ چه کاربردی دارد؟**
۱۲. **تفاوت اشکال و شرایط خطا چیست؟**
۱۳. **چگونه از کیفیت کدی که نوشته‌اید مطمئن می‌شوید؟**

تمرین‌ها

۱. JUnit را تهیه کنید و cookstour را بخوانید.
۲. برای HourlyEmployee از فصل ۷، یک آزمون واحد بنویسید که روال calculatePay() را آزمایش کند.

روز ۱۵

Object Oriented Programming

ادغام تئوری و عمل

طی هفته اول سعی کردیم مفاهیم OOP را به شما منتقل کنیم. در هفته دوم فرایندی پیش پای شما قرار دادیم تا مطالب تئوری را به صورت عملی امتحان کنید. هفته سوم به شما نشان خواهد داد که چگونه دروس ارایه شده در هفته‌های اول و دوم را با یکدیگر ترکیب کنید تا یک پروژه OOP را به پایان برسانید. در این پروژه تمام مراحل توسعه نرم‌افزار و همه راههایی که باید طی شود تا پروژه به اتمام برسد را انجام خواهیم داد.

در درس امروز با بازی ورق بیست ویک (Balckjack) که بین عموم مردم فراگیر است، آشنا خواهید شد. در انتهای درس امروز مراحل مقدماتی تحلیل و طراحی و پیاده‌سازی بازی را انجام داده‌اید. لازم به توضیح است که ایجاد بازی فوق را به یکباره انجام نخواهید داد بلکه در طی هفته فرایندهای تکراری را انجام خواهید داد تا نوشتن بازی به اتمام برسد.

آنچه امروز خواهید آموخت:

- اعمال تحلیل و طراحی شیءگرا به یک بازی ورق واقعی
- استفاده از فرایندهای تکراری برای به دست آوردن نتایج سریع
- چشم‌پوشی از مراحل غیرضروری در توسعه نرم‌افزار
- جلوگیری از نفوذ مشخصه‌های رویه‌ای به درون برنامه

Blackjack

Blackjack یک بازی ورق است که هدف از آن به دست آوردن مجموع خال بیشتری نسبت به بازیکن دیگر است به شرط آنکه مجموع خالها از ۲۱ تجاوز نکند. در این هفته بازی فوق را با استفاده از ویژگیهای OOP که در طی دروس گذشته فراگرفتید و به زبان Java خواهیم نوشت. البته برخی از قسمتها به دلیل روشن بودن و اختصار در این هفته گنجانده نشده‌اند.

توجه نویسنده هیچ مسئولیتی در قبال از دست دادن زمان و اتلاف وقت به هنگام بازی با این برنامه را نمی‌پذیرد!

چرا Backjack؟

این سؤال ممکن است به ذهن خطور کند چرا Blackjack؟

هدف از نوشتن برنامه Blackjack آن نیست که شما را یک قمارباز حرفه‌ای بار بیاورد. بلکه دو دلیل عمده زیر برای نوشتن یک پروژه مقدماتی OOP می‌توان ذکر کرد:

- **عموم مردم حداقل با یکی از بازیهای ورق آشنایی دارند.**
- **از روی تجربه مشخص شده است که بازی Blackjack معیارهای OOP را بهتر برآورده می‌کند.**

به طور کلی عموم مردم با بازیهای ورق آشنایی دارند. یکی از مهمترین اجزای OOA و OOD شناخت کافی نسبت به مسأله است و واضح است که نمی‌توان شما را در خلال یک درس در مورد موضوعی خاص حرفه‌ای ساخت.

بازی ورق نیاز به حرفه‌ای بودن ندارد. در عوض تجربیات شخصی و یک کتاب راهنمای خوب و تعدادی از سایتهای وب چیزهای بسیار خوبی هستند تا شما بتوانید تمام مراحل تحلیل و طراحی را شخصاً انجام دهید.

به عنوان یک نتیجه، یک بازی ورق به مراتب قابل دسترس‌تر است تا یادگیری کامل یک حوزه ناآشنا. Blackjack و کلاً بازیهای ورق به خوبی در چارچوبهای تعیین شده در OOP قرار می‌گیرند. تعاملات بین بازیکنان، واسطه‌ها، دستان آنها و خودکارتها به شما کمک می‌کنند تا یک سیستم شیءگرایی که اشیاء مختلف آن شامل تعاملات زیادی است را ببینید.

چشم‌انداز

زمانی که پروژه جدیدی را شروع می‌کنید بهتر است در ابتدا هدف از انجام پروژه و یا چشم‌انداز آن را مشخص کنید. چشم‌انداز نقطه شروع خوبی برای آغاز کردن تحلیل است.

واژه جدید چشم‌انداز هدف و غرض اصلی سیستمی که شما موظف به ساخت آن هستید را در یک جا جمع می‌کند. در زیر چشم‌انداز سیستم Blackjack آمده است:

بازی Blackjack به یک بازیکن اجازه می‌دهد بر اساس قوانین کازینو اقدام به بازی نماید.

جایگزینی نیازمندیها

قبل از شروع به تحلیل، بهتر است نیازمندیها را بشناسیم. یکی از مهمترین مواردی که باید دانسته شود نحوه تعامل کاربر با سیستم است. بازی Blackjack به کاربر اجازه می‌دهد تا از طریق یک رابط کاربری گرافیکی و

یا از طریق خط فرمان با سیستم به تعامل بپردازد. با این حال در قدمهای اولیه ارتباط کاربر با سیستم از طریق خط فرمان خواهد بود. در ضمن این بازی باید با استفاده از زبان Java نوشته شود. قبل از شروع به کار لازم است مواردی که در پیاده‌سازی سیستم به شما تحمیل می‌گردد را خوب بشناسید تا خدای ناکرده در حین پیاده‌سازی با مشکلات لاینحل مو جه نشوید.

تحلیل مقدماتی Blackjack

در فصل نهم «مقدمه‌ای بر تحلیل شیءگرا» با OOA آشنا شدید. بر اساس مطالب ارایه شده در فصل ۹، می‌خواهیم فرایند توسعه تکراری را بر روی پروژه پیاده کنیم و در هر مرحله از تکرار فرایند را با تحلیل شروع کنیم.

قبل از شروع تحلیل بهتر است چشم‌انداز پروژه را بررسی کنیم:

بازی Blackjack به بازیکن اجازه می‌دهد بر اساس قوانین کازینو اقدام به بازی نماید.

با استفاده از چشم‌انداز فوق می‌توان فهرستی از سؤالات را ترتیب داد و بر اساس این سؤالات تحلیل را شروع کرد.

قوانین Blackjack

با توجه به چشم‌انداز مطرح شده، ممکن است این پرسش به ذهن بیاید که قوانین بازی کدامند؟ از قوانین ارایه شده در زیر می‌توان مدلی برای بازی استخراج کرد.

در این بازی یکی از بازیکنان ورقها را در دست دارد و برحسب نیاز به سایرین ورق می‌دهد، این بازیکن را پخش کننده یا واسط می‌نامیم. هدف از بازی جمع‌آوری مجموع خالهایی است که بیش از مجموع ورقهای پخش‌کننده باشد و عدد آن هم از ۲۱ تجاوز نکند. هر ورق یا کارت شماره‌ای بین ۱ تا ۱۱ دارد. آس برحسب نیاز دارای ارزش ۱ یا ۱۱ بوده و کارت‌های شماره‌دار دارای ارزشی هستند که عدد رویشان نشان می‌دهد. در ضمن کارتهای تصویردار دارای ارزش ۱۰ هستند. شکل ۱۵ ـ ۱ مثالهای متنوعی را ارایه داده است.

اجازه دهید به هریک از قسمتهای بازی نظری بیاندازیم.

شرط‌بندی

قبل از آنکه واسطه کارتها را پخش کند، هریک از بازیکنان باید شرط‌بندی کنند. میزان شرط‌بندی مقداری بین ۲۵ تا ۵۰ دلار در هر بازی است.

پخش کارتها

پس از آنکه هریک از بازیکنان پول شرط‌بندی را روی میز گذاشتند. واسط شروع به پخش کارتها می‌کند. پخش کارتها با اولین فرد شروع شده و واسط روبروی هریک از بازیکنان کارتی را قرار می‌دهد. واسط کارتهای بازیکنان را رو به بالا و کارت خود را رو به پایین قرار می‌دهد. کارت رو به پایین واسط، کارت شاخص نامیده می‌شود.

پخش کارتها تا آنجایی ادامه پیدا می‌کند که روبروی هر بازیکن و منجمله خودش دو کارت قرار گرفته باشد.

بازی

نحوه بازی بر حسب آنکه کارت واسط چه چیزی باشد، متفاوت است. اگر کارت واسط دارای ارزش ۱۰ باشد و یا آنکه از نوع آس (با ارزش ۱۱) باشد، واسط می‌باید کارت شاخص خود را چک کند. در صورتی که کارت شاخص باعث شود که مجموع امتیازات به ۲۱ (و به عبارت دیگر Blackjack) برسد، بازی به طور خودکار تمام خواهد شد. اما اگر مجموع امتیازات به ۲۱ نرسد، بازی ادامه پیدا می‌کند.

اگر کارتی که واسط بیرون کشیده، دارای ارزش ۱۰ نباشد و یا نوع آن آس نباشد، بازی به اولین بازیکن بعدی منتقل می‌شود. در صورتی که مجموع امتیازات بازیکن به ۲۱ برسد، بازیکن دارای Blackjack بوده و بازیکن بعدی باید بازی را ادامه دهد. اما اگر امتیازات بازیکن به ۲۱ نرسیده باشد، دو انتخاب برای بازیکن وجود دارد: توقف و یا ادامه دادن.

ادامه دادن: اگر بازیکن از کارتی که کشیده است، راضی نباشد می‌توان کارت دیگری بکشد. این کار را می‌تواند تا آنجایی ادامه دهد که مجموع امتیازاتش بیشتر از ۲۱ شود و یا آنکه بخواهد که کشیدن کارتها را متوقف کند.

توقف: اگر بازیکن از کارتهایی که کشیده است، رضایت داشته باشد، می‌تواند کار را متوقف کند.

پس از توقف کار، ادامه بازی به بازیکن بعدی منتقل می‌شود. این کار آنقدر ادامه پیدا می‌کند تا همه بازیکنها بازی کرده باشند.

پس از آن که همه بازیکنها بازی کردند، نوبت به واسط می‌رسد. پس از آنکه واسط کارتی را بیرون کشید، بازی برای تعیین برنده و بازنده متوقف می‌شود.

تعیین برندگان

پس از آنکه واسط کارتی را بیرون کشید، بازی متوقف می‌شود تا برندگان و بازندگان مشخص شوند. در این قسمت بازندگان پولهایی را که شرط‌بندی کرده بودند، از دست می‌دهند.

هر بازیکنی که مجموع امتیازاتش بیشتر از واسط باشد برنده و آنهایی که کمتر از واسط هستند بازنده هستند. بازیکنانی که امتیازشان به اندازه امتیاز واسط است، بدون آنکه پولی از دست داده باشند، در بازی می‌مانند.

در صورتی که بازیکنی امتیازش Blackjack باشد و واسط Blackjack نداشته باشد، به اندازه ۱/۵ برابر پولی که شرط‌بندی کرده است می‌برد. برای مثال اگر ۱۰۰ دلار شرط‌بندی کرده باشد، مقدار ۱۵۰ دلار (۱/۵ × ۱۰۰) دریافت خواهد کرد.

چند نکته

لازم است نکات چندی در مورد بازی Blackjack را خاطرنشان سازم:

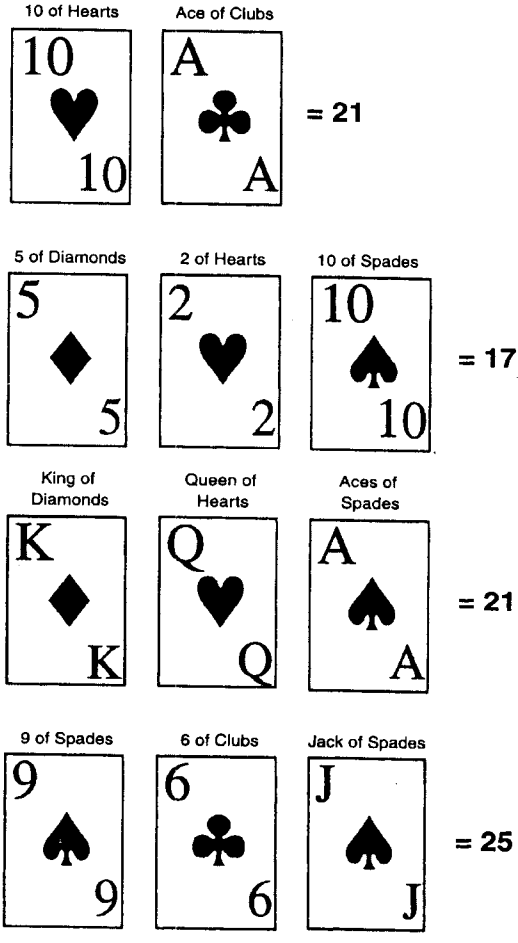
میز بازی: شامل ۴ دسته ۵۲ عددی از کارتهای استاندارد است. این ۴ دسته تشکیل یک دسته بزرگ می‌دهند.

تعداد بازیکنان: بین ۱ تا ۷ بازیکن می‌توانند Blackjack را بازی کنند.

پس از آنکه بازی دو دور گشت، بازیکن می‌تواند دو کارت تقاضا کند. در این حالت باید پول شرط‌بندی را نیز دو برابر کند. با این کار، یک کارت بیشتر از واسط دریافت می‌کند. (Doubling Down)

بیمه: اگر کارتی که واسط بیرون می‌کشد آس باشد، بازیکن اختیار دارد که پولی را به عنوان بیمه

شکل ۱۵ ـ ۱
دسته کارتهای نمونه در بازی Blackjack



(Insurance) در وسط قرار دهد. این پول باید برابر و یا نصف پول اولیه شرط‌بندی باشد. اگر کارت شاخص مجموع امتیازات را به ۲۱ نرساند بازیکن پول بیمه را از دست خواهد داد.

تقسیم زوجها: بازیکنی که به هنگام تقسیم کارتها (دو کارت اول)، کارتهای هم‌ارزش دریافت کرده باشد، گفته می‌شود که یک زوج کارت دارد. در این حالت بازیکن اختیار دارد که این زوج را به دو دست تقسیم کند. در این صورت واسط به هر دست یک کارت اضافی دیگر می‌دهد. بازیکن می‌تواند هر زوج راکه از تقسیم متوالی به وجود می‌آید (به غیر از آس) به دو دسته تقسیم کند. این نکته نیز مهم است که اگر به واسطه تقسیم مجموع ارزش کارتها به ۲۱ برسد، این مقدار ارزش واقعی به حساب نمی‌آید. با انجام تقسیم، بازیکن مختار است که به کشیدن کارتها ادامه دهد و یا آنکه متوقف شود.

تشخیص عاملها

دو دلیل مهم برای انجام این تحلیل وجود دارد. در هنگام تحلیل، مدل موارد کاربردی را خواهید ساخت: توضیح آنکه کاربران چگونه از سیستم استفاده خواهند کرد. همچنین در خلال تحلیل، مدل دامنه را می‌سازید: توضیح واژه‌های اصلی (و کارکرد) سیستم.

اولین قدم در تعریف موارد کاربردی، تعریف عاملهایی است که از سیستم استفاده خواهند کرد. با مراجعه به آخرین قسمت دو عامل مهم بازی Blackjack شناسایی می‌شوند، بازیکنان و شخص واسط یا پخش‌کننده.

این دو عامل پرسش «چه کسی در ابتدا از سیستم استفاده می‌کند» را پاسخ خواهند گفت. از طریق موارد کاربردی، می‌توان تعریف کرد که چگونه عاملها از سیستم استفاده خواهند کرد.

ساخت لیستی مقدماتی از موارد کاربردی

یک راه مؤثر برای ایجاد موارد کاربردی مقدماتی طرح این سؤال است که هریک از عاملها چه کاری می‌توانند انجام دهند.

بازیکنان موارد زیر را می‌توانند انجام دهند:

۱. قرار دادن پول شرط‌بندی (Bet)
۲. ادامه دادن (Hit)
۳. متوقف شدن (Stand)
٤. بردن/باختن
۵. دریافت Blackjack
۶. قرار دادن پول بیمه (Insurance)
۷. تقسیم زوجها (split)
۸. کشیدن یا تقاضای دو کارت
۹. تصمیم به بازی دوباره
۱۰. خروج از بازی

شخص واسط نیز می‌تواند اعمال زیر را انجام دهد:

۱. پخش کارتها
۲. ختم بازی
۳. ادامه دادن
٤. متوقف کردن
۵. بردن/باختن
۶. دریافت Blackjack

البته موارد کاربردی بیشتری می‌توان تعریف کرد با این حال لیست ارایه شده موارد اولیه لازم جهت بازی را در بر دارد.

طراحی تکرارها

در فصل نهم فرایند توسعه تکرارها معرفی شد. با دنبال کردن یک فرایند تکرار می‌توانید اسکلت بازی Blackjack را بسازید. در هر تکرار تعدادی از قابلیتهای جدید به بازی اضافه خواهد شد.

کلید طراحی فرایندهای تکراری درواقع طراحی تکراری است که می‌توان بهترین شروع را با آن داشت. می‌توان در هر تکرار طرح را دوباره بررسی کرد و قابلیتهای جدید را به آن افزود. اما شروع کردن پایه‌ای و اصولی پروژه از ابتدا باعث می‌شود که تکمیل پروژه روند صحیحی داشته باشد و اهدافیکه پروژه دنبال

می‌کند، بهتر قابل دسترسی قرار گیرد.

عموماً طراحی تکرارها با دسته‌بندی کردن موارد کاربردی بر اساس اهمیت آنها صورت می‌گیرد. بهتر است دسته‌بندی کردن موارد کاربردی توسط خود مشتری صورت گیرد. در مورد بازی Blackjack موارد کاربردی بر اساس نقشی که در بازی ایفا می‌کنند، رتبه‌بندی می‌شوند. مورد کاربردی را انتخاب کنید که اساساً برای بازی الزامی است. اینگونه موارد کاربردی را اول از همه انتخاب کنید. باقی موارد کاربردی را در تکرارهای بعدی انجام خواهیم داد.

برای بازی فوق چهار تکرار مهم موجود است.

تکرار اول: بازی کردن مقدماتی

در تکرار اول یک بازی ابتدایی را خواهیم ساخت. در این تکرار موارد کاربردی زیر پیاده‌سازی خواهند شد:

۱. ادامه دادن
۲. متوقف شدن
۳. بردن/باختن

موارد کاربردی واسط:

۱. پخش کارتها
۲. ادامه دادن
۳. متوقف شدن
٤. بردن/باختن

در انتهای تکرار اول، بازی‌ای خواهیم داشت که در خط فرمان (Command Prompt) اجراء خواهد شد. بازی شامل دو شرکت‌کننده خواهد بود: واسط یا پخش‌کننده و یک بازیکن. واسط کارتها را پخش کرده و به هریک از بازیکنان اجازه بازی می‌دهد تا آنجا که بازیکن برنده شود، بازنده شود و یا بازی را متوقف کند. پس از آنکه همه بازیکنان بازی کردند، بازی خاتمه می‌یابد.

تکرار دوم: قوانین

در تکرار دوم، قوانین به بازی اضافه می‌شود. در این تکرار موارد کاربردی زیر تعریف مجدد شده و یا پیاده‌سازی می‌شوند.

موارد کابردی بازیکنان:

۱. دریافت Blackjack

واسط می‌تواند:

۱. اعلام نتیجه بازی (تشخیص برندگان، بازندگان و آنهایی که نبرده‌اند و یا نباخته‌اند)
۲. دریافت Blackjack

در انتهای تکرار دوم، همه چیز از تکرار اول تا موارد کاربردی ذکر شده، پیاده‌سازی می‌شوند. به اضافه آنکه در بازی کسی که بازی را برده است و یا آنکه بازی را متوقف کرده و یا باخته است، مشخص می‌شود.

تکرار سوم: شرط‌بندی

در تکرار سوم شرط‌بندی ابتدایی و تقاضای دو کارت (Double down) به بازی اضافه می‌شود. در این تکرار موارد کاربردی زیر اضافه خواهند شد:

موارد کاربردی بازیکنان:

۱. قرار دادن پول شرط‌بندی
۲. کشیدن دو کارت

موارد کاربردی واسط:

۱. اعلام نتیجه بازی (برای دریافت پول شرط‌بندی شده)

در انتهای تکرار سوم، همه چیز از تکرار اول و دوم تا موارد کاربردی ذکر شده، پیاده‌سازی خواهند شد. به اضافه آنکه اجازه شرط‌بندی مقدماتی را نیز می‌دهد.

تکرار چهارم: رابط کاربری (UI)

در تکرار چهارم تغییراتی در رابط کاربری خط فرمان ایجاد شده و رابط کاربری گرافیکی (GUI) ساخته خواهد شد. موارد کاربردی زیر در این مرحله بررسی خواهند شده:

موارد کاربردی بازیکنان:

۱. تصمیم برای بازی مجدد
۲. خروج از بازی

نکته

برای برآورده کردن اهداف پروژه، دو مورد بیمه (Insurance) و جداسازی زوجها نادیده گرفته شده‌اند. این دو مورد برای تمرین به خواننده واگذار می‌شود. بازی Blackjack برنامه‌ای است شامل تغییرات زیاد. با فرض فوق قابلیتهای بیمه و جداسازی زوجها در برنامه راه ندارد. در مورد این تغییرات می‌توان کتاب مفصلی نوشت!

تکرار اول: بازی کردن مقدماتی

درس امروز در تکرار اول یادگیری و اعمال مطالب گفته شده در مورد بازی کارتی Blackjack است. در انتهای درس امروز اسکلتی ابتدایی از بازی فوق ساخته خواهد شد.

نکته

قبل از هر بخش، بهتر است کتاب را کنار گذاشته و خودتان سعی کنید مراحل تحلیل، طراحی و پیاده‌سازی را انجام دهید. پس از آنکه موارد فوق را شخصاً تجربه کردید، به کتاب بازگشته و هر بخش را مطالعه کرده و آن را با کار خود مقایسه کنید. در هر مرحله راه حل خود را با آنچه که در کتاب ارایه شده است مقایسه کرده و در مورد آن قضاوت کنید. به خاطر داشته باشید راه حلهای زیادی برای پیاده‌سازی این بازی وجود دارد. مراقب باشید راه حل شما بر اساس آموخته‌های شما از OOP باشد.

تحلیل Blackjack

تکرار امروز شامل موارد کاربردی زیر خواهد بود:

موارد کاربردی بازیکنان:

۱. ادامه دادن
۲. متوقف شدن
۳. بردن/باختن

موارد کاربردی واسط:

۱. پخش کارتها
۲. ادامه دادن
۳. متوقف شدن
٤. بردن/باختن

پس از تعریف موارد کاربردی باید مدل اولیه‌ای از دامنه را ساخت و طراحی را شروع کرد.

تعریف مجدد موارد کاربردی

اجازه دهید با مورد کاربردی مربوط به پخش‌کننده شروع کنیم چراکه عملکرد وی باعث شروع بازی می‌شود. اولین کار توصیف مورد کاربردی در یک پاراگراف است:

واسط از اولین بازیکن شروع کرده و اقدام به پخش کارتها می‌کند و به خودش ختم می‌کند و این کار را ادامه می‌دهد. کارتهای بازیکنان رو به بالا و کارت واسط رو به پایین قرار می‌گیرد. پس از اتمام پخش کارتها، بازی با رو کردن و تقاضای کارتها آغاز می‌شود.

- پخش کارتها:

۱. واسط برای هر بازیکن و همچنین خودش یک کارت قرار می‌دهد به نحوی که کارت رو به بالا باشد.
۲. واسط کارت دومی را برای همه بازیکنان (غیر از واسط) رو به بالا قرار می‌دهد.
۳. واسط کارتی رو به پایین به خودش تخصیص می‌دهد.

- شرایط قبل:
  - بازی جدید
- شرایط بعد:
  - تمام بازیکنان و خود واسط در یک دست، دو کارت دارند.

موارد کاربردی مرتبط به ادامه کشیدن کارتها و یا توقف در کشیدن کارتها به صورت سر هم انجام می‌گردند. با مورد کاربردی کشیدن کارتها شروع می‌کنیم:

در صورتی که بازیکن کارت را بیرون بکشد و از نتیجه راضی نباشد، می‌تواند اقدام به کشیدن دوباره نماید.

- ادامه دادن کشیدن کارتها

۱. بازیکن تصمیم می‌گیرد که به دلیل عدم رضایت از دسته کارت خود اقدام به ادامه کشیدن کارتها یا

تقاضای کارت کند.

۲. بازیکن از واسط درخواست کارت جدیدی می‌کند.

۳. بازیکن تصمیم می‌گیرد تا دریافت کارتها را ادامه دهد و یا آنکه در صورتی که جمع ارزش کارتهایش از ۲۱ کمتر و یا مساوی آن بود، متوقف شود.

- شرایط قبل:
  - بازیکن دسته کارتی دارد که جمع ارزش آنها کمتر و یا مساوی ۲۱ است.
- شرایط بعد:
  - کارت جدیدی به دسته کارتهای بازیکن اضافه می‌شود.
  - راههای دیگر: باخت بازیکن

کارت جدید باعث می‌شود که جمع ارزش کارتهایش از ۲۱ بالا رود. در این صورت بازیکن بازی را واگذار کرده است. در این صورت بازیکن بعدی بازی را ادامه می‌دهد.

توقف بازیکن مورد کاربردی ساده‌ای است:

احساس بازیکن آن است که دسته کارت موجود برای وی کفایت می‌کند و در نتیجه در همانجا متوقف می‌شود.

- توقف بازیکن:

۱. بازیکن به دلیل آنکه از کارتهای موجود رضایت دارد، متوقف می‌شود.

- شرایط قبل:
  - بازیکن دسته کارتی دارد که جمع ارزش آنها کمتر و یا مساوی ۲۱ است.
- شرایط بعد:
  - دور بازیکن خاتمه می‌یابد.

در اینجا روشن می‌شود که باخت بازیکنان /واسط خود به تنهایی یک مورد کاربردی محسوب نمی‌شود بلکه نتیجه دیگر اعمال به حساب می‌آید. درواقع بازیکن و یا واسط هیچگاه عمل باختن را انجام نمی‌دهند.

با حذف مورد کاربردی باخت، تنها موارد کاربردی ادامه دادن و متوقف شدن برای بازیکن و شخص واسط باقی می‌ماند.

در صورتی که ارزش کارتهای واسط کمتر از ۱۷ باشد، واسط باید به کشیدن کارتها ادامه دهد. در صورتی که پس از کشیدن کارت، نبازد و همچنان مجموع ارزش کارتهایش کمتر از ۱۷ باشد، دوباره کارت دیگری را باید بکشد. در صورتی که مجموع ارزش کارتها مساوی ۱۷ و یا بیشتر از آن شد، واسط متوقف می‌شود. با توقف و یا باخت واسط بازی خاتمه می‌یابد.

- ادامه دادن (برای واسط):

۱. در صورتی که دسته کارتهای واسط ارزشی کمتر از ۱۷ داشته باشد، به کشیدن کارت ادامه می‌دهد.

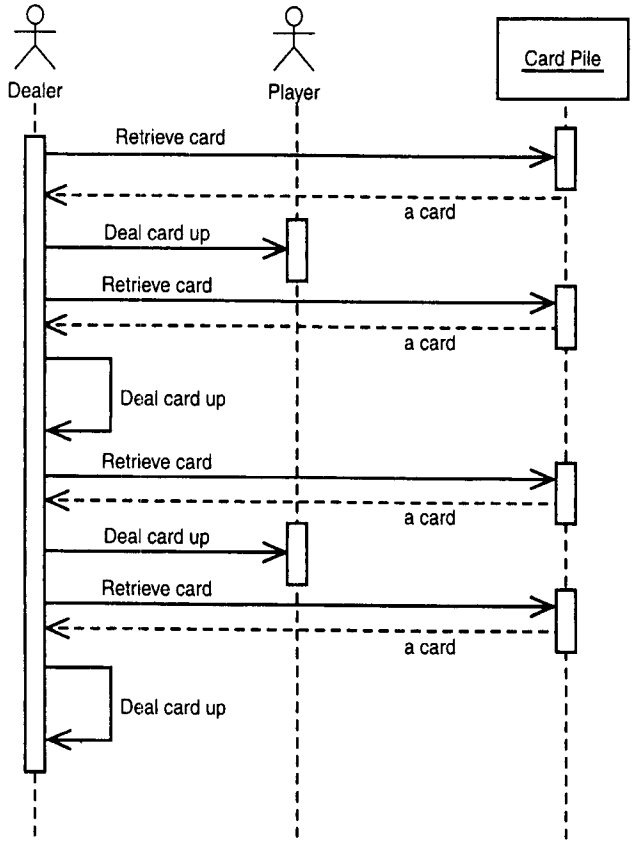
۲. کارت جدیدی به دسته کارتهای واسط اضافه می‌شود.

۳. در صورتی که جمع ارزش کارتها کمتر از ۱۷ باشد، کشیدن کارتها ادامه پیدا می‌کند.

- شرایط قبل:
  - واسط دسته کارتهایی دارد که مجموع ارزش آنها کمتر از ۱۷ است.

- شرایط بعد:
  - کارت جدیدی به دسته کارتهای واسط اضافه می‌شود.
  - بازی خاتمه می‌یابد.
- راههای دیگر:

باخت واسط: کارت جدید باعث می‌شود که مجموع ارزش کارتهای واسط بیش از ۲۱ شود. در این صورت واسط می‌بازد.



شکل ۱۵ ـ ۲
نمودار سلسله مراتبی برای مورد کاربردی کارتهای پخش شده

- راههای دیگر:

توقف واسط: کارت جدید باعث می‌شود که مجموع ارزش کارتهای واسط مساوی یا بیشتر از ۱۷ شود. در این صورت واسط متوقف می‌شود.

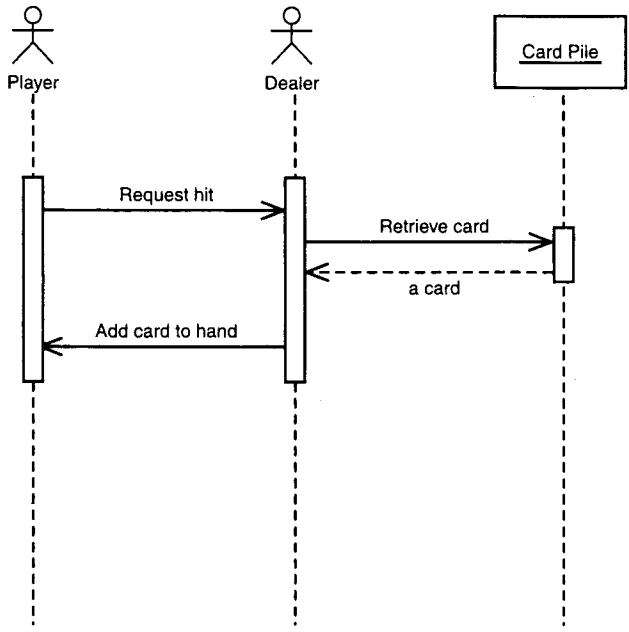
- توقف واسط:

٤. دسته کارتهای واسط دارای ارزشی برابر ۱۷ و یا بیشتر از آن هستند و واسط را متوقف می‌کنند.

- شرایط بعد:
  - بازی خاتمه می‌یابد.

مدلسازی موارد کاربردی

برای تکرار اول، موارد کاربردی، ساده هستند. مدلهای موارد کاربردی بیشتر از آنکه ارزش داشته باشند،

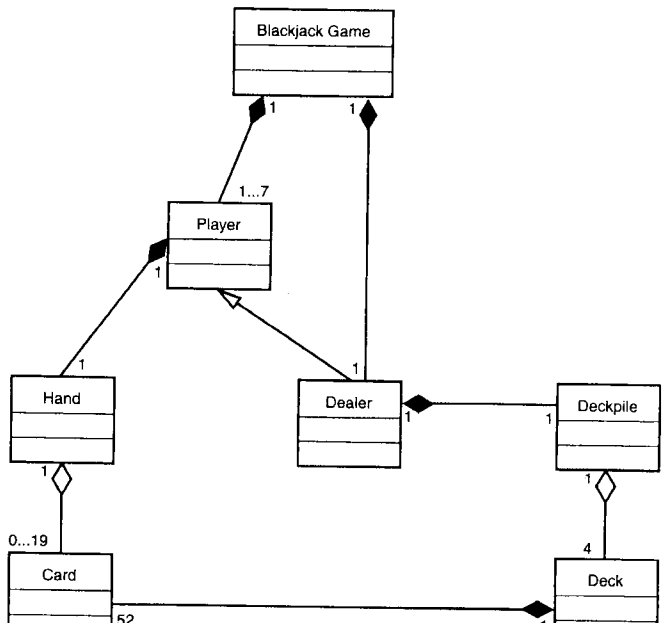


شکل ۱۵ ـ ۳
نمودار سلسله مراتبی برای موارد کاربردی کشیدن کارتها توسط بازیکن

دردسرساز هستند. بنابراین از آنها در اینجا صرفنظر می‌کنیم.

تعامل بین واسط و بازیکنان جالب‌تر است. شکل ۱۵ ـ ۲ سلسله مراتب رخدادهایی که بین مورد کاربردی کارت‌های پخش شده اتفاق می‌افتد را نشان می‌دهد. شکل ۱۵ ـ ۳ سلسله مراتب رخدادهایی راکه با کشیدن و ادامه دادن کارتها توسط بازیکن اتفاق می‌افتد را نشان می‌دهد.

شکل ۱۵ ـ ۴
مدل دامنه Blackjack



Blackjack	
create players	Player
create hands	Hand
create dealer	Dealer
create Deckpile	Deck, Deckpile
connect all of the players, dealer, and console together	Console, Dealer, Player, Deckpile
start game	Dealer

شکل ۱۵ ـ ۵
کارت CRC بازی Blackjack

مدلسازی دامنه

با استفاده از موارد کاربردی به عنوان پایه‌ای برای مدلسازی دامنه، می‌توان ۷ نوع شیء دامنه را از یکدیگر ایزوله کرد: Deckpile، Deck، Card، Player، Dealer، BlackjackGame و Hand. شکل ۱۵ ـ ۴ نمودار مدل دامنه حاصله را نشان می‌دهد.

طراحی Blackjack

با اعمال طراحی شیءگرا به تحلیل ارایه شده، مدل کلاسهای پایه در طراحی، مسئولیتهای آنها و تعریف هریک و اینکه تعاملات آنها با دیگر اشیاء چگونه است مشخص می‌شود. سپس طراحی را گرفته و می‌توان بر روی پیاده‌سازی آن کار کرد.

کارتهای CRC

مدل دامنه برای ارایه فهرست اولیه‌ای از اشیاء شروع خوبی است. با این فهرست از اشیاء می‌توانید از کارتهای CRC برای مشخص کردن مسئولیتهای مختلف و همکاریهای اشیاء با یکدیگر استفاده کنید.

Deck	
add itself to the Deckpile	Deckpile
construct 52 cards	Card

شکل ۱۵ ـ ۶
کارت CRC مجموعه کارتهای بازی

شکل ۱۵ ـ ۷
کارت CRC هریک
ازکارتها

Card	
hold a suit	Suit
hold a rank	Rank
display itself	Rank, Suit, Console
hold and toggle a face state: up or down	

نکته قبل از آنکه ادامه دهید، تمرین خوبی است اگر فهرستی ازکارتهای CRC را خودتان تولید کنید.

شکلهای ۱۵ ـ ۵ تا ۱۱ ـ ۱۵ خروجیهای ممکن از یک کارت CRC را نشان می‌دهند.

شکل ۱۵ ـ ۸
کارت CRC بازیکن

Player	
hold a hand	Hand
add cards to hand	Hand, Card
decide whether to hit or stand	Hand
inform dealer when done playing	Dealer
update observers	Player Listener
display itself	Hand, Console
detect busted hand	Hand

شکل ۱۵ ـ ۹
کارت CRC واسط

Dealer inherits from Player	
hit a player	Player
pass the turn to the next player/self	Players/self
track players in game	Player
start a new game	
deal cards	Deckpile, Card, Player, self
decide whether to hit or stand	
tell players when they can play	Player/self

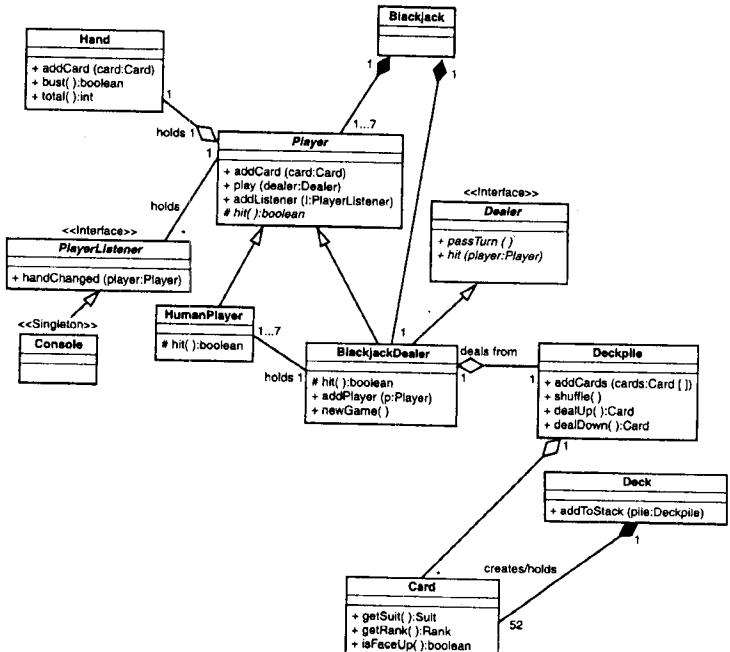
شکل ۱۵ ـ ۱۰
کارت CRC دسته
کارتهای بازیکنان

Hand	
hold onto cards	Card
add cards to itself	Card
reset itself	
turn all cards over	Card
display itself	Card, Console
calculate its total	Card, Rank
detect bust	Card

شکل ۱۵ ـ ۱۱
کارت CRC یک
مجموعه کارت

Deckpile	
accept cards to hold onto	Card
shuffle the cards	Card
deal cards up	Card
deal cards down	Card
reset itself	

شکل ۱۵ ـ ۱۲
مدل کلاس Blackjack



رابط کاربری خط فرمان

در فصل ۱۳، «شیءگرایی برنامه‌نویسی رابط کاربر» الگوی طراحی MVC معرفی شد. بازی Blackjack نیز از این الگو بهره خواهد برد. به عنوان یک نتیجه برای طراحی، نیاز است مکانیزم گیرندگی را به اشیاء بازیکن (Player) و کنسول (Console) برای نمایش وضعیت بازیکنان و دریافت ورودیهای کاربر اضافه کنید.

از آنجا که تنها یک کنسول وجود دارد، می‌توان شیء Console را کاندیدای مناسبی برای الگوی یک کارت دانست.

مدل Blackjack

نه کلاس و دو رابط مدل کامل کلاس Blackjack را می‌سازند. شکل ۱۵ ـ ۱۲ مدل را نمایش می‌دهد. بخش بعدی توضیحات کامل پیاده‌سازی این مدل را در بر می‌گیرد.

پیاده‌سازی

بخشهای بعدی پیاده‌سازی قسمتهای عمده‌ای از مدل نمایش داده شده در شکل ۱۵ ـ ۱۲ را ارایه می‌کند.

توجه

تمام سرس کدها از طریق وب سایت کتاب قابل دسترسی است. از سایت www.samspublishing.com بازدید کرده و به دنبال کتابی با شابک (ISBN) 0672321092 بگردید. سپس بر روی لینک سرس کد کلیک کنید.

کارت (Card)

کلاس Card آنگونه که در فصل ۱۲، «الگوهای پیشرفته طراحی» آمده است، پیاده‌سازی گردیده با این تفاوت که Rank اندکی تغییر پیدا کرده است. لیست ۱۵ ـ ۱ پیاده‌سازی جدید Rank را نشان می‌دهد.

لیست ۱۵-۱ Rank.java

```
import java.util.Collections;
import java.util.List;
import java.util.Arrays;

public final class Rank {

    public static final Rank TWO   = new Rank( 2, "2" );
    public static final Rank THREE = new Rank( 3, "3" );
    public static final Rank FOUR  = new Rank( 4, "4" );
    public static final Rank FIVE  = new Rank( 5, "5" );
    public static final Rank SIX   = new Rank( 6, "6" );
    public static final Rank SEVEN = new Rank( 7, "7" );
    public static final Rank EIGHT = new Rank( 8, "8" );
    public static final Rank NINE  = new Rank( 9, "9" );
    public static final Rank TEN   = new Rank( 10, "10" );
    public static final Rank JACK  = new Rank( 10, "J" );
    public static final Rank QUEEN = new Rank( 10, "Q" );
```

لیست ۱۵-۱ (ادامه)

```
    public static final Rank KING  = new Rank( 10, "K" );
    public static final Rank ACE   = new Rank( 11, "A" );

    private static final Rank [] VALUES =
            { TWO, THREE, FOUR, FIVE, SIX, SEVEN,
              EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE };

    // provide an unmodifiable list to loop over
    public static final List RANKS =
        Collections.unmodifiableList( Arrays.asList( VALUES ) );

    private final int    rank;
    private final String display;

    private Rank( int rank, String display ) {
        this.rank = rank;
        this.display = display;
    }

    public int getRank() {
        return rank;
    }

    public String toString() {
        return display;
    }
}
```

Deck و Deckpile

کدهای مربوط به Deck نسبت به آنچه که در فصل ۱۲، نشان داده شده است، تغییرات قابل ملاحظه‌ای کرده است. لیست ۱۵ ـ ۲ پیاده‌سازی جدید را نشان می‌دهد.

لیست ۱۵-۲ Deck.java

```
import java.util.Iterator;
import java.util.Random;

public class Deck {

    private Card [] deck;
    private int index;

    public Deck() {
```

لیست ۱۵-۲ (ادامه)

```
        buildCards();
    }
    public void addToStack( Deckpile stack ) {
        stack.addCards( deck );
    }

    private void buildCards() {

        deck = new Card[52];

        Iterator suits = Suit.SUITS.iterator();

        int counter = 0;
        while( suits.hasNext() ) {
            Suit suit = (Suit) suits.next();
            Iterator ranks = Rank.RANKS.iterator();
            while( ranks.hasNext() ) {
                Rank rank = (Rank) ranks.next();
                deck[counter] = new Card( suit, rank );
                counter++;
            }
        }
    }
}
```

Deck به سادگی Cardهای خود را ساخته سپس خود را به Deckpile اضافه می‌کند. لیست ۱۵ ـ ۳، Deckpile و پیاده‌سازی آن را نشان می‌دهد.

لیست ۱۵-۳ Deckpile.java

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Random;

public class Deckpile {

    private ArrayList stack = new ArrayList();
    private int index;
    private Random rand = new Random();

    public void addCards( Card [] cards ) {
        for( int i = 0; i < cards.length; i ++ ) {
            stack.add( cards[i] );
        }
    }
```

لیست ۱۵-۳ (ادامه)

```
public void shuffle() {
    reset();
    randomize();
    randomize();
    randomize();
    randomize();
}

public Card dealUp() {
    Card card = deal();
    if( card != null ) {
        card.setFaceUp( true );
    }
    return card;
}

public Card dealDown() {
    Card card = deal();
    if( card != null ) {
        card.setFaceUp( false );
    }
    return card;
}

public void reset() {
    index = 0;
    Iterator i = stack.iterator();
    while( i.hasNext() ) {
        Card card = (Card) i.next();
        card.setFaceUp(false);
    }
}

private Card deal() {
    if( index != stack.size() ) {
        Card card = (Card) stack.get( index );
        index++;
        return card;
    }
    return null;
}

private void randomize() {
    int num_cards = stack.size();
```

لیست ۱۵-۳ (ادامه)

```
        for( int i = 0; i < num_cards; i ++ ) {
            int index = rand.nextInt( num_cards );
            Card card_i = (Card) stack.get( i );
            Card card_index = (Card) stack.get( index );
            stack.set( i, card_index );
            stack.set( index, card_i );
        }
    }
}
```

Deckpile به راحتی کارتها را بر زده و سپس آنها را پخش می‌کند. برخلاف کدهای Deck اولیه، Deckpile مرجعی از همه کارتهای بازگردانده شده را نگه می‌دارد. از این طریق به راحتی کارتها را دریافت کرده و می‌توان خود را سازماندهی کند. اگرچه این کار مدل کاملی برای آنچه در دنیای واقعی اتفاق می‌افتد، نیست ولی مدیریت کارتها را به مراتب ساده‌تر می‌کند.

فهم دلیل تغییرات رخ داده، بسیار مهم است. هر دوی Deck و Deckpile تنها رفتارهایی که بازی به آنهااحتیاج دارد، پیاده‌سازی کرده‌اند. این کلاسها قابلیتهای جدیدی را ایجاد نکرده‌اند. این امر نیز امکان‌پذیر نیست که هر نوع قابلیتی را اضافه کرد. تنها قابلیتهایی را اضافه کنید که به آنها نیاز دارید.

اگر بخواهید تمام قابلیتها را پیاده‌سازی نمایید، هیچگاه نخواهید توانست پیاده‌سازی کلاسهایتان را به اتمام برسانید. اگر برای اتمام آنها نیز برنامه‌ریزی کنید، باز هم این شانس که قابلیتهای اضافه شده درست نباشند، وجود دارد.

پیاده‌سازی همه قابلیتها مشکل مشترک همه برنامه‌نویسانی است که تازه به دنیای شیءگرایی قدم گذاشته‌اند. تنها سعی کنید قابلیتهایی را که به آنها نیاز دارید، پیاده‌سازی نمایید.

Player و HumanPlayer

کلاس Playerاز کلاس Hand (یک دسته کارت مربوط به یک بازیکن) استفاده می‌نماید. لیست ۱۵ـ۴ کدهای کلاس Hand را نشان می‌دهد.

لیست ۱۵-٤ Hand.java

```
import java.util.ArrayList;
import java.util.Iterator;

public class Hand {

    private ArrayList cards = new ArrayList();
    private static final int BLACKJACK = 21;

    public void addCard( Card card ) {
        cards.add( card );
    }
```

لیست ۱۵-٤ (ادامه)

```
    public boolean bust() {
        if( total() > BLACKJACK ) {
            return true;
        }
        return false;
    }

    public void reset() {
        cards.clear();
    }

    public void turnOver() {
        Iterator i = cards.iterator();
        while( i.hasNext() ) {
            Card card = (Card)i.next();
            card.setFaceUp( true );
        }
    }

    public String toString() {
        Iterator i = cards.iterator();
        String string = "";
        while( i.hasNext() ) {
            Card card = (Card)i.next();
            string = string + " " + card.toString();
        }
        return string;
    }

    public int total() {
        int total = 0;
        Iterator i = cards.iterator();
        while( i.hasNext() ) {
            Card card = (Card) i.next();
            total += card.getRank().getRank();
        }
        return total;
    }

}
```

کلاس Hand نحوه اضافه کردن کارتها به خود را می‌داند و اینکه جمع ارزش کارتهای موجود چقدر است و چگونه خود را با استفاده از یک رشته (String) نمایش دهد. توجه داشته باشید که کلاس Hand تنها آس را با ارزش ۱۱ می‌شمارد. در تکرار بعدی آس با ارزشهای ۱ یا ۱۱ شمارش می‌شود.

لیست‌های ۵ـ۱۵ و ۱۶ـ۱۵، به ترتیب کلاسهای Player و HumanPlayer را نشان می‌دهند.

لیست ۱۵-۵ Player.java

```
import java.util.ArrayList;
import java.util.Iterator;

public abstract class Player {

    private Hand hand;
    private String name;
    private ArrayList listeners = new ArrayList();

    public Player( String name, Hand hand ) {
        this.name = name;
        this.hand = hand;
    }

    public void addCard( Card card ) {
        hand.addCard( card );
        notifyListeners();
    }

    public void play( Dealer dealer ) {
        // as before, play until the player either busts or stays
        while( !isBusted() && hit() ) {
            dealer.hit( this );
        }
        // but now, tell the dealer that the player is done, otherwise nothing
        // will happen when the player returns
        stopPlay( dealer );
    }

    public void reset() {
        hand.reset();
    }

    public boolean isBusted() {
        return hand.bust();
    }

    public void addListener( PlayerListener l ) {
        listeners.add( l );
    }

    public String toString() {
```

لیست ۱۵-۵ (ادامه)

```
        return ( name + ": " + hand.toString() );
    }

    protected Hand getHand() {
        return hand;
    }

    protected void notifyListeners() {
        Iterator i = listeners.iterator();
        while( i.hasNext() ) {
            PlayerListener pl = (PlayerListener) i.next();
            pl.handChanged( this );
        }
    }

    /**
     * The call to passTurn MUST be inside of a protected method. The Dealer
     * needs to override this behavior! Otherwise it will loop forever.
     */
    protected void stopPlay( Dealer dealer ) {
        dealer.passTurn();
    }

    protected abstract boolean hit();

}
```

لیست ۱۵-۶ HumanPlayer.java

```
public class HumanPlayer extends Player {

    private final static String HIT   = "H";
    private final static String STAND = "S";
    private final static String MSG   = "[H]it or [S]tay";
    private final static String DEFAULT = "invalid";

    public HumanPlayer( String name, Hand hand ) {
        super( name, hand );
    }

    protected boolean hit() {
        while( true ) {
            Console.INSTANCE.printMessage( MSG );
            String response = Console.INSTANCE.readInput( DEFAULT );
```

لیست ۱۵-۶ (ادامه)

```
            if( response.equalsIgnoreCase( HIT ) ) {
                return true;
            } else if( response.equalsIgnoreCase( STAND ) ) {
                return false;
            }
            // if we get here loop until we get meaningful input
        }
    }

}
```

کلاس مجرد Player تمام رفتارها و خصوصیات مشترک بازیکنان و واسط را در بر می‌گیرد. در این کلاس یک متد مجرد تعریف شده است: public boolean hit(). در هنگام بازی، کلاس پایه Player برای تشخیص اینکه باید کشیدن کارتها را ادامه داد و یا متوقف شد، این متد را فراخوانی می‌کند. زیرکلاسها می‌توانند متد فوق را بسته به رفتارها و واکنشهای خود، پیاده‌سازی نمایند. برای مثال کلاس HumanPlayer از کاربر می‌پرسد که کشیدن کارتها را ادامه دهد و یا آنکه متوقف شود. زمانی که Player بازی خود را انجام داد، کلاس Dealer را با فراخوانی متد passTurn() مطلع می‌سازد. با فراخوانی این متد، کلاس Dealer، از بازیکن بعدی می‌خواهد که بازی خود را شروع کند.

Dealer

Dealer رابطی است که متدهای بیشتری را نسبت به آنچه واسط می‌تواند انجام دهد، مشخص می‌کند. لیست ۱۵ ـ ۷ رابط Dealer را نشان می‌دهد.

لیست ۱۵-۷ Dealer.java

```
public interface Dealer {
    public void hit( Player player );

    public void passTurn();
}
```

لیست ۱۵ ـ ۸ کلاس BlackjackDealer را نشان می‌دهد.

لیست ۱۵-۸ BlackjackDealer.java

```
import java.util.ArrayList;
import java.util.Iterator;

public class BlackjackDealer extends Player implements Dealer {

    private Deckpile cards;
    private ArrayList players = new ArrayList();
    private int player_index;
```

لیست ۱۵-۸ (ادامه)

```
public BlackjackDealer( String name, Hand hand, Deckpile cards ) {
    super( name, hand );
    this.cards = cards;
}

public void passTurn() {
    if( player_index != players.size() ) {
        Player player = (Player) players.get( player_index );
        player_index++;
        player.play( this );
    } else {
        this.play( this );
    }
}

public void addPlayer( Player player ) {
    players.add( player );
}

public void hit( Player player ) {
    player.addCard( cards.dealUp() );
}

// override so that the dealer shows his cards before he starts play
public void play( Dealer dealer ) {
    exposeCards();
    super.play( dealer );
}

public void newGame() {
    // deal the cards and tell the first player to go
    deal();
    passTurn();
}

public void deal() {

    cards.shuffle();

    // reset each player and deal 1 card up to each and self
    Player [] player = new Player[players.size()];
    players.toArray( player );
    for( int i = 0; i < player.length; i ++ ) {
        player[i].reset();
```

لیست ۱۵-۸ (ادامه)

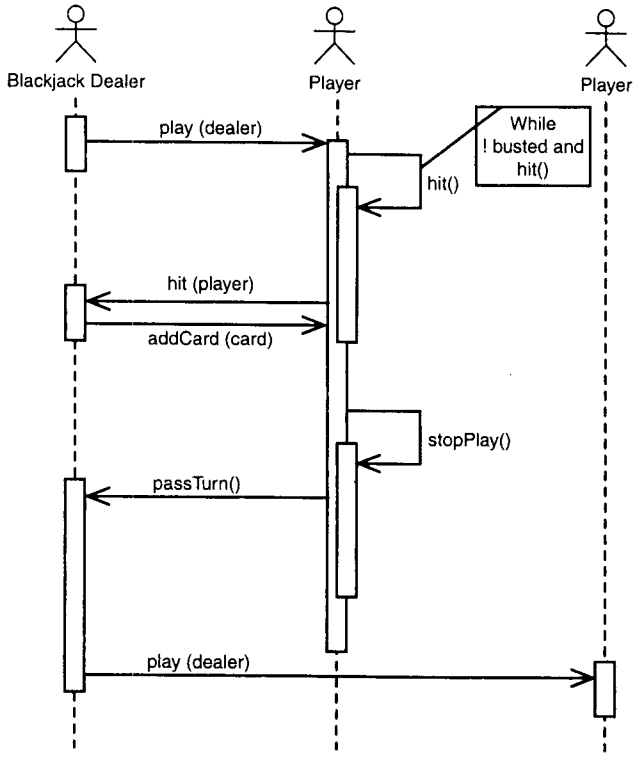
```
            player[i].addCard( cards.dealUp() );
        }
        this.addCard( cards.dealUp() );

        // deal 1 more up card to each player and one down to self
        for( int i = 0; i < player.length; i ++ ) {
            player[i].addCard( cards.dealUp() );
        }
        this.addCard( cards.dealDown() );

    }

    protected void stopPlay( Dealer dealer ) {
        // do nothing here in the dealer, simply let the game stop
        // if this were not overridden it would call passTurn() and
        // loop forever
    }

    protected boolean hit() {
        if( getHand().total() <= 16 ) {
            return true;
        }
        return false;
    }

    private void exposeCards() {

        getHand().turnOver();
        notifyListeners();
    }

}
```

کلاس BlackjackDealer از کلاس Player مشتق شده است چرا که واسط خود یک بازیکن است. به اضافه آنکه متدهای ارایه شده برای کلاس Player در مورد واسط (Dealer) نیز صدق می‌کند. اگر بازیکنی متد passTurn() از کلاس Dealer را فراخوانی کند، کلاس Dealer به سراغ بازیکن بعدی رفته و از وی می‌خواهد که بازی خود را شروع کند.

شکل ۱۵ ـ ۱۳ تعامل بین واسط و بازیکنان را نشان می‌دهد.

کلاس BlackjackDealer متد stopPlay() را جایگزین کرده است تا بتواند بازی را متوقف کند.

همچنین این کلاس متد hit() را پیاده‌سازی کرده تا در صورتی که دسته کارتها ارزشی کمتر از ۱۷ دارند، مقدار true را برگرداند یا در صورتی که ارزش کارتها بیشتر از ۱۷ است مقدار false را برگرداند.



شکل ۱۵ ـ ۱۳
تعامل بین بازیکنان و واسط

BlackjackGame

لیستهای ۱۵ ـ ۹ و ۱۵ ـ ۱۰ کلاسهای Blackjack و همچنین Console را نمایش می‌دهند.

لیست ۱۵-۹ Blackjack.java

```
public class Blackjack {

    public static void main( String [] args ) {

        Deckpile cards = new Deckpile();
        for( int i = 0; i < 4; i ++ ) {
            cards.shuffle();
            Deck deck = new Deck();
            deck.addToStack( cards );
            cards.shuffle();
        }

        Hand dealer_hand = new Hand();
        BlackjackDealer dealer = new BlackjackDealer( "Dealer", dealer_hand, cards );
        Hand human_hand = new Hand();
        Player player = new HumanPlayer( "Human", human_hand );
        dealer.addListener( Console.INSTANCE );
        player.addListener( Console.INSTANCE );
```

لیست ۱۵-۹ (ادامه)

```
        dealer.addPlayer( player );

        dealer.newGame();
    }

}
```

لیست ۱۵-۱۰ Console.java

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class Console implements PlayerListener {

    // console singleton
    public final static Console INSTANCE = new Console();

    private BufferedReader in =
        new BufferedReader( new InputStreamReader( System.in ) );

    public void printMessage( String message ) {
        System.out.println( message );
    }

    public String readInput( String default_input ) {
        String response;
        try {
            return in.readLine();
        } catch (IOException ioe) {
            return default_input;
        }
    }

    public void handChanged( Player player ) {
        printMessage( player.toString() );
    }

    // private to prevent instantiation
    private Console() {}

}
```

بازی Blackjack هریک از کلاسهای Dealer، Hand، Deckpile و Deck را ساخته و همه آنها را به هم متصل می‌کند. پس از اتصال آنها به هم با فراخوانی Dealer برای شروع یک بازی جدید، برنامه آغاز می‌شود. از طریق Console می‌توان به خط فرمان دسترسی داشت. همچنین کلاس فوق به تغییرات بازیکنان واکنش نشان داده و هر زمان که به‌روز شوند، آنها را بر روی صفحه نمایش، نشان می‌دهد.

اشتباهات در برنامه‌نویسی رویه‌ای

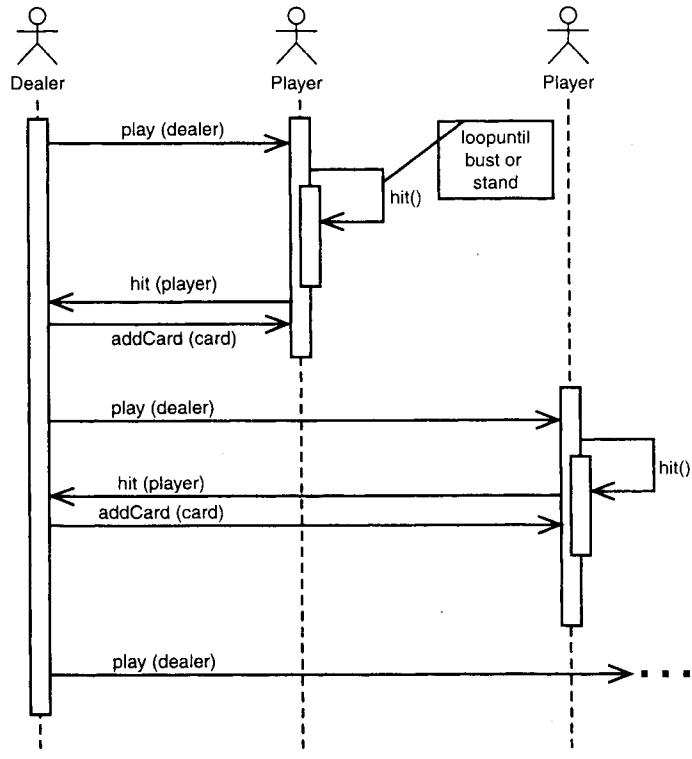
اگر با پیش‌زمینه‌ای در برنامه‌نویسی رویه‌ای نوشتن بازی فوق را آغاز کنید، ممکن است دچار اشتباهی بزرگ در این زمینه شوید. درواقع متد newGame() از BlackjackDealer را ممکن است به صورت زیر پیاده‌سازی کنید.

لیست ۱۵-۱۱ یک پیاده‌سازی رویه‌ای از BlackjackDealer

```
public void newGame() {
    cards.shuffle();

    // reset each player and deal 1 card up to each and self
    Player [] player = new Player[players.size()];
    players.toArray( player );
    for ( int i = 0; i < player.length; i ++ ) {
        player[i].reset();
        player[i].addCard( cards.dealUp() );
    }
    this.addCard( cards.dealUP() );

    // deal 1 more up card to each player and one down to self
    for( int i = 0; i < player.length; i ++ ) {
        player[i].addCard( cards.dealUP() );
    }
    this.addCard( cards.dealDown() );

    // have each player play and then dealer
    for( int i = 0; i < player.length; i++ ) {
        player[i].play(this);
    }
    exposeCards();
    this.Play( this );

}
```

پیاده‌سازی فوق، متد passTurn() را فراخوانی نمی‌کند. چرا که برای ایجاد حلقه بازی از روش رویه‌ای استفاده کرده است. به جای آنکه کلاسهای player با Dealer در انتهای بازی در ارتباط باشند، Dealer به طور متوالی حلقه‌ای را اجرا کرده و هریک از بازیکنان را پی‌درپی فراخوانی می‌کند.

شکل ۱۵ ـ ۱۴ تعامل بین بازیکنان و واسط را نشان می‌دهد.



شکل ۱۵ ـ ۱۴
تعامل رویه‌ای بین بازیکنان و واسط

توجه داشته باشید تعاملات نشان داده شده در شکل ۱۵ ـ ۱۳ به مراتب پویاتر از تعاملاتی است که در شکل ۱۵ ـ ۱۴ نشان داده شده است. شکل ۱۵ ـ ۱۳ سیستم واقعی از تعاملات بین اشیاء است. در شکل ۱۵ ـ ۱۴، Dealer منتظر می‌ماند تا کنترل برنامه از Player بازگردد. به بازیکن بعدی انتقال یابد. هیچ تعاملی بین واسط و دیگر بازیکنان وجود ندارد تا به بازیکن بعدی اعلام شود تا بازی را شروع کند. اگرچه این روش کار می‌کند، به هیچ وجه پایدار نیست و روش شیءگرایی که در ۱۵ ـ ۱۳ نشان داده شده است را در بر ندارد.

آزمایش سیستم

آنچنانکه در فصل چهاردهم، نشان داده شده، تست باید فرایندی مداوم باشد. مجموعه کاملی از تستها به همراه سرس کدهای آنها برای انتقال از اینترنت آماده است. مطالعه بر روی این مورد به عنوان تمرین به خواننده واگذار می‌گردد.

خلاصه

امروز یک بازی ابتدایی را تحلیل، طراحی و سپس پیاده‌سازی کردید. با استفاده از فرایند تکراری توانستید نتایج کار خود را سریع ببینید. با دروس ارایه شده در روزهای در پیش رو، قابلیتهای جدیدی را به بازی Blackjack اضافه خواهید کرد. در خلال درس امروز، مطالب دیگری را هم فرا گرفتید. مثلاً آنکه مراقب اشتباهاتی که برنامه‌نویس ممکن است با آنها درگیر شود، نظیر نوشتن کد به شیوه رویه‌ای باشید. همچنین باید از اضافه کردن جزییات زیاد و غیرضروری در برنامه چشم‌پوشی کرد.

پرسشها و پاسخها

اگر تست و آزمایش نرم‌افزار نوشته شده، بسیار مهم است چرا از آن گذشتید!
من قسمت تست را به کلی کنار نگذاشتم. تمام کدهایی که می‌توانید آن را از اینترنت بیاورید، شامل تمام موارد تست است. متن فوق تنها به خاطر صرفه‌جویی، مطالب مربوط به تست را نیاورده است. بنابراین مطالعه و فهم کدهای تست به عنوان تمرینی به خواننده واگذار می‌گردد.

به نظر می‌رسد کدهای مربوط به بازی بیش از این چیزی است که در این فصل چاپ شده است باقی کدها کجاست؟
بله، کدهای زیادی برای این برنامه وجود دارد ولی نمی‌توان تمام کدها را داخل متن آورد. فرض کنید کدهای آورده تکه فیلمهای انتخابی از کدها است! باید وقت قابل ملاحظه‌ای را بگذارید تا از تمام کدها سر در بیاورید. هدف از ارایه این پروژه در روزهای آخر این کتاب، ارایه یک پروژه کلی است که کدهای برنامه قسمتی از آن است. تحلیل و طراحی به اندازه هم مهم هستند. برای فهم بیشتر باید وقت بیشتری بر روی کدهای ارایه شده بگذارید.

کارگاه

پرسشها و پاسخهای ارایه شده تنها برای فهم بیشتر از مطالب آورده شده‌اند.

پرسشها

۱. **دو الگوی طراحی را که در امروز دیدید، نام ببرید. از این الگوها در کجا استفاده شد؟**
۲. **یک مثال از چند شکلی را که در کدها آورده شده است، نشان دهید.**
۳. **یک مثال از وراثت را نشان دهید.**
٤. **کلاس Deck چگونه کارتهای موجود را در خود کپسوله می‌کند؟**
۵. **چگونه کلاسهای BlackjackDealer و HumanPlayer به صورت چندشکلی عمل می‌کنند؟**

تمرین‌ها

۱. سرس کدهای مربوط به تکرار امروز را از اینترنت بیاورید. پس از دریافت کدها، آن را کامپایل کرده و سپس اجرا کنید. سپس سعی کنید مطالب ارایه شده را خوب بفهمید. برای این کار وقت صرف کرده و حوصله به خرج دهید.

۲. درس امروز طولانی و خسته کننده بوده، تمرین دیگری برای امروز نیاز نیست. حتماً سرس کدها و مطالب را مرور کنید.

روز ۱۶

Object Oriented Programming

تکرار دوم Blackjack: افزودن قوانین

در درس دیروز به تحلیل و طراحی اولیه بازی Blackjack پرداختیم. امروز هم در ادامه، قوانین جدیدی به بازی خواهیم افزود.

امروز خواهیم آموخت که چگونه

- حالتهای بازی را مدل کنیم
- از حالتها برای حذف منطق شرطی استفاده کنیم.

قوانین Blackjack

در درس دیروز یک بازی Blackjack ساده ایجاد کردیم. در آن بازی می‌شد کارت پخش کرد و تا حدی به بازی پرداخت. اما بازی واقعی کمی پیچیده‌تر و فراتر از این بازی ساده است. در بازی واقعی آس‌ها دارای ۱ یا ۱۱ امتیاز هستند. بازیکنان می‌توانند ببرند، ببازند، Blackjack شوند یا مساوی کنند. پخش کننده در صورتی که بقیه ۲۱ شوند (bust) یا خودش یک دست Blackjack داشته باشد، اصلاً نمی‌تواند بازی کند.

امروز این قوانین و برخی قوانین دیگر را به بازی می‌افزاییم. مانند همیشه کار را با بررسی و تحلیل حالتهای مختلف آغاز خواهیم کرد.

تحلیل قوانین

برای درک کامل تمام قوانین Blackjack باید تمام موارد مطرح شده در

درس دیروز و نیز حالتهای جدید را مورد بررسی قرار دهیم. وقتی که حالتهای استفاده و موارد کاملاً واضح شدند، باید مدل دامنه را به روز کنیم.

تحلیل حالتهای قوانین Blackjack

افزودن قوانین، برخی حالتهای بررسی شده دیروز را تغییر خواهد داد. حالت جدیدی هم وجود دارد: پخش کننده بازی کند. بگذارید با مورد پخش ورق بررسی شده در درس دیروز آغاز کنیم:

با شروع از بازیکن اول، واسط به هر نفر یک کارت می‌دهد طوری که کارت رو به بالا باشد و آخرین کارت را به خودش می‌دهد. پخش کننده همین عمل را تکرار می‌کند، اما ورق خود را رو به پایین می‌گذارد. پخش کردن به پایان می‌رسد و بازی آغاز می‌شود.

- پخش ورق
  1. واسط به هر بازیکنی از جمله خودش، یک کارت رو به بالا می‌دهد.
  2. واسط به همه غیر از خودش، یک کارت رو به بالا می‌دهد.
  3. واسط برای خودش یک کارت رو به پایین می‌گذارد.
- شرایط قبل
  - بازی جدید
- شرایط بعد
  - همه بازیکنان از جمله پخش‌کننده دستی با دو ورق داشته باشند.
  - نوبت بازیکنی است که دستش Blackjack (جمع خالهای دست ۲۱ شود) نیست.
  - بازی برای هر کس که دستش ۲۱ نیست ادامه می‌یابد.
- حالت‌های دیگر: پخش کننده Blackjack شود.

  اگر دست پخش کننده ۲۱ یا Blackjack شود، بازی تمام می‌شود.

به این مورد چند حالت شرایط بعدی و یک حالت جانشین افزوده شده است. توجه کنید که اگر دست پخش کننده ۲۱ یا Blackjack شود، بازی همانجا تمام می‌شود. به همان صورت هر بازیکنی که Blackjack دارد، نمی‌تواند بازی کند.

حال بپردازیم به ورق خواستن بازیکنان اگر بازیکنی از دستش راضی نباشد، می‌تواند تقاضای یک ورق دیگر بکند، یا پاس کند (کاری انجام ندهد تا نوبتش رد شود) اگر دست بازیکن ۲۱ نشود نوبت به بازیکن بعدی می‌رسد.

- تقاضای ورق (کارت کشیدن)
  1. بازیکن از دست خودش راضی نیست.
  2. بازیکن یک ورق دیگر تقاضا می‌کند.
  3. اگر دستش ۲۱ یا کمتر شد، می‌تواند دوباره درخواست ورق کند یا نوبتش را به بازیکن بعدی بدهد.
- شرایط قبل

- دست بازیکن ۲۱ یا کمتر است.
- بازیکن Blackjack نشده.
- واسط Blackjack نیست.
- شرایط بعد
- یک ورق جدید به دست بازیکن اضافه می‌شود.
- حالت دیگر: بازیکن می‌بازد

ورق جدید مجموع خالهای دست بازیکن را از ۲۱ بیشتر می‌کند. بازیکن یک آس دارد. مقدار آس می‌تواند ۱ یا ۱۱ شمرده شود. لذا دست می‌تواند ۲۱ یا کمتر شود. بازیکن تصمیم می‌گیرد که درخواست ورق کند یا پاس کند.

لازم به ذکر است بازیکن در صورتی می‌تواند بازی کند که نه خودش و نه پخش کننده هیچ کدام Blackjack نشده باشند.

صبر کردن یا پاس کردن هم کمی تغییر می‌کند. در این حالت بازیکن از دست خودش راضی است، لذا آن را همانطور نگاه می‌دارد.

- بازیکن توقف می‌کند
۱. بازیکن از دستش راضی است.
- شرایط قبل
- دست بازیکن، دارای مجموعی کمتر یا مساوی ۲۱ است.
- بازیکن Blackjack نیست.
- پخش کننده Blackjack نیست.
- شرایط بعد
- نوبت بازیکن تمام می‌شود

در صورتی که مجموع خالهای دست پخش کننده کمتر از ۱۷ باشد، او باید تقاضای ورق کند. اگر پخش کننده بعد از ورق جدید نبازد و هنوز مجموع دستش زیر ۱۷ باشد، باید دوباره ورق درخواست کند. در صورتی که مجموع دست پخش کننده ۱۷ یا بیشتر شد، باید صبر کند. در صورتی که پخش کننده توقف کند یا ببازد، بازی قطع می‌شود.

- درخواست کارت پخش کننده(واسط کارت می‌کشد)
۱. اگر مجموع دست پخش کننده کمتر از ۱۷ باشد، او درخواست ورق می‌کند.
۲. ورق جدید به دست او اضافه می‌شود.
۳. در صورتی که جمع خالهای دست کمتر از ۱۷ باشد، پخش کننده باید دوباره درخواست ورق کند.
- پیشفرض‌ها
- مجموع خالهای دست پخش کننده کمتر از ۱۷ است.
- باید بازیکنی در وضعیت توقف یا صبر باشد.

- شرایط بعد
  - ورق جدید در دست پخش کننده
  - پایان بازی
- حالت دیگر: پخش کننده می‌بازد

کارت جدید، مجموع دست او را از ۲۱ بیشتر می‌کند. پخش کننده می‌بازد.

- حالت دیگر: پخش کننده می‌ماند.

کارت جدید، باعث می‌شود مجموع خالهای دست پخش کننده بزرگتر یا مساوی ۱۷ شود، پخش کننده می‌ماند.

- حالت دیگر: پخش کننده آس دارد، پاس می‌کند.

ورق جدید مجموع خالهای دست را بزرگتر یا مساوی ۲۱ می‌کند، ولی یک آس در دست وجود دارد. در صورتی که تبدیل ارزش آس از ۱۱ به ۱ دست را به زیر ۱۷ بیاورد، پخش کننده درخواست ورق می‌کند. پیش فرض «بازیکنی در حالت توقف است.» یعنی اینکه حداقل یکی از بازیکنان نه باخته و نه Blackjack شده. به همین صورت اگر هیچ بازیکن دیگری در حالت توقف یا پاس نباشد، پخش کننده هم به صورت خودکار در وضع پاس یا انتظار قرار می‌گیرد.

- پخش کننده پاس می‌کند.
  - دست پخش کننده ۱۷ یا بالاتر است و او توقف می‌کند.
- شرایط قبل
  - دست پخش کننده بزرگتر یا مساوی ۱۷ است.
  - حداقل یک بازیکن باید در وضعیت پاس یا انتظار باشد.
- حالت دیگر: هیچ بازیکنی در حالت انتظار نیست.

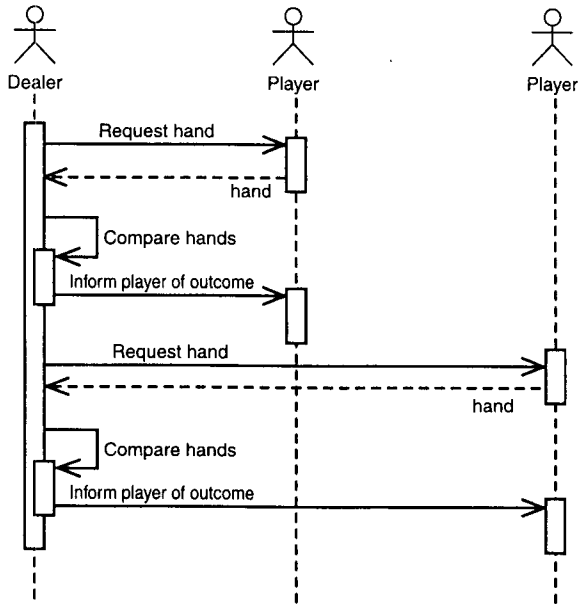
در این صورت پخش کننده به صورت خودکار در حالت پاس قرار می‌گیرد.

وقتی بازی انجام شد، پخش کننده باید مشخص کند چه کسی برده، باخته یا مساوی شده.

- تعیین نتیجه

۱. پخش کننده دست بازیکن اول را با دست خودش مقایسه می‌کند.
۲. اگر دست بازیکن بزرگتر از پخش کننده باشد ولی نباخته باشد، بازیکن می‌برد.
۳. روند فوق برای تمام بازیکنان تکرار می‌شود.

- شرایط قبل
  - تمام بازیکنان پاس کرده‌اند.
  - پخش کننده پاس کرده است.
- شرایط بعد
  - نتایج نهایی معلوم می‌شوند.
- حالت دیگر: باخت بازیکن



شکل ۱۶ ـ ۱
نمای رویدادهای تعیین نتیجه بازی

- دست بازیکن کمتر از دست پخش کننده است، لذا می‌بازد.
- حالت دیگر: تساوی
- دست بازیکن برابر دست پخش کننده است، بنابراین نتیجه مساوی است.
- حالت دیگر: پخش کننده می‌بازد.
- اگر پخش کننده ببازد، تمام بازیکنانی که پاس کرده‌اند یا Blackjack شده‌اند برنده هستند و بقیه بازنده.

مدلسازی حالتها

اکثر تغییرات لازم ساده هستند. شاید رسم نمای رویدادهای تعیین نتیجه بازی مفید باشد. شکل ۱۶ ـ ۱ نمای مزبور را نشان می‌دهد.

به‌روزآوری مدل دامنه

تغییرات و حالتهای جدید تغییری در دامنه ایجاد نمی‌کند.

طرح قوانین

در این نقطه ایدهٔ شروع پیاده‌سازی واقعاً وسوسه‌انگیز است. با دیدی سطحی به نظر می‌رسد که می‌توان از روی شرایط و فرض‌ها قوانین بازی را توسط عبارات شرطی پیاده سازی کرد. درواقع می‌توان این کار را کرد. در لیست ۱۶ ـ ۱ می‌توان مشاهده کرد که این پیاده‌سازی‌های شرطی به چه صورتی خواهند بود.

لیست ۱۶-۱ پیاده‌سازی شرطی

```
protected void stopPlay(Dealer dealer ) {
    // the game is over, pick the winners and point out the losers
```

لیست ۱۶-۱ (ادامه)

```
    if( isBusted() ) {
        Iterator i = players.iterator();
        while( i.hasNext() ) {
            Player player = (Player) i.next();
            if( !player.isBusted() ) {
                Console.INSTANCE.printMessage( player.toString() + " WINNER!" );
            }
        }
    } else {
        if( hasBlackjack() ) {
            Iterator i = players.iterator();
            while( i.hasNext() ) {
                Player player = (Player) i.next();
                if( player.hasBlackjack() ) {
                    Console.INSTANCE.printMessage( player.toString() + " STANDOFF!" );
                } else {
                    Console.INSTANCE.printMessage( player.toString() + "LOSER!" );
                }
            }
        } else { //deal is not busted and does not have blackjack
            Iterator i = players.iterator();
            while( i.hasNext() ) {
                Player player = (Player) i.next();
                if( player.hasBlackjack() ) {
                Console.INSTANCE.printMessage( player.toString() + "WINNER WITH BLACKJACK!" );
                } else if( player.isBusted() ) {
                    Console.INSTANCE.printMessage( player.toString() + " BUSTED!" );
                } else if( player.getHand().greatedThan( getHand() ) ) {
                    Console.INSTANCE.printMessage( player.toString() + " WINNER!" );
                } else if( player.getHand().equalTo( getHand() ) ) {
                    Console.INSTANCE.printMessage( player.toString() + " STANDOFF!" );
                } else {
                    Console.INSTANCE.printMessage( player.toString() + " LOSER!" );
                }
            }
        }
    }
}
```

البته، این پیاده سازی فقط به تعیین نتیجه مرتبط است. پخش کننده احتیاج به بررسی‌ها و شرایط بیشتری دارد، آن هم فقط برای اینکه بداند از پخش ورق، بازی را شروع کند یا خیر. مثلاً در صورتی که دست پخش کننده Blackjack شود، بازی باید به صورت خودکار تمام شود. برای انجام این حالت و بسیاری حالتهای دیگر، نیاز به تعداد زیادی عبارت شرطی داریم.

چنین راهبردی آسیب‌پذیر، مشکل، خطاساز و زشت است! وقتی با عبارات شرطی کار می‌کنیم، می‌بینیم که با افزودن هر شرط جدید، یک رفتار موجود تخریب می‌شود. فهم کدی که مملو از عبارات شرطی باشد هم مشکل است.

عبارات شرطی ذاتاً شیءگرا نیستند. استفاده نامناسب از عبارات شرطی محدودهٔ وظایف متناسب که در روش شیءگرا کلیدی است را بر هم می‌زند. عبارات شرطی مزایای خاص خود را دارند، اما هرگز نباید بگذارید در وظایف اشیاء اشکال ایجاد کنند یا تقسیم آنها را بر هم بزنند.

در عوض دانش اینکه بازیکنی باخته یا Blackjack شده باید در خود کلاس Player بماند. در این صورت وقتی رویدادی رخ می‌دهد، را مطلع کند. به Player می‌تواند عملیات لازم را بر عهده بگیرد و در صورت لزوم واقعی را بهتر مدل BlackjackDealer جای اینکه پخش کننده برای بازیکنان تصمیم بگیرد، بازکنان باید از حالتهای داخلی خود برای تصمیم‌گیری استفاده کنند. چنین راهبردی بازی می‌کند. Blackjack

اولین قدم برای رها شدن از عبارات شرطی فهم این مطلب است که بازی Blackjack توسط رویدادها و حالتها به پیش برده می‌شود. در طی بازی، بازیکنان مختلف شرایط و حالتهای مختلفی به خود می‌گیرند. گاهی پاس می‌کنند، گاهی ورق درخواست می‌کنند. بعد از بازی حالت بازیکن به صورت پاس یا به صورت باخته در می‌آید. به همین صورت پخش کننده از پخش کردن، به حالت منتظر نوبت شدن، یا بازی کردن و یا پاس کردن می‌رود.

حالتهای جانشینی هم وجود دارند. مثلاً بعد از پخش ورق‌ها، بازیکنی ممکن است به حالت Blackjack رانده شود. این در صورتی است که دستش Blackjack باشد.

برای درک کامل حالتهای گوناگون و رویدادهایی که این حالتها را باعث می‌شوند، مدل کردن حالتهای مختلف در دیاگرام حالت مفید خواهد بود.

دیاگرام حالت

UML مجموعه‌ای غنی از نمادها برای مدلسازی دیاگرام‌های حالت دارد. به جای سردرگم شدن در جزییات، فقط به جنبه‌هایی خواهیم پرداخت که برای مدلسازی بازی به آنها نیازمندیم.

در مدل Blackjack از حالت (State)، انتقال (Transition)، رویداد (Event)، فعالیت (Activity) و شرط (Condition) استفاده می‌کنیم.

در Blackjack حالت عبارت است از وضعیت جاری یک بازیکن. این حالتها عبارتند از انتظار، بازی کردن، باختن، پاس کردن و Backjack.

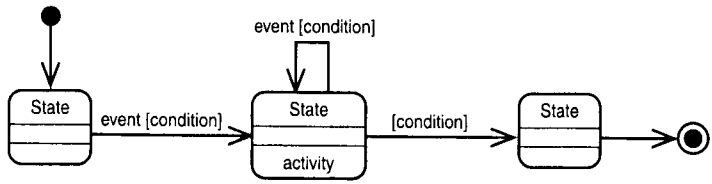
انتقال وقتی رخ می‌دهد که بازیکنی از حالتی به حالت دیگر منتقل شود. مثلاً بازیکن از حالت بازی کردن به حالت باخته می‌رود.

رویدادها محرکهایی هستند که باعث انتقال بازیکن از حالتی به حالت دیگر می‌شوند. برای مثال وقتی دست بازیکن از ۲۱ بیشتر می‌شود، بازیکن از حالت بازی کردن به حالت باخته منتقل می‌شود.

فعالیتها اعمالی هستند که در حالت خاصی صورت می‌پذیرند. مثلاً وقتی بازیکنی در حالت بازی است، تا وقتی ببازد یا راضی شود درخواست ورق می‌کند.

شرایط محافظ عباراتی منطقی هستند که انتقال‌ها را اجباری می‌کنند. برای مثال اگر بازیکن نخواهد درخواست ورق کند به حالت پاس رانده می‌شود.

شکل ۱۶ ـ ۲
نمادهای دیاگرام حالت

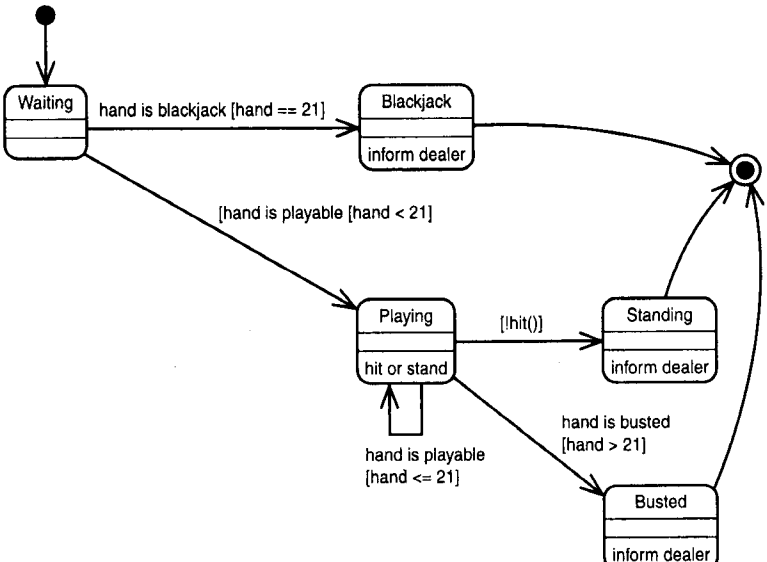


شکل ۱۶ ـ ۲ نمادهای مورد استفاده برای مدلسازی Blackjack را نشان می‌دهد.
در مدل فوق انتقال‌ها توسط پیکان مدل شده‌اند. از آنجایی که انتقال در اثر رویداد یا شرایط رخ می‌دهد، روی پیکان باید نام رویداد یا شرطی که باعث انتقال شده نوشته شود.
مشاهده می‌کنید که می‌توان از حالتی به همان حالت منتقل شد. چنین انتقالی را خودانتقالی (Self Transition) می‌نامند.
سرانجام، اگر در حالتی کار خاصی انجام شود، آن عمل تحت عنوان فعالیت درون سمبل حالت ثبت می‌شود. کاملاً امکان دارد که در حالتی هیچ فعالیتی انجام نشود.

مدل‌کردن حالتهای بازیکن

شکل ۱۶ ـ ۳ دیاگرام حالت بازیکن را نمایش می‌دهد.
بازیکن پنج حالت اصلی دارد: صبر کردن، Blackjack، بازی کردن، پاس کردن و باختن. بازیکن همواره در حالت انتظار شروع می‌کند. بعد از پخش ورق، بازیکن یا به حالت Blackjack می‌رود یا به حالت بازی کردن. وقتی نوبت بازی او رسید، بازی می‌کند (فعالیت حالت بازی کردن). در حین بازی کردن بازیکن می‌تواند

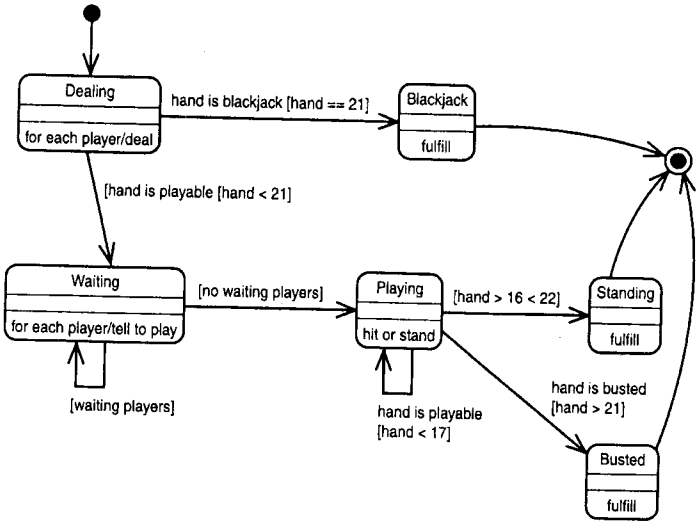
شکل ۱۶ ـ ۳
دیاگرام حالت بازیکن



مدلسازی حالتهای پخش‌کننده

شکل ۱۶ ـ ۴ دیاگرام حالت پخش کننده را نشان می‌دهد.
تصمیم بگیرد که پاس کند، در این صورت به حالت پاس منتقل می‌شود. اگر بازیکن ورق بخواهد، یا به حالت

شکل ۱۶ ـ ۴
دیاگرام حالت پخش‌کننده



باخته می‌رود یا دوباره به حالت بازی برمی‌گردد. این عمل تا وقتی ادامه می‌یابد که بازیکن ببازد یا پاس کند.

پخش کننده شش حالت اصلی دارد: پخش کردن و حالتهای بازیکن. پخش کننده در حالت پخش کردن شروع می‌کند و بین بازیکنان پخش می‌کند. او بعد از پخش، یا به حالت Blackjack می‌رود یا به حالت صبر کردن.

در حالت صبر کردن، پخش کننده منتظر تمام شدن نوبت بقیهٔ بازیکنان می‌شود. در این حالت وی به حالت بازی کردن منتقل می‌شود تا نوبتش را بازی کند. بازی پخش کننده مانند بازیکن است با این تفاوت که اگر دستش کمتر از ۱۷ باشد، باید حتماً ورق درخواست کند و اگر دستش برابر یا بزرگتر از ۱۷ باشد، باید حتماً پاس کند.

لازم به ذکر است که در هریک از حالتهای Blackjack و باخته یا پاس، فعالیت پخش کننده تعیین نتیجه بازی و به پایان بردن آن است.

مدل به روز شدهٔ Blackjack

وقتی حالتهای بازی مدل شدند، باید مورد چگونگی گنجاندن آنها در طرح تصمیم بگیرید. با تبدیل هریک از حالتها به کلاس مربوط به آن شروع کنید. سپس باید در مورد منبع رویدادها تصمیم‌گیری کنید.

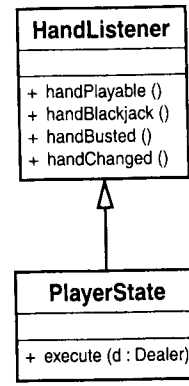
استفاده از عبارت حالت با تعریف اصلی که قبلاً مورد بررسی قرار گرفت مناسبت دارد. در اینجا باید تنها شیء برای هر حالت Playerایجاد کنید. این کار شما را از سروکله زدن با تعداد زیادی متغیر داخلی می‌رهاند. علاوه بر این شیء حالت، به خوبی تمام مقادیر درونی مختلف شیء هم حالت و هم رفتار دارد را کپسوله می‌کند.

به سرعت قابل درک است که تمام رویدادها حول و حوش حالت دست رخ می‌دهند. بنابراین احتمالاً باید ایجاد و ارسال رویدادهای مناسب برای وقتی که Cardها به Handاضافه می‌شوند به خود Hand سپرده شود. باید فکری هم برای مکانیزم دریافت رویدادها توسط حالتها کرد.

حالتها خود بسیار ساده‌اند و سه وظیفه اساسی دارند.

- انجام فعالیتها
- پاسخگویی به رویدادها
- بدانند در پاسخ به هر رویداد به چه حالتی منتقل شوند.

شکل ۱۶ـ۵
دیاگرام کلاس State



شکل ۱۶ـ۵ دیاگرام کلاس رابط State را نشان می‌دهد.

مشاهده می‌کنید که هر رویدادی دارای روال مرتبطی در حالت است. Hand باتوجه به اینکه می‌خواهد کدام حالت را گزارش کند، یکی از این روالها را فراخوانی می‌کند. علاوه بر این State یک روال execute() هم دارد. این روال وقتی فراخوانده می‌شود که باید فعالیتی انجام شود.

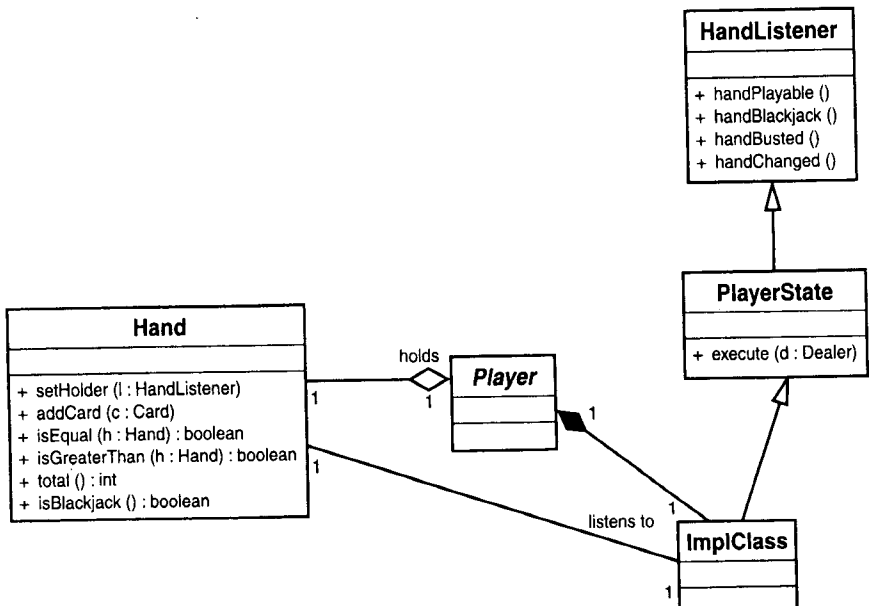
شکل ۱۶ـ۶ روابط بین Player، Hand و State را مدل می‌کند.

Player یک شیء State دارد. وقتی نوبت به Player می‌رسد، Player تنها روال execute() آن را اجرا می‌کند. در این صورت State هر فعالیتی که لازم باشد را انجام می‌دهد، به Hand گوش فرا می‌دهد و به State بعدی منتقل می‌شود. بعد از انتقال State بعدی عین همین اعمال را انجام می‌دهد. این فرایند تا پایان بازی ادامه خواهد داشت.

شکل ۱۶ـ۷ دیاگرام کامل کلاس بازی Blackjack را نشان می‌دهد.

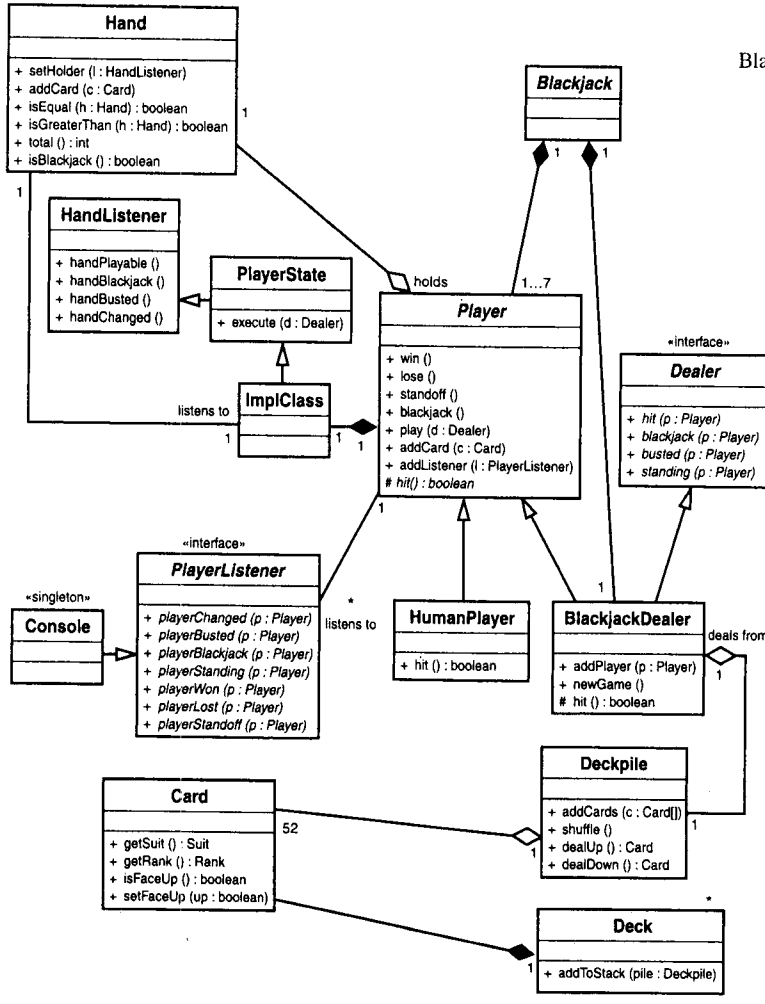
در این تکرار افزودن چهارچوب State مهمترین تغییر ایجاد شده است بقیهٔ کلاسها و رابطها هم برای استفاده از این شرایط جدید به‌روز شده‌اند.

شکل ۱۶ـ۶
دیاگرام کلاس
چهارچوب State



شکل ۱۶ ـ ۷

دیاگرام کامل کلاس Blackjack



پیاده‌سازی قوانین

برای پشتیبانی از ویژگیهای جدید بازی، تغییراتی در کلاسهای Player، Dealer، BlackjackDealer و Hand باید صورت گیرد.

همچنین کلاسها و رابطهای جدیدی باید افزوده شوند. بخش‌های آتی تغییرات عمده را بررسی خواهند کرد.

تغییرات در Hand

Hand باید حالت خود را گزارش کند. لیست ۱۶ ـ ۲ رابط جدید HandListener را نشان می‌دهد.

لیست ۱۶-۲ HandListener.java

```
public interface HandListener {

    public void handPlayable();
```

لیست ۱۶-۲ (ادامه)

```
    public void handBlackjack();

    public void handBusted();

    public void handChanged();

}
```

لیست ۱۶ ـ ۳ کلاس Hand به‌روز شده را نشان می‌دهد.

لیست ۱۶-۳ Hand.java

```
import java.util.ArrayList;
import java.util.Iterator;

public class Hand {

    private ArrayList cards = new ArrayList();
    private static final int BLACKJACK = 21;
    private HandListener holder;
    private int number_aces;

    public Hand() {
        // set the holder to a blank listener so it will not be null if not
        // externally set
        setHolder(
            new HandListener() {
                public void handPlayable() {}
                public void handBlackjack() {}
                public void handBusted() {}
                public void handChanged() {}
            }
        );
    }

    public void setHolder( HandListener holder ) {
        this.holder = holder;
    }

    public Iterator getCards() {
        return cards.iterator();
    }

    public void addCard( Card card ) {
        cards.add( card );

        holder.handChanged();

        if( card.getRank() == Rank.ACE ) {
```

لیست ۱۶-۳ (ادامه)

```
            number_aces++;
        }

        if( bust() ) {
            holder.handBusted();
            return;
        }
        if( blackjack() ) {
            holder.handBlackjack();
            return;
        }
        if ( cards.size() >= 2 ){
            holder.handPlayable();
            return;
        }
    }

    public boolean isEqual( Hand hand ) {
        if( hand.total() == this.total() ) {
            return true;
        }
        return false;
    }

    public boolean isGreaterThan( Hand hand ) {
        return this.total() > hand.total();
    }

    public boolean blackjack() {
        if( cards.size() == 2 && total() == BLACKJACK ) {
            return true;
        }
        return false;
    }

    public void reset() {
        cards.clear();
        number_aces = 0;
    }

    public void turnOver() {
        Iterator i = cards.iterator();
        while( i.hasNext() ) {
            Card card = (Card)i.next();
            card.setFaceUp( true );
        }
    }

    public String toString() {
        Iterator i = cards.iterator();
```

لیست ۱۶-۳ (ادامه)

```
        String string = "";
        while( i.hasNext() ) {
            Card card = (Card)i.next();
            string = string + " " + card.toString();
        }
        return string;
    }

    public int total() {
        int total = 0;
        Iterator i = cards.iterator();
        while( i.hasNext() ) {
            Card card = (Card) i.next();
            total += card.getRank().getRank();
        }
        int temp_aces = number_aces;
        while( total > BLACKJACK && temp_aces > 0 ) {
            total = total - 10;
            temp_aces--;
        }
        return total;
    }

    private boolean bust() {
        if( total() > BLACKJACK ) {
            return true;
        }
        return false;
    }

}
```

total()، اکنون این امکان را ایجاد کرده که آس بتواند هم مقدار ۱ و هم مقدار ۱۱ بگیرد. به همان صورت addCart() باعث شده که Hand بتواند تغییرات خود را به HandListener گزارش تغییرات کند. سرانجام تغییر isEqual() و isGreaterthan() تغییرات مقایسهٔ دستها را آسانتر کرده.

تغییرات Player

بزرگترین تغییر سلسله مراتب Player، افزوده شدن حالت‌ها است. لیست ۱۶ ـ ۴ رابط PlayerState را نمایش می‌دهد.

لیست ۱۶-٤ PlayerState.java

```
public interface PlayerState extends HandListener {
    public void execute( Dealer dealer );
}
```

پیاده‌سازی PlayerState به رویدادهای HandListener پاسخ می‌دهد و انجام فعالیتهای آن را توسط execute() تضمین می‌کند.

Player حالت فعلی را توسط متغیر current_state نگهداری می‌کند. روال Play() برای استفاده از این متغیر، تغییر داده شده:

```
public void play (Dealer dealer){
    current_state.execute(dealer);
}
```

به جای تعریف رفتار در Play()، Player به سادگی رفتار را به حالت واگذار می‌کند. به این صورت می‌توان رفتارهای جدید را تنها با تغییر اشیاء State، فراهم کرد. این کار بسیار بهتر از استفاده از عبارات شرطی است.

لیست‌های ۱۶ ـ ۵ تا ۱۶ ـ ۹ پیاده‌سازی PlayerState متعلق به Play را نشان می‌دهند. این حالت‌ها دقیقاً مدل حالت را پیاده‌سازی می‌کنند.

لیست ۱۶-۵ حالت انتظار پیش‌فرض

```
private class Waiting implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        setCurrentState( getPlayingState() );
        //transition
    }
    public void handBlackjack() {
        setCurrentState( getBlackjackState() );
        notifyBlackjack();
        //transition
    }
    public void handBusted() {
        //not possible in waiting state
    }
    public void execute( Dealer dealer ) {
        // do nothing while waiting
    }
}
```

لیست ۱۶-۶ حالت باخت پیش‌فرض

```
private class Busted implements PlayerState {
    public void handChanged() {
        // not possible in busted state
    }
    public void handPlayable() {
        // not possible in busted state
    }
```

لیست ۱۶-۶ (ادامه)

```
    public void handBlackjack() {
        // not possible in busted state
    }
    public void handBusted() {
        // not possible in busted state
    }
    public void execute( Dealer dealer ) {
        dealer.busted( Player.this );
        // terminate
    }
}
```

لیست ۱۶-۷ حالت Blackjack پیش‌فرض

```
private class Blackjack implements PlayerState {
    public void handChanged() {
        // not possible in blackjack state
    }
    public void handPlayable() {
        // not possible in blackjack state
    }
    public void handBlackjack() {
        // not possible in blackjack state
    }
    public void handBusted() {
        // not possible in blackjack state
    }
    public void execute( Dealer dealer ) {
        dealer.blackjack( Player.this );
        //terminate
    }
}
```

لیست ۱۶-۸ حالت توقف پیش‌فرض

```
private class Standing implements PlayerState {
    public void handChanged() {
        // not possible in standing state
    }
    public void handPlayable() {
        // not possible in standing state
    }
    public void handBlackjack() {
        // not possible in standing state
```

لیست ۱۶-۸ (ادامه)

```
    }
    public void handBusted() {
        // not possible in standing state
    }
    public void execute( Dealer dealer ) {
        dealer.standing( Player.this );
        // terminate
    }
}
```

لیست ۱۶-۹ حالت ادامه بازی پیش‌فرض

```
private class Playing implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        // can ignore in playing state
    }
    public void handBlackjack() {
        // not possible in playing state
    }
    public void handBusted() {
        setCurrentState( getBustedState() );
    }
    public void execute( Dealer dealer ) {
        if( hit() ) {
            dealer.hit( Player.this );
        } else {
            setCurrentState( getStandingState() );
            notifyStanding();
        }
        current_state.execute( dealer );
        // transition
    }
}
```

تمام این حالتها به عنوان کلاسهای درونی Player پیاده‌سازی شدند، چون آنها درواقع بسط کلاس Player هستند. به عنوان کلاسهای درونی، این حالت‌ها به متغیرها و روالهای درونی کلاس Player دسترسی کامل دارند. این کلاسهای درونی کپسوله‌سازی منطق حالت درون کلاس مربوطه آن را بدون تخریب کپسوله‌سازی خود کلاس ممکن کرده‌اند.

زیر کلاسهای Player می‌توانند از پیاده‌سازی حالت خود با جایگزینی روالهای زیر در Player استفاده کنند:

```
protected PlayerState getBustedState() {
    return new Busted();
}
protected PlayerState getStandingState() {
    return new Standing();
}
protected PlayerState getPlayingState() {
    return new Playing();
}
protected PlayerState getWaitingState() {
    return new Waiting();
}
protected PlayerState getBlackjackState() {
    return new Blackjack();
}
protected PlayerState getInitialState() {
    return new WaitingState();
}
```

تا وقتی که حالتها برای بازیابی حالتهای دیگر از این روالها استفاده کنند، زیر کلاسها می‌توانند حالتهای خاص خودشان را معرفی کنند. getInitialState() توسط کلاس پایهٔ Player برای تنظیم مقدار اولیه Player مورد استفاده قرار گرفته است. اگر زیرکلاسی بخواهد با حالت دیگری شروع کند، باید این روال را جایگزین کند.

و سرانجام تعدادی روال آگهی به کلاس Player افزوده شده‌اند. حالتها از این روالها برای خبر کردن Listener از ایجاد تغییر استفاده می‌کنند. لیست ۱۶ ـ ۱۰ رابط به روز شدهٔ PlayerListener را نشان می‌دهد.

لیست ۱۶-۱۰ PlayerListener.java

```
public interface PlayerListener {

    public void playerChanged( Player player );

    public void playerBusted( Player player );

    public void playerBlackjack( Player player );

    public void playerStanding( Player player );

    public void playerWon( Player player );
    public void playerLost( Player player );

    public void playerStandoff( Player player );

}
```

از آنجایی که Console یک PlayerListener است، روالهای زیر به آن افزوده شده‌اند.

```
public void playerChanged( Player player ) {
    printMessage( player.toString() );
}

public void playerBusted( Player player ) {
    printMessage( player.toString() + " BUSTED!" );
}

public void playerBlackjack( Player player ) {
    printMessage( player.toString() + " BLACKJACK!" );
}

public void playerStanding( Player player ) {
    printMessage( player.toString() + " STANDING!" );
}

public void playerWon( Player player ) {
    printMessage( player.toString() + " WINNER!" );
}

public void playerLost( Player player ) {
    printMessage( player.toString() + " LOSER!" );
}

public void playerStandoff( Player player ) {
    printMessage( player.toString() + " STANDOFF!" );
}
```

این تغییرات Console را قادر می‌سازند، که اتفاقات و رویدادهای اصلی بازی را نشان دهد. روالهای جدیدی هم به Player اضافه شده‌اند:

```
public void win() {
    notifyWin();
}

public void lose() {
    notifyLose();
}

public void standoff() {
    notifyStandoff();
}

public void blackjack() {
    notifyBlackjack();
}
```

این روالها Dealer را قادر می‌سازند که برد و باخت را به اطلاع Player برساند.

تغییرات BlackjackDealer, Dealer

لیست ۱۶ ـ ۱۱ رابط به روز شده Dealer را نشان می‌دهد.

لیست ۱۶-۱۱ Dealer.java

```
public interface Dealer {
    // used by the player to interact with the dealer
    public void hit( Player player );

    // used by the player to communicate state to dealer
    public void blackjack( Player player );
    public void busted( Player player );
    public void standing( Player player );
}
```

Player از این روالهای جدید برای گزارش حالت به Dealer استفاده می‌کند. این روالها فقط کمی جزیی‌تر از passTurn() قبلی عمل می‌کنند.

Dealer با فراخوانی این روالها Playerها را دسته‌بندی می‌کند. این کار اعلام نتیجه بازی را ساده‌تر می‌کند. مثلاً به پیاده‌سازی busted() از BlackjackDealer توجه کنید:

```
public void busted ( Player player ) {
   busted_players.add( player );
   play(this);
}
```

بقیه روالها هم به همین صورت عمل می‌کنند: BlackjackDealer حالت DealerDealing را اضافه می‌کند. این کلاس همچنین بسیاری از حالتهای Player را برای خود سفارشی می‌کند. لیست‌های ۱۶ ـ ۱۲ تا ۱۶ ـ ۱۶ این حالتها را نشان می‌دهند.

لیست ۱۶-۱۲ حالت باخت سفارشی شده واسط

```
private class DealerBusted implements PlayerState {
    public void handChanged() {
        // not pOssible in busted state
    }
    public void handPlayable() {
        // not pOssible in busted state
    }
    public void handBlackjack() {
        // not pOssible in busted state
    }
    public void handBusted() {
        // not pOssible in busted state
    }
```

لیست ۱۶-۱۲ (ادامه)

```
    public void execute( Dealer dealer ) {
        Iterator i = standing_player.iterator();
        while( i.hasNext() ) {
            Player player = (Player) i.next();
            player.win();
        }
        i = blackjack_players.iterator();
        while( i.hasNext() ) {
            Player player = (Player) i.next();
            player.win();
        }
        i = busted_players.iterator();
        while( i.hasNext() ) {
            Player player = (Player) i.next();
            player.lose();
        }
    }
}
```

لیست ۱۶-۱۳ حالت Blackjack سفارشی شده واسط

```
private class DealerBlackjack implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        // not possible in blackjack state
    }
    public void handBlackjack() {
        // not possible in blackjack state
    }
    public void handBusted() {
        // not possible in blackjack state
    }
    public void execute( Dealer dealer ) {
        exposeHand();
        Iterator i = players.iterator();
        while( i.hasNext() ) {
            Player player = (Player) i.next();
            if( player.getHand().blackjack() ) {
                player.standoff();
            } else {
                player.lose();
            }
        }
    }
}
```

لیست ۱۶-۱۴ حالت توقف سفارشی شده واسط

```
private class DealerStanding implements PlayerState {
    public void handChanged() {
        // not possible in standing state
    }
    public void handPlayable() {
        // not possible in standing state
    }
    public void handBlackjack() {
        // not possible in standing state
    }
    public void handBusted() {
        // not possible in standing state
    }
    public void execute( Dealer dealer ) {
        Iterator i = standing_players.iterator();
        while( i.hasNext() ) {
            Player player = (Player) i.next();
            if( player.getHand().isEqual( getHand() ) ) {
                player.standoff();
            } else if( player.getHand().isGreaterThan( getHand() ) ) {
                player.win();
            } else {
                player.lose();
            }
        }
        i = blackjack_players.iterator();
        while( i.hasNext() ) {
            Player player = (Player) i.next();
            player.win();
        }
        i = busted_players.iterator();
        while( i.hasNext() ) {
            Player player = (Player) i.next();
            player.lose();
        }
    }
}
```

لیست ۱۶-۱۵ حالت انتظار سفارشی شده واسط

```
private class DealerWaiting implements PlayerState {
    public void handChanged() {
        // not possible in standing state
    }
    public void handPlayable() {
```

لیست ۱۶-۱۵ (ادامه)

```
        // not possible in standing state
    }
    public void handBlackjack() {
        // not possible in standing state
    }
    public void handBusted() {
        // not possible in standing state
    }
    public void execute( Dealer dealer ) {
        if( !waiting_players.isEmpty() ) {
            Player player = (Player) waiting_players.get( 0 );
            waiting_players.remove( player );
            player.play( dealer );
        } else {
            setCurrentState( getPlayingState() );
            exposeHand();
            getCurrentState().execute( dealer );
            // transition and execute
        }
    }
}
```

لیست ۱۶-۱۶ حالت شرط‌بندی سفارشی شده واسط

```
private class DealerDealing implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        setCurrentState( getWaitingState() );
        // transition
    }
    public void handBlackjack() {
        setCurrentState( getBlackjackState() );
        notifyBlackjack();
        // transition
    }
    public void handBusted() {
        // not possible in dealing state
    }
    public void execute( Dealer dealer ) {
        deal();
        getCurrentState().execute( dealer );
        // transition and execute
    }
}
```

BlackjackDealer حالت بازی خود را تعریف نمی‌کند. در عوض از حالت بازی Player استفاده می‌کند. اما برای این کار BlackjackDealer باید توابع get و کلاس Player را جایگزین کند مثلاً:

```
protected PlayerState getBlackjackState() {
    return new DealerBlackjack();
}
protected PlayerState getBustedState() {
    return new DealerBusted();
}
protected PlayerState getStandingState() {
    return new DealerStanding();
}
protected PlayerState getWaitingState() {
    return new DealerWaiting();
}
```

آزمون

مانند کد فصل ۱۵، کد این فصل را هم می‌توانیم از www.samspublishing.com دریافت کنید. علاوه بر این مجموعه‌ای کامل از آزمونها برای این کد موجود است. این آزمون‌ها شامل مجموعه‌ای از آزمونهای واحد و اشیاء کاذب هستند که توسط آنها می‌توان سیستم Blackjack را کاملاً آزمایش کرد.

آزمون بخش مهمی از فرایند توسعه است. بررسی کد آزمون به عنوان تمرین به خواننده واگذار می‌شود.

خلاصه

امروز به تکرار دوم بازی Blackjack پرداختیم. از این طریق در وهله اول دریافتیم که چگونه از فرایند تکرار برای حصول به یک راه حل کامل بهره بگیریم.

هر تکرار به صورت بنیانی برای تکرار بعدی قابل استفاده است. به جای شروع کردن از صفر، امروز با طرح‌ها و تحلیل‌های دیروز کار را آغاز کردیم.

در درس فردا این ساختار را با افزودن شرط‌بندی به آن تکمیل خواهیم کرد.

پرسشها و پاسخها

اگر حالتها اینقدر اهمیت دارند، چرا قبلاً به آنها نپرداختیم؟

تکرار قبلی بسیار ساده بود. آنچه در درس دیروز طراحی شد، تنها می‌توانست باخت را تشخیص دهد. علاوه بر این لزومی نداشت یکباره با حلی پیچیده به مسأله هجوم برد. اما نیاز این تکرار راه حل پیچیده‌تری بود، زیرا تشخیص Blackjack و تعیین نتیجه را به سیستم می‌افزود.

کارگاه

پرسشها

۱. **خطر عبارات شرطی در چیست؟ دو راه برای حذف آنها پیشنهاد کنید.**

۲. کپسوله‌سازی نسخه امروز **Hand** نسبت به دیروز بهبود یافته است. این امر چگونه ممکن شده؟
۳. **Hand** و **HandListerer** چه الگویی را پیاده‌سازی می‌کنند.
٤. در مورد الگوی حالت در اینترنت جستجو کنید.

تمرین‌ها

۱. کد منبع درس امروز را از اینترنت تهیه کنید. آن را کامپایل کرده، اجرا کنید و سعی کنید بفهمید چطور کار می‌کند.
۲. کد زیر در تعریف کلاس Player موجود است.

```
protected void notifyChanged() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener) i.next();
        pl.playerChanged( this );
    }
}

protected void notifyBusted() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener) i.next();
        pl.playerBusted( this );
    }
}

protected void notifyBlackjack() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener) i.next();
        pl.playerBlackjack( this );
    }
}

protected void notifyStanding() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener) i.next();
        pl.playerStanding( this );
    }
}

protected void notifyStandoff() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
```

```
            PlayerListener pl = (PlayerListener) i.next();
            pl.playerStandoff( this );
        }
    }

    protected void notifyWin() {
        Iterator i = listeners.iterator();
        while( i.hasNext() ) {
            PlayerListener pl = (PlayerListener) i.next();
            pl.playerWon( this );
        }
    }

    protected void notifyLose() {
        Iterator i = listeners.iterator();
        while( i.hasNext() ) {
            PlayerListener pl = (PlayerListener) i.next();
            pl.playerLost( this );
        }
    }
```

این روالها کار خود را به درستی انجام می‌دهند. چگونه می‌توان از اشیاء برای جلوگیری از تکرار کد در همه روالها استفاده کرد؟ یک راه حل طراحی و سپس پیاده‌سازی کنید.

روز ۱۷

Object Oriented Programming

تکرار سوم Blackjack: اضافه کردن شرط‌بندی

در فصل قبل پیاده‌سازی ساده‌ای از بازی Blackjack را دیدید. امروز پیاده‌سازی همین بازی ولی با قابلیتهای بیشتر را با اضافه کردن شرط‌بندی ادامه می‌دهیم.

درس امروز تجربیات بیشتری از فرایندهای طراحی همچون تحلیل و طراحی شیءگرا را آموزش خواهد داد. در انتهای درس امروز با معماری مبتنی بر حالت (وضعیت) که درس روز گذشته بدان پرداخته است، به طور عملی آشنا خواهید شد. یکی از تمرین‌های مطرح شده در امروز از شما می‌خواهد تا وضعیت جدیدی را به سیستم اضافه کنید.

امروز خواهید آموخت:

- چگونه بازی Blackjack را با معماری مبتنی بر حالت توسعه دهید تا قابلیتهای آن افزایش یابد.
- مزایای طراحی درست OO و اعمال آن بر روی سیستم

شرط‌بندی در بازی

تکرار ارایه شده در درس دیروز، بازی کاملی را ارایه داده است. با این تفاوت که بازی روز گذشته شامل هیچ شرط‌بندی نبوده است. امروز این قابلیت را به سیستم اضافه خواهید کرد.

همچون دیگر درسهای ارایه شده در این هفته، از روش گفته شده در فصل نهم «مقدمه‌ای بر تحلیل شیءگرا (OOA)» پیروی خواهیم کرد.

تحلیل شرط‌بندی

برای فهم کامل شرط‌بندی باید موارد کاربردی پرداخت پول شرط‌بندی و همچنین دو برابر کردن شرط‌بندی (Double Down) را نهایی کنیم. همچنین باقی موارد کاربردی را نیز باید بررسی کنیم تا هماهنگی کامل بین همه موارد کاربردی وجود داشته باشد. با اتمام موارد کاربردی، مدل دامنه را نیز تصحیح و به‌روز خواهیم کرد.

تحلیل مورد کاربردی شرط‌بندی

اجازه دهید با مورد کاربردی شرط‌بندی یک بازیکن جدید شروع کنیم:

بازیکنان بازی را با قرار دادن پولی معادل ۱۰۰$ شروع می‌کنند. درواقع قبل از اینکه کارتها پخش شود، هریک از بازیکنان مبلغ فوق را باید پرداخت کنند.

- قرار دادن پول شرط‌بندی
  1. بازیکن مبلغی معادل ۱۰$، ۵۰$ و یا ۱۰۰$ را به عنوان پول شرط‌بندی پرداخت می‌کند.
  2. پرداخت پول شرط‌بندی توسط همه بازیکنان انجام می‌شود.
- شرایط قبل
  - بازی جدید
- شرایط بعد
  - بازیکنان پول شرط‌بندی را پرداخت کرده‌اند.

در بازی واقعی Blackjack هر بازی قانون خاص خود را دارد. این قوانین می‌توانند شامل حداقل مقدار شرط‌بندی، حداکثر مقدار شرط‌بندی و میزانی که پول شرط‌بندی در هر مورد اضافه می‌شود، باشد. در این بازی، هر بازیکن می‌تواند مقدار ۱۰$، ۵۰$ و یا ۱۰۰$ را شرط‌بندی کند. برای سادگی در این بازی، بازیکن می‌تواند میزان نامحدودی شرط‌بندی کند. مقدار پول هر بازیکن ۱۰۰۰$ است و اگر از این مقدار تجاوز شود، موجودی وی منفی می‌شود. با این حال بازی متوقف نمی‌شود و بازیکن می‌تواند تا جایی که می‌خواهد به بازی ادامه دهد.

مورد کاربردی دیگر دوبرابر کردن پول شرط‌بندی است:

اگر بازیکن احساس کند با این مقدار شرط‌بندی ارضا نمی‌شود، می‌تواند پول شرط‌بندی را دوبرابر کند. این کار موجب می‌شود در صورت برد، پول بیشتری را ببرد. در ضمن شخص واسط کارت دیگری را به بازیکن اختصاص می‌دهد.

- دوبرابر کردن پول شرط‌بندی
  1. بازیکن احساس می‌کند با این مقدار پول شرط‌بندی ارضا نمی‌شود.
  2. در این حالت تصمیم به دوبل کردن پول شرط‌بندی می‌کند.
  3. پول شرط‌بندی بازیکن دو برابر می‌شود.
  4. واسط کارت دیگری را به بازیکن می‌دهد.

- شرایط قبل

۱. بازیکن کارت جدیدی را نکشیده و یا در حالت توقف نیست.
۲. بازیکن Blackjack ندارد.
۳. واسط Blackjack ندارد.

- شرایط بعد

۱. بازیکن دارای ۳کارت است.
۲. دور بازیکن تمام می‌شود.

- حالت دیگر: باخت بازیکن

کارت جدید باعث می‌شود بازیکن ببازد.
در واقع کارت جدید باعث می‌شود مجموع ارزش کارتها بیش از ۲۱ شود.

- حالت دیگر: مجموع ارزش کارتها بیش از ۲۱ است ولی بازیکن کارت آس دارد.

در این حالت ارزش کارت آس می‌تواند یکی از دو مقدار ۱ یا ۱۱ باشد. بنابراین مجموع ارزش کارتها در این موقعیت می‌تواند کمتر و یا مساوی ۲۱ شود.

تنها موارد کاربردی از قبل موجود که باید آنها را تغییر داد، مورد کاربردی واسط و مورد کاربردی ختم بازی توسط واسط است. باقی موارد کاربردی بدون تغییر باقی می‌مانند. شخص واسط برای هر بازیکن کارتی را رو به بالا قرار می‌دهد و پخش کارتها را به خودش ختم می‌کند. سپس این کار را دوباره تکرار می‌کند ولی کارت خودش را رو به پایین قرار می‌دهد. پخش کارتها با ارایه دو کارت به هر بازیکن خاتمه می‌یابد. در این صورت واسط نیز، خود دو کارت دارد.

- پخش کارتها

۱. واسط یک کارت به هر بازیکن و منجمله خودش اختصاص می‌دهد و کارتها را رو به بالا می‌گذارد.
۲. واسط کارت دومی را به هر بازیکن اختصاص می‌دهد و کارتها را رو به بالا می‌گذارد.
۳. واسط کارتی را رو به پایین به خود اختصاص می‌دهد.

- شرایط قبل
  - تمام بازیکنان پولهای شرط‌بندی خود را پرداخته‌اند.
- شرایط بعد
  - همه بازیکنان و منجمله واسط دو کارت در دست دارند.
  - حالت دیگر: واسط Backjack دارد.

اگر ارزش کارتهای واسط ۲۱ شود و یا آنکه واسط Backjack داشته باشد، بازی بدون آنکه بازیکنان بازی کرده باشند خاتمه می‌یابد.

پخش کارتها تا زمانی که همه بازیکنان پول شرط‌بندی را نپرداخته باشند، شروع نمی‌شود. حال ببینیم

چگونه پولهای شرط‌بندی پایان بازی را تغییر می‌دهند.

پس از آنکه همه بازی خود را انجام دادند، واسط ارزش کارتهای هریک از بازیکنان و اینکه چه کسی برده و چه کسی بازنده است و یا اینکه آیا بازی متوقف شده است یا نه را مشخص می‌کند.

- ختم بازی توسط واسط

۱. واسط ارزش کارتهای بازیکن اول را تعیین کرده و آن را با ارزش کارتهای خود مقایسه می‌کند.
۲. اگر ارزش کارتهای بازیکن از واسط بیشتر باشد، در این صورت بازیکن برده است.
۳. مجموعه پولهای شرط‌بندی شده به بازیکن تعلق می‌گیرد.
٤. این کار برای همه بازیکنان تکرار می‌شود.

- شرایط قبل

۱. همه بازیکنان دور خود را انجام داده باشند.
۲. واسط نیز دور خود را به اتمام رسانیده باشد.

- شرایط بعد
- نتیجه نهایی هریک از بازیکنان مشخص شده است.
- حالت دیگر: باخت بازیکنان

واسط کارتهای بازیکنان را بررسی کرده و ارزش آنها را با ارزش کارتهای خود مقایسه می‌کند. اگر ارزش کارتهای بازیکنان از واسط کمتر باشد، بازیکنان باخته‌اند.

- حالت دیگر: توقف بازی

اگر واسط کارتهای بازیکنان را بررسی کرده و ارزش آنها با ارزش کارتهای وی برابر باشد، بازی متوقف می‌شود. هیچ پولی به هیچ بازیکنی تعلق نمی‌گیرد.

- حالت دیگر: باخت واسط

در صورتی که واسط ببازد هر بازیکنی که Blackjack داشته و یا متوقف بوده است، می‌برد. دیگر بازیکنان نیز خواهند باخت.

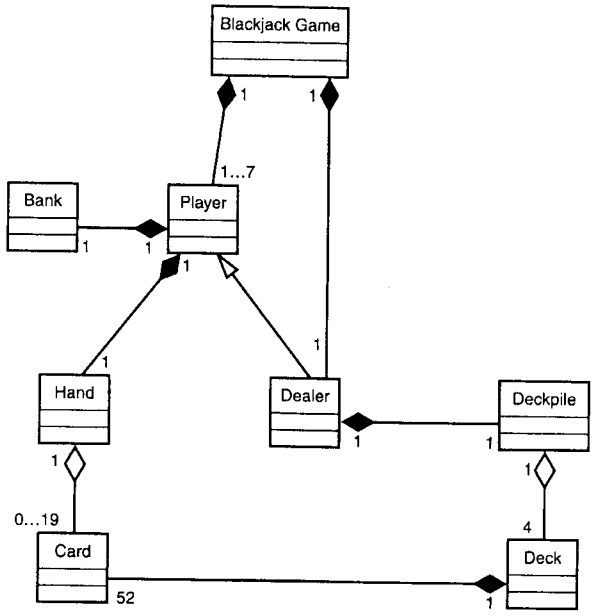
- حالت دیگر: بازیکنان با Blackjack ببرند.

اگر بازیکنی Blackjack داشته و واسط Blackjack نداشته باشد، بازیکن بازی را می‌برد و به نسبت ۳:۲ پول دریافت می‌کند (برای مثال اگر ۱۰۰$ شرط‌بندی کرده باشد، ۱۵۰$ برنده می‌شود).

با آنچه که در بالا آمد می‌توان تغییرات ایجاد شده در موارد کاربردی راکه مستقیماً به هم مرتبط می‌باشند را دید. با توجه به این تغییرات مدل دامنه را باید به‌روز کرد.

به‌روزکردن مدل دامنه

تحلیل موردکاربردی شرط‌بندی ایجاب می‌کند تا مدل دامنه را به‌روز کنیم. این کار باید بادقت انجام شود. در ضمن باید یک شیء دامنه دیگر را نیز اضافه کرد: Bank. هر بازیکن در بازی دارای یک بانک شخصی می‌باشد. شکل ۱۷ ـ ۱ مدل دامنه تغییر یافته را نشان می‌دهد.



شکل ۱۷ ـ ۱
مدل دامنه بازی Blackjack

طراحی شرط‌بندی

طراحی را باید با طراحی کلاس جدید Bank شروع کرد. با اتمام کلاس Bank باید به فکر بود که چگونه شرط‌بندی را در بازی دخالت داد. برای هدف امروز مورد کاربردی دوبل کردن مقدار شرط‌بندی (Double Down) به عنوان تمرینی برای خواننده واگذار می‌شود.

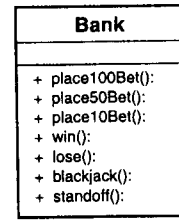
طراحی بانک

اولین کار آن است که وظایف بانک را مشخص کنیم. شکل ۱۷ ـ ۲ کارت CRC کلاس Bank را نشان می‌دهد. کلاس بانک موظف است میزان و مقدار فعالیتهای صورت گرفته در شرط‌بندی را ضبط و نگهداری کند. از طریق این کلاس می‌توان جزییات شرط‌بندی را ذخیره کرده و به پولهای شرط‌بندی دسترسی داشت. شکل ۱۷ ـ ۳ نمودار کلاس و رابطه آن با بازیکن را نشان می‌دهد.

شکل ۱۷ ـ ۲
کارت CRC کلاس Bank

Bank	
hold total $ for player	
place $100 bet	
place $50 bet	
place $10 bet	
payoff win	
payoff blackjack	
settle loss	
settle standoff	
represent itself as a string	String

شکل ۱۷ ـ ۳
نمودار کلاس Bank



طراحی شرط‌بندی

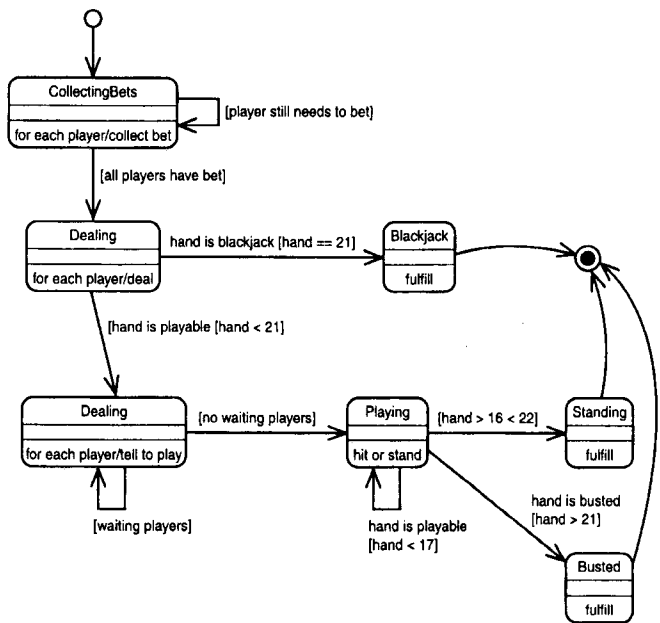
همانگونه که گفته شد تمام عملیات مربوط به شرط‌بندی باید در معماری حالت گنجانده شود. هم بازیکنان و هم واسط نیازمند حالتی اضافه برای پشتیبانی از شرط‌بندی هستند. واسط نیازمند حالتی برای ذخیره مجموعه شرط‌بندیها (CollectingBets) و بازیکنان نیازمند ذخیره شرط‌بندی خودشان هستند. شکلهای ۱۷ـ۴ و ۱۷ ـ ۵ نمودارهای حالت جدید برای واسط و بازیکنان را ترسیم کرده‌اند.

همانگونه که ملاحظه می‌کنید، بازیکنان از حالت شرط‌بندی شروع می‌کنند و این در حالی است که واسط با حالت CollectingBets شروع می‌کند. پس از آنکه همه پولها جمع شد، واسط به حالت پخش کارتها منتقل می‌شود. بازیکنان نیز پس از پرداخت پول شرط‌بندی به حالت انتظار منتقل می‌شوند.

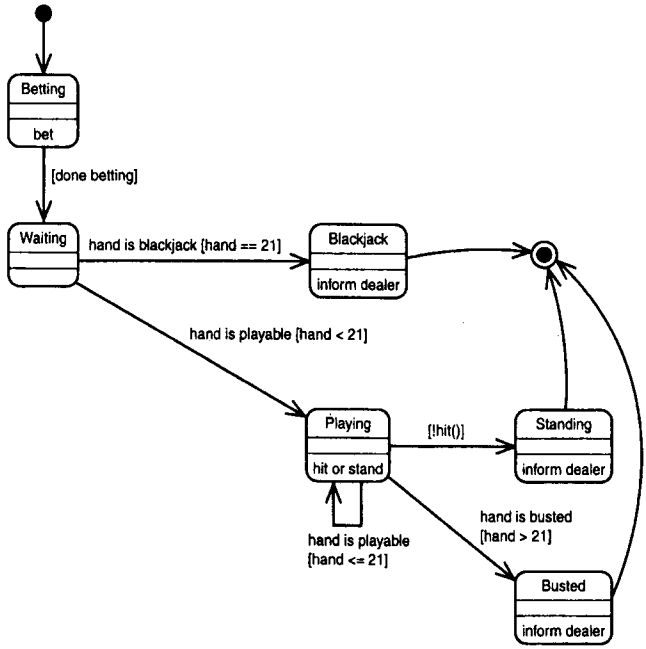
بازسازی سلسله مراتب بازیکن

در این قسمت از طراحی ملاحظه می‌شود که دو کلاس Player و BlackjackDealer از یکدیگر جدا می‌شوند. اگرچه کلاس BlackjackDealer کلاس Player را توسعه می‌دهد ولی نیازی به کلاس Bank ندارد، که برخلاف نیاز کلاس HumanPlayer است. دلیل آن هم این است که واسط شرط‌بندی نمی‌کند. اگر به کلاس Player پشتیبانی از شرط‌بندی اضافه شود، کلاس BlackjackDealer نیز آن را به ارث خواهد برد که برای آن

شکل ۱۷ ـ ۴
نمودار حالت برای واسط



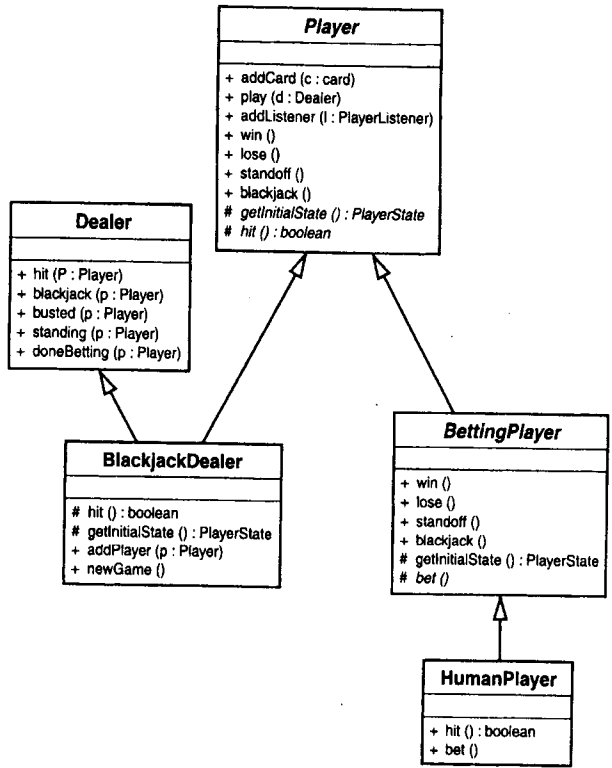
شکل ۱۷ ـ ۵
نمودار حالت برای بازیکنان



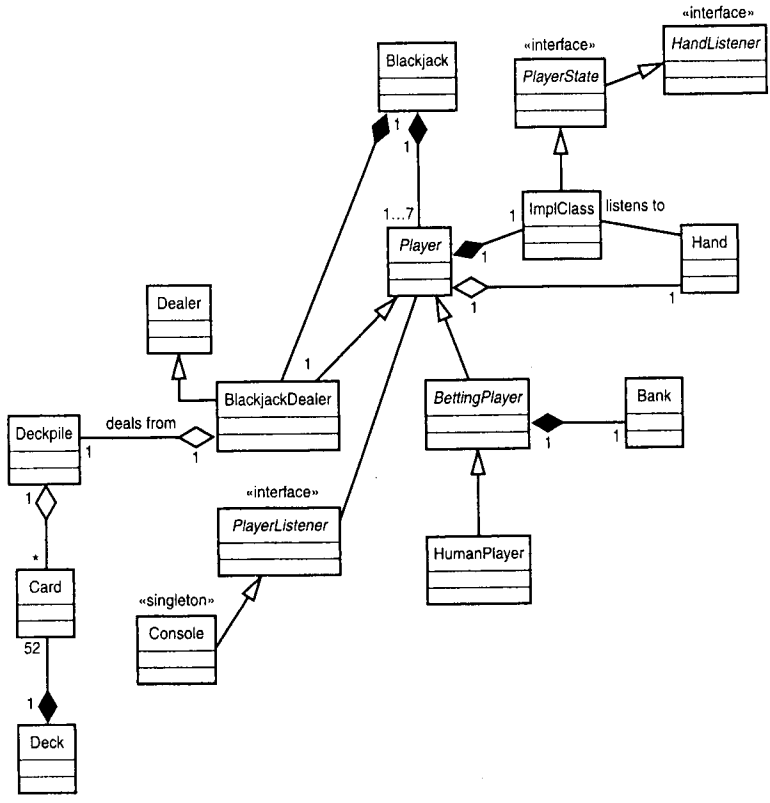
قابل استفاده نخواهد بود و در ضمن باید قابلیت فوق را جایگزین (Override) نمود.

حال زمان مناسبی است که سلسله مراتب وراثت از کلاس Player را با تقسیم عناصر مشترک و ارسال آنها

شکل ۱۷ ـ ۶
سلسله مراتب Player



شکل ۱۷ ـ ۷
نمودار کلاس
Blackjack



به زیرکلاسها بازسازی کنیم. در سلسله مراتب جدید، پشتیبانی از شرط‌بندی به کلاس پایه Player اضافه نخواهد شد. در عوض کلاس جدید BettingPlayer باید از کلاس Player مشتق شده و در ضمن حالات و متدهای مورد نیاز برای شرط‌بندی را نیز اضافه کند.

کلاس BlackjackDealer همچنان می‌تواند از کلاس Player مشتق شود و البته کلاس HumanPlayer این بار از کلاس BettingPlayer مشتق می‌شود. شکل ۱۷ ـ ۶ سلسله مراتب جدید را نشان می‌دهد.

مدل به‌روز شده Blackjack

با اتمام طراحی، ایده خوبی است که نمودار کلاس Blackjack را به‌روز کنیم. شکل ۱۷ ـ ۷ نمودار کلاس مربوطه را نشان می‌دهد. زمان آن رسیده که پیاده‌سازی را شروع کنیم.

پیاده‌سازی شرط‌بندی

برای پیاده‌سازی شرط‌بندی نیاز است کلاسهای Bank و BettingPlayer را ساخته و تغییرات لازمه را در کلاسهای BlackjackDealer، Dealer و HumanPlayer اعمال کنیم. با کلاس Bank شروع می‌کنیم.

پیاده‌سازی کلاس Bank

آنگونه که دیدید، کلاس Bank موظف به نگهداری و مدیریت شرط‌بندیها است. لیست ۱۷ ـ ۱ یک راه پیاده‌سازی ممکن را برای این کلاس نشان می‌دهد.

لیست ۱۷-۱ Bank.java

```
public class Bank {
    private int total;
    private int bet;

    public Bank( int amount ) {
        total = amount;
    }

    public void place100Bet() {
        placeBet( 100 );
}
    public void place50Bet() {
        placeBet( 50 );
    }

    public void place10Bet() {
        placeBet( 10 );
    }

    public void win() {
        total += ( 2 * bet );
        bet = 0;
    }

    public void lose() {
        // already taken out of total
        bet = 0;
    }

    public void blackjack() {
        total += ( ( ( 3 * bet ) / 2 ) + bet );
        bet = 0;
    }

    public void standoff() {
        total += bet;
        bet = 0;
    }

    public String toString() {
```

لیست ۱۷-۱ (ادامه)

```
        return ( "$" + total + ".00" );
    }

    private void placeBet( int amount ) {
        bet = amount;
        total -= amount;
    }
}
```

زمانی که بازیکنی نیاز دارد تا پولی را برای شرط‌بندی پرداخت کند، این کار از طریق کلاس Bank صورت می‌گیرد. به خاطر داشته باشید که کلاس Bank تمام جزییات شرط‌بندی را در خود دارد. زمانی که بازیکنی می‌برد، می‌بازد، کارت Blackjack را بیرون می‌کشد و یا متوقف می‌شود، Bank را خبردار می‌کند. باقی کارها را Bank انجام می‌دهد.

پیاده‌سازی BettingPlayer

کلاس BettingPlayer از کلاس Player مشتق می‌شود و وضعیت (حالت) شرط‌بندی را برای بازیکن تعریف می‌کند. لیست ۱۷ ـ ۲ کلاس BettingPlayer را نشان می‌دهد.

لیست ۱۷-۲ BettingPlayer.java

```
public abstract class BettingPlayer extends Player {

    private Bank bank;

    public BettingPlayer( String name, Hand hand, Bank bank ) {
        super( name, hand );
        this.bank = bank;
    }

    // overridden behavior
    public String toString() {
        return ( getName() + ": " + getHand().toString() + "\n" + bank.toString() );
    }

    public void win() {
        bank.win();
        super.win();
    }

    public void lose() {
        bank.lose();
        super.lose();
```

لیست ۱۷-۲ (ادامه)

```
    }

    public void standoff() {
        bank.standoff();
        super.standoff();
    }

    public void blackjack() {
        bank.blackjack();
        super.blackjack();
    }
tate getInitialState() {
        return getBettingState();
    }

    // newly added for BettingPlayer
    protected final Bank getBank() {
        return bank;
    }

    protected PlayerState getBettingState() {
        return new Betting();
    }

    protected abstract void bet();

    private class Betting implements PlayerState {
        public void handChanged() {
            // not possible in busted state
        }
        public void handPlayable() {
            // not possible in busted state
        }
        public void handBlackjack() {
            // not possible in busted state
        }
        public void handBusted() {
            // not possible in busted state
        }
        public void execute( Dealer dealer ) {
            bet();
            setCurrentState( getWaitingState() );
            dealer.doneBetting( BettingPlayer.this );
            // terminate
        }
    }
}
```

ذکر این نکته نیز الزامی است که کلاس فوق شامل متد مجرد جدیدی است که به محض آنکه فعالیت جدیدی در مورد شرط‌بندی اتفاق بیافتد، فراخوانی می‌شود: protected abstract void bet(). هریک از زیرکلاسها باید متد فوق را آنگونه که لازم است جایگزین نمایند.

تغییرات در Dealer و HumanPlayer

با مروری بر کدهای BettingPlayer متوجه شده‌اید که متد جدیدی به کلاس Dealer اضافه شده است:

```
public void doneBetting (Player p);
```

کلاس BettingPlayer متد فوق را برای اطلاع واسط از اینکه پول شرط‌بندی پرداخت شده است، فراخوانی می‌کند. از این طریق واسط می‌فهمد که کارش با این بازیکن تمام شده و باید به سراغ بازیکن بعدی برود.

لیست ۱۷ ـ ۳ کلاس HumanPlayer جدید را نشان می‌دهد.

لیست ۱۷-۳ HumanPlayer.java

```
public class HumanPlayer extends BettingPlayer {

    private final static String HIT   = "H";
    private final static String STAND = "S";
    private final static String PLAY_MSG = "[H]it or [S]tay";
    private final static String BET_MSG = "Place Bet: [10] [50] or [100]";
    private final static String BET_10  = "10";
    private final static String BET_50  = "50";
    private final static String BET_100 = "100";
    private final static String DEFAULT = "invalid";

    public HumanPlayer( String name, Hand hand, Bank bank ) {
        super( name, hand, bank );
    }

    protected boolean hit() {
        while( true ) {
            Console.INSTANCE.printMessage( PLAY_MSG );
            String response = Console.INSTANCE.readInput( DEFAULT );
            if( response.equalsIgnoreCase( HIT ) ) {
                return true;
            } else if( response.equalsIgnoreCase( STAND ) ) {
                return false;
            }
            // if we get here loop until we get meaningful input
        }
    }

    protected void bet() {
        while( true ) {
            Console.INSTANCE.printMessage( BET_MSG );
```

لیست ۱۷-۳ (ادامه)

```
            String response = Console.INSTANCE.readInput( DEFAULT );
            if( response.equals( BET_10 ) ) {
                getBank().place10Bet();
                return;
            }
            if( response.equals( BET_50 ) ) {
                getBank().place50Bet();
                return;

            }
            if( response.equals( BET_100 ) ) {
                getBank().place100Bet();
                return;
            }
            // if we get here loop until we get meaningful input
        }
    }

}
```

کلاس را HumanPlayer به جای آنکه مستقیماً از کلاس Player مشتق شود، از کلاس BettingPlayer مشتق می‌شود. این کلاس همچنین یک پیاده‌سازی برای متد ()bet ارایه کرده است. هر زمان که این متد فراخوانی شود، از طریق خط فرمان (Command Prompt) بازخوردی از کاربر دریافت می‌شود.

تغییرات در BlackjackDealer

در کلاس BlackjackDealer متد جدید ()doneBetting که در کلاس Dealer تعریف شده است، پیاده‌سازی می‌شود. با فراخوانی این متد، BlackjackDealer بازیکن را گرفته و آن را در حالت انتظار قرار می‌دهد.

کلاس BlackjackDealer همچنین حالت جدیدی را تعریف می‌کند: DealerCollectingBets. لیست ۱۷ ـ ۴ وضعیت جدید را نشان می‌دهد.

لیست ۱۷-۴ حالت جدید DealerCollectingBets

```
private class DealerCollectingBets implements PlayerState {
    public void handChanged() {
        // not possible in betting state
    }
    public void handPlayable() {
        // not possible in betting state
    }
    public void handBlackjack() {
        // not possible in betting state
    }
    public void handBusted() {
```

لیست ۱۷-٤ (ادامه)

```
        // not possible in betting state
    }
    public void execute( Dealer dealer ) {
        if( !betting_players.isEmpty() ) {
            Player player = (Player) betting_players.get( 0 );
            betting_players.remove( player );
            player.play( dealer );
        } else {
            setCurrentState( getDealingState() );
            getCurrentState().execute( dealer );
            // transition and execute
        }
    }
}
```

حالت جدید به هر بازیکن اعلام می‌کند تا پول شرط‌بندی خود را ارایه کند. تذکر آنکه این حالت به صورت تکرار و حلقه‌ای صورت نمی‌گیرد. درواقع پس از آنکه هر بازیکن پول خود را ارایه کرد، این متد اجرا می‌شود. این رفتار در داخل متد doneBetting() تعریف شده است:

```
public void doneBetting (Player player) {
    waiting_players.add(player);
  play (this);
}
```

به خاطر داشته باشید فراخوانی متد Play()، حالت (وضعیت) جاری را اجرا می‌کند.

تغییرات متفرقه

تنها تغییر باقیمانده آن است که متد getInitialState() کلاس Player به صورت مجرد تعریف شده است. با تعریف این متد به صورت مجرد، تمام زیرکلاسها باید آن را جایگزین کرده و بنابر نیاز خود آن را پیاده‌سازی نمایند.

یک آزمایش کوچک: یک شیء کاذب

آنگونه که معمول است همراه سرس کدها، روتینهای تست نیز ارایه می‌شود. لیست ۱۷ ـ ۵ نمونه‌ای را نشان می‌دهد که مطمئن می‌شود که واسط کارت Blackjack دارد یا خیر.

لیست ۱۷-۵ DealerBlackjackPile.java

```
public class DealerBlackjackPile extends Deckpile {

    private Card [] cards;
    private int index = -1;

    public DealerBlackjackPile() {
```

لیست ۱۷-۵ (ادامه)

```
        cards = new Card[4];
        cards[0] = new Card( Suit.HEARTS, Rank.TWO );
        cards[1] = new Card( Suit.HEARTS, Rank.ACE );
        cards[2] = new Card( Suit.HEARTS, Rank.THREE );
        cards[3] = new Card( Suit.HEARTS, Rank.KING );
    }

    public void shuffle() {
        // do nothing
    }

    public Card dealUp() {
        index++;
        cards[index].setFaceUp( true );
        return cards[index];
    }

    public Card dealDown() {
        index++;
        return cards[index];
    }

    public void reset() {
        // do nothing
    }

}
```

از کد فوق می‌توانید برای آزمودن بازی استفاده کنید تا مطمئن شوید اگر واسط کارت Blackjack بیاورد بازی جواب درست را ارایه خواهد کرد.

خلاصه

امروز، تکرار سوم را تکمیل کردید، یک قدم دیگر به پیش!

در این فصل دیدید که چگونه می‌توان معماری حالت (وضعیت) را توسعه داد. همچنین ملاحظه کردید که با تغییر نیازمندیها باید سلسله مراتب ایجاد شده را دوباره مرور کرده و آن را بازسازی نمود.

در روز آینده، رابط کاربری گرافیکی (GUI) را به بازی اضافه خواهیم کرد.

پرسشها و پاسخها

چرا اختتامیه بازی را به عنوان یک حالت (وضعیت) مدل نکردید؟

می‌توان پایان بازی را نیز به عنوان یک حالت مدل کرد ولی طراحی به صورت فوق فایده چندانی ندارد چرا که با مدلسازی دیگر فعالیتها نظیر باختن، آوردن Blackjack و یا حالتهایی نظیر توقف بازی می‌تواند به پایان برسد. درواقع پایان بازی به صورتی پنهانی مدل شده است.

کارگاه

سؤالاتی که در زیر آورده شده‌اند تنها برای فهم بیشتر شما از مطالب ارایه شده‌اند.

پرسشها

۱. **چگونه می‌توانید به صورت مؤثر پروتکلهای وراثت را پیاده‌سازی کنید؟**

۲. **در خلال هفته اول، درس مربوط به وراثت خاطرنشان می‌ساخت، که سلسله مراتب وراثت عموماً کشف می‌شود تا طرح‌ریزی شود. امروز چه سلسله مراتبی را کشف کردید؟**

۳. **با توجه به پرسش دوم، چرا باید مدل مجردی را طرح‌ریزی کنیم حال آنکه چنین کارهایی را چندین مرتبه انجام داده‌ایم؟**

٤. **در امروز سلسله مراتب Player را بازسازی کردیم. دو مزیتی را که از این تغییرات به دست آوردیم، ذکر کنید.**

تمرین‌ها

۱. منبع کد تکرار امروز را از اینترنت بیاورید. پس از آنکه آن را ذخیره کردید، آن را کامپایل کنید و سپس Blackjack را اجرا نمایید. سپس سعی کنید منطق آن را بفهمید. فهم کامل کد منبع وقت و حوصله می‌طلبد.

۲. مورد کاربردی دوبرابر کردن پول شرط‌بندی (Double Down) را طراحی و سپس پیاده‌سازی کنید. کار خود را بر اساس کد منبعی که در امروز ارایه شد، قرار دهید.

روز ۱۸

Object Oriented Programming

تکرار چهارم Blackjack: افزودن رابط گرافیکی کاربر (GUI)

تا به حال در درسهای این هفته برنامه بازی Blackjack را تحلیل، طراحی و اجراء کردیم. برای شروع از یک بازی ساده و افزودن قابلیتهای جدید در تکرارهای متوالی توانستیم برنامه‌ای پیچیده ایجاد کنیم. در درس امروز فرایند تکراری را ادامه می‌دهیم و لایه نمایشی برنامه را بهبود می‌دهیم.

امروز خواهیم آموخت که چگونه:

- تحلیل‌ها، طراحی‌ها و پیاده‌سازی‌ها را در رابط کاربری اعمال کنیم
- الگوی MVC را به بازی Blackjack بیافزاییم.

نمایش Blackjack

تا به حال تنها رابط بازی Blackjack، به صورت خط فرمان بوده است. چیز زیادی در مورد این UI گفته نشده است. درواقع، خیلی کم به آن پرداخته شده و تنها به این موضوع اشاره کردیم که در آن از الگوی MVC استفاده خواهیم کرد. به جای اینکه برای تحلیل و طراحی رابط تحت خط فرمان وقت تلف کنیم، ساده‌ترین UI ممکن برای آن مورد استفاده قرار گرفت و به این صورت تمرکز کار روی خود سیستم قرار گرفت.

در طیِ فرایند توسعه نرم‌افزار اغلب مجبوریم علاوه بر خود سیستم، روی نکات حاشیه‌ای کمی هم متمرکز شویم. رابط کاربر (UI) یکی از

همین نکات حاشیه‌ای است. این موارد در طراحی و تحلیل پی گرفته نمی‌شوند، بلکه در موارد لزوم در طی پیاده‌سازی به آنها پرداخته می‌شود.

در مورد Blackjack، احتیاج به راهی برای تعامل با سیستم وجود داشت، با این حال اضافه کردن GUI از ابتدا چندان عملی نیست. از آنجایی که UI خط فرمانی جزیی از سیستم نهایی نبود، نپرداختن زیاد به آن چندان عجیب نیست.

در درس امروز آخرین حک و اصلاح لازم UI خط فرمان اولیه را انجام می‌دهیم و سپس به سراغ تحلیل، طراحی و پیاده‌سازی یک رابط گرافیکی کاربر (GUI) کامل برای بازی Blackjack می‌رویم.

تکمیل رابط خط فرمان

قبل از شروع کار روی GUI اصلی بازی، بد نیست یک تغییر نهایی در رابط تحت خط فرمان ایجاد کنیم. اینکه هر بار می‌خواهید بازی کنید، بازی را از اول شروع کنید چندان جالب نیست. کد لیست ۱۸ ـ ۱ تابع main() جدیدی را ارایه می‌دهد که با آن می‌توانید تا هر چند بار که مایل باشید، بدون نیاز به شروع از اول، بازی کنید.

لیست ۱۸-۱ Blackjack.java

```
public class Blackjack {

    public static void main( String [] args ) {
        Deckpile cards = new Deckpile();
        for( int i = 0; i < 4; i++) {
            cards.shuffle();
            Deck deck = new Deck();
            deck.addToStack( cards );
            cards.shuffle();
        }

        Hand dealer_hand = new Hand();
        BlackjackDealer dealer = new BlackjackDealer( "Dealer", dealer_hand, cards );
        Bank human_bank = new Bank( 1000 );
        Hand human_hand = new Hand();
        Player player = new CommandLinePlayer( "Human", human_hand, human_bank);
        dealer.addListener( Console.INSTANCE );
        player.addListener( Console.INSTANCE );
        dealer.addPlayer( player );

        do {
            dealer.newGame();
        } while( playAgain() );

        Console.INSTANCE.printMessage( "Thank you for playing!" );

    }

    private static boolean playAgain() {
```

لیست ۱۸-۱ (ادامه)

```
        Console.INSTANCE.printMessage( "Would you like to play again? [Y]es [N]o" );
        String response = Console.INSTANCE.readInput( "invalid" );
        if( response.equalsIgnoreCase( "y" ) ) {
            return true;
        }
        return false;
    }

}
```

اعمال این تغییر دارای ارزش خاصی است چون به یافتن خطاهای پنهان در برنامه ضمن بازیهای متوالی کمک شایانی می‌کند. مثلاً، قبل از شروع دور بعدی بازی هر دست باید کاملاً خالی شود. با پیدا کردن و رفع اشکالات ممکن، در آینده غافلگیر نخواهیم شد و تقصیر را گردن GUI نخواهیم انداخت.

تحلیل GUI بازی

برای تحلیل GUI، باید مطابق تکرارهای قبلی، حالتها و موارد مختلف را آنالیز کنیم. در زمان طراحی GUI برگزاری جلسه با مشتری، کاربر و متخصصان یا همکاران بسیار مفید است. درواقع، شمای برنامه‌نویس نظر هرچه کمتر در مورد GUI بدهید بهتر است. هر کسی تخصص خود را دارد. به عنوان یک توسعه دهنده، تخصص شما در تحلیل مسایل، طرح راه حل و پیاده‌سازی آن راه حل‌ها است. وقتی با مشتری جلسه را می‌گذارید، هدف فهمیدن این است که مشتری GUI چگونه بهتر می‌پسندد.

متأسفانه، مشتری‌ها، کاربران و متخصصین را نمی‌توان به این کتاب وارد کرد، بنابراین باید امروز را بدون آنها بگذرانیم.

حالتهای GUI

برخلاف حالتهای خود بازی، حالتهای GUI دامنه Blackjack را تغییر نمی‌دهند. بلکه حالتهای GUI به بنا کردن روش دسترسی و اختیارات کاربر در طی بازی کمک خواهند کرد.

اولین حالتی که باید مورد بررسی قرار گیرد، شروع بازی است.

وقتی برنامه تازه اجرا می‌شود یا بازیکن تازه بازی را تمام کرده و می‌تواند یک بازی تازه را شروع کند.

- بازی جدید GUI

۱. بازیکن دکمه New Game را کلیک می‌کند و بازی شروع می‌شود.

- شرایط قبل
  - بازیکن یا باید تازه برنامه را اجرا کرده باشد، یا تازه یک بازی را به پایان رسانده باشد.
- شرایط بعد
  - بازی جدید آغاز می‌شود.

چنانکه می‌بینید، این حالت دامنه Blackjack را تغییر نمی‌دهد. تنها قوانین اولیه و پایه‌ای UI را تنظیم می‌کند.

حالت بعدی شرط‌بندی را تحلیل می‌کند:

بازیکنان بازی را با ۱۰۰۰$ اندوخته شروع می‌کنند. قبل از اینکه ورقی پخش شود، هر بازیکنی باید شرط‌بندی کند. شروع از بازیکن اول است و هریک می‌توانند ۱۰، ۵۰ یا ۱۰۰ دلار شرط ببندند. اگر اندوخته بازیکنی از صفر کمتر شود، باز هم می‌تواند بازی کند، اما مقدار اندوخته‌اش منفی نمایش داده خواهد شد.

- شرط‌بندی در GUI

۱. بازیکن یکی از سطوح شرط ۱۰، ۵۰ یا ۱۰۰ را انتخاب می‌کند و شرط به سرعت نمایش داده می‌شود.
۲. شرط‌بندی به بازیکن بعدی منتقل می‌شود و تا شرط‌بندی تمام بازیکنان ادامه می‌یابد.

- شرایط قبل
  - بازی جدید شروع شده
- شرایط بعد
  - بازیکن شرط‌بندی کرده
  - واسط می‌تواند شروع به پخش کند.

اکنون باید به درخواست ورق و پاس کردن بپردازیم. ابتدا به درخواست ورق می‌پردازیم:

بازیکن تصمیم می‌گیرد که ورق درخواست کند. بازیکن هنوز نباخته. اگر بازیکن نبازد، می‌تواند دوباره درخواست ورق کند یا پاس کند. اگر بازیکن نبازد، نوبت به بازیکن بعدی می‌رسد.

- درخواست ورق در GUI

۱. بازیکن از دست خود راضی نیست
۲. بازیکن دکمه Hit را کلیک می‌کند، با این کار ورق دیگری درخواست می‌کند.
۳. اگر دست بازیکن برابر یا کمتر از ۲۱ شود، می‌تواند به صورت اختیاری دوباره ورق درخواست کند یا پاس کند.

- شرایط قبل
  - مجموع خالهای دست بازیکن کمتر از ۲۱ است.
  - بازیکن Blackjack نیست.
  - پخش کننده Blackjack نشده است.
- شرایط بعد
  - یک ورق جدید به دست بازیکن اضافه می‌شود.
- حالت دیگر: بازیکن می‌بازد

کارت جدید دست بازیکن را از ۲۱ بیشتر می‌کند. بازیکن می‌بازد. نوبت به بازیکن بعدی یا پخش کننده می‌رسد.

- حالت دیگر: دست بازیکن از ۲۱ بیشتر است ولی یک ورق آس است.

ورق جدید دست بازیکن را از ۲۱ بیشتر می‌کند. بازیکن یک آس دارد. با تغییر ارزش آس از ۱۱ به ۱، دست را کمتر یا مساوی ۲۱ می‌کند. بازیکن می‌تواند درخواست ورق کند یا پاس کند. اگر بازیکن درخواست ورق نکند، باید پاس کند. حالت بعدی پاس در GUI است.

بازیکن از دستی که در اختیار دارد راضی است و پاس می‌کند.

- بازیکن پاس می‌کند.

۱. بازیکن از دستی که در اختیار دارد راضی است، پس دکمه Stand را کلیک می‌کند.

- شرایط قبل
  - بازیکن دستی کمتر یا مساوی ۲۱ دارد.
  - بازیکن Blackjack نشده است.
  - واسط Blackjack نشده است.
- شرایط بعد
  - نوبت بازیکن تمام می‌شود.

و سرانجام حالت مهم دیگر، پایان دادن به بازی توسط خود بازیکن است.
بازیکن دیگر نمی‌خواهد بازی کند و تصمیم می‌گیرد به اجرای بازی پایان بدهد.

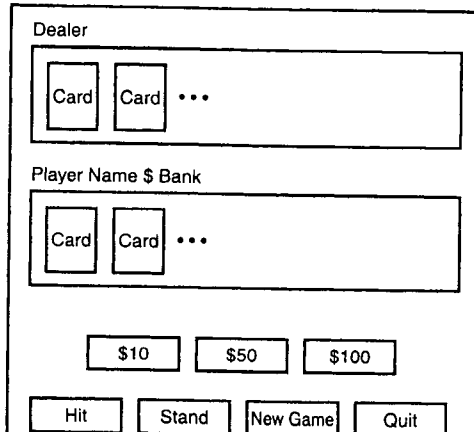
- پایان دادن به بازی در GUI

۱. بازیکن دکمه Quit را کلیک می‌کند.
۲. بازی بسته می‌شود.

- شرایط قبل
  - بازی نباید در حال انجام باشد. یعنی یا بازی تمام شده باشد، یا اصلاً بازی شروع نشده باشد.
- شرایط بعد
  - بازی بسته می‌شود.

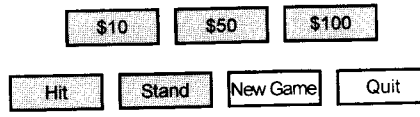
مدل نمایشی GUI

می‌توان با بهره‌گیری از حالتهایی که در قسمت قبل برشمرده شدند، به طراحی چیدمان GUI پرداخت. شکل ۱۸ ـ ۱ یک GUI پیشنهادی طرح شده بر اساس نیازمندیهای مطرح شده در تحلیل را نمایش می‌دهد.
اکنون می‌توان به رفتارهای دیگر GUI پرداخت. شکل ۱۸ ـ ۲ وضعیت دکمه‌ها را در شروع بازی نشان می‌دهد.



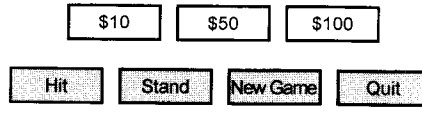
شکل ۱۸ ـ ۱
GUI بازی Blackjack

شکل ۱۸ ـ ۲
وضعیت اولیه دکمه‌ها



در ابتدای بازی تمام دکمه‌ها پدیدار هستند. اما فقط New Game و Quit فعال خواهند بود. شکل ۱۸ ـ ۳ حالت دکمه‌ها را بعد از کلیک New Game نشان می‌دهد.

شکل ۱۸ ـ ۳
حالت دکمه‌ها بعد
از کلیک New Game



بعد از شروع بازی، بازیکن باید شرط بگذارد. در نتیجه تنها دکمه‌های شرط‌بندی فعال هستند. شکل ۱۸ـ۴ حالت دکمه‌ها را بعد از شرط‌بندی نشان می‌دهد.

شکل ۱۸ ـ ۴
حالت دکمه‌ها بعد
از شرط‌بندی

بعد از شرط‌بندی، کاربر تنها می‌تواند ورق درخواست کند یا پاس کند. بنابراین فقط Hit و Stand فعال هستند. دکمه‌ها در همین حالت می‌مانند، اما تنها تا وقتی که بازیکن پاس کند یا ببازد، در چنین حالتی بازی تمام می‌شود و دکمه‌ها به حالت شکل ۱۸ ـ ۲ برمی‌گردند.

بعد از پایان بازی، کارتها تا وقتی کاربر دکمه New Game را کلیک کند، روی صفحه باقی می‌مانند. سپس کارتها از صفحه پاک می‌شوند. در طی بازی، دست گرافیکی کاربر هر بار که کارتی به کاربر داده شود، به‌روز می‌شود.

چیدمان GUI، از اشکالی که در بخش بعدی طراحی می‌کنیم تشکیل می‌شود.

طراحی GUI بازی

طراحی کلاسهایی که GUI را ایجاد می‌کنند، تفاوت چندانی با طراحی کلاسهای دیگر ندارد. باید ابتدا کلاسهای مختلف و وظایف آنها را معرفی کنید.

با استفاده از شکل ۱۸ ـ ۱ به عنوان نقطه شروع، می‌توان فهرستی اولیه از کلاسها ایجاد کرد. بنابراین برای نمایش کلی، نمایش بازی و نشان دادن انتخابهای کاربر هرکدام یک کلاس مورد نیاز است.

کارتهای CRC برای GUI

از کارتهای CRC می‌توان در اینجا استفاده برد. اجباری در کار نیست، این موضوع فقط به راحتی خودتان بستگی دارد. برای طراحی یک GUI بزرگتر بهتر است در مراحل مختلفی از آنها برای حصول اطمینان از صحت تقسیم مسئولیتها یا وظایف استفاده کرد.

GUI بازی به حدی ساده است که نیازی به یک فرایند کامل استفاده از کارتهای CRC وجود ندارد. در عوض وظایف را در اینجا لیست خواهیم کرد.

PlayerView

وظیفه این کلاس نمایش Player در بازی است. نمایش باید شامل دست بازیکن، نام او و میزان اندوخته‌اش (در صورت موجود بودن) باشد. PlayerView تنها وسیله‌ای برای نمایش است، لذا نیازی به کنترل کننده ندارد. تنها باید گوش به زنگ تغییرات Player و مترصد نمایش آنها باشد.

OptionView و OptionViewController

Optionview وظیفه نمایش انتخابهای بازیکن انسانی را بر عهده دارد. همچنین این کلاس باید به اعمال کاربر پاسخ دهد، بنابراین به کنترل کننده نیاز دارد.

CardView

مسئول نمایش اشیاء Card است. تعامل ندارد، لذا کنترل کننده هم مورد نیاز نیست. PlayerView با استفاده از این کلاس دست ورق را نشان می‌دهد.

BlackjackGUI

مسئول یکی کردن و نمایش تمام نمایش‌های دیگر است. از آنجایی که این کلاس فقط به صورت یک پوسته ساده عمل می‌کند، نیازی به کنترل کننده ندارد.

موارد متفرقه

CardView نیاز به راهی برای مرتبط کردن Card با تصاویری که به نمایش در می‌آیند دارد. می‌توان پیاده‌سازی را توسط یک ساختار شرطی عظیم انجام داد. چنین راهبردی اگر نگوییم کند، لااقل بسیار زشت است.

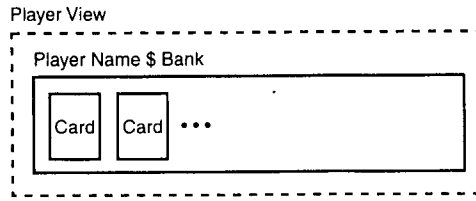
روش بهتر، مشتق کردن از Deck و Card است. دو کلاس جدید را می‌توان VDeck و VCard نامید. VCard یک آرگومان کنترلی اضافی می‌پذیرد: نام یک فایل بیت‌مپ. VDeck هم VCardها را خواهد ساخت. از آنجایی که به جای واداشتن BlackjackDealer به ایجاد کپه برای خود، Deckpile را به آن پاس کرده‌ایم، می‌توانیم کارتهای تصویری را به پخش کننده پاس کنیم.

علاوه بر اینها باید برای GUI بازیکن انسان ایجاد کنیم. این کلاس GUIPlayer جدید می‌تواند مستقیماً از BettingPlayer مشتق شود. البته باید فکری برای حالت‌های آن کرد. وقتی بازیکن در حالت خاصی قرار می‌گیرد، باید بتواند خود انتخاب کند.

در کل باید روالهای زیر را در GUIPlayer جایگزین کرد. ()Stand، ()getBettingState، ()getPlayingState، ()place10Bet، ()place50Bet، ()place100Bet و ()takeCard.

ساختار GUI

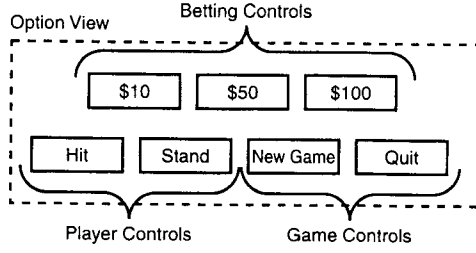
گاهی در ضمن کار با GUI کشیدن طرح اتصال اجزاء به یکدیگر مفید است. از آنجایی که خود GUI، تصویری است، کشیدن طرح آن بیشتر از دیاگرام‌های کلاس عادی به کار می‌آید. شکل ۱۸ ـ ۵ کلاس PlayerView را به تصویر می‌کشد.



شکل ۱۸ ـ ۵
تصویر PlayerView

می‌بینید که PlayerView در شکل ۱۸ ـ ۵ از تعدادی CardView تشکیل شده است. علاوه بر این PlayerView یک مستطیل به دور خود می‌کشد که نام بازیکن و اندوخته او در گوشه بالای چپ آن نوشته خواهد شد.

خوشبختانه javax.swing.JPanel قابلیت رسم و چیدمان اشکال دیگر و نیز رسم مستطیل را در خود دارد. در ادامه شکل ۱۸ ـ ۶ کلاس OptionView را به تصویر می‌کشد.

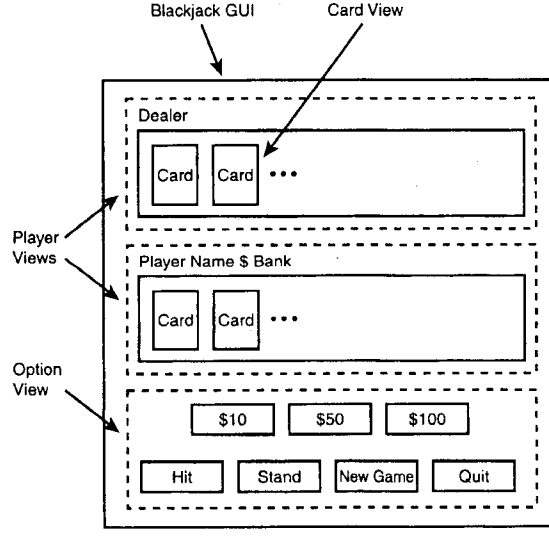


شکل ۱۸ ـ ۶
تصویر OptionView

OptionView تنها مجموعه‌ای از چند دکمه است. ترکیب javax.swing.JPanel (برای قرار دادن دکمه‌ها) و javax.swing.JButton تمام نیاز شما برای پیاده‌سازی این بخش را رفع می‌کند.

شکل ۱۸ ـ ۷ تمام اشکال را با هم ترکیب کرده است.

شکل فوق، نشان می‌دهد که چگونه تمام نمایش‌ها جزیی با هم تشکیل نمای بصری GUI را می‌دهند. درک چنین چیزی کمک بزرگی در پیاده‌سازی است.



شکل ۱۸ ـ ۷
پنجره اصلی

بازسازی

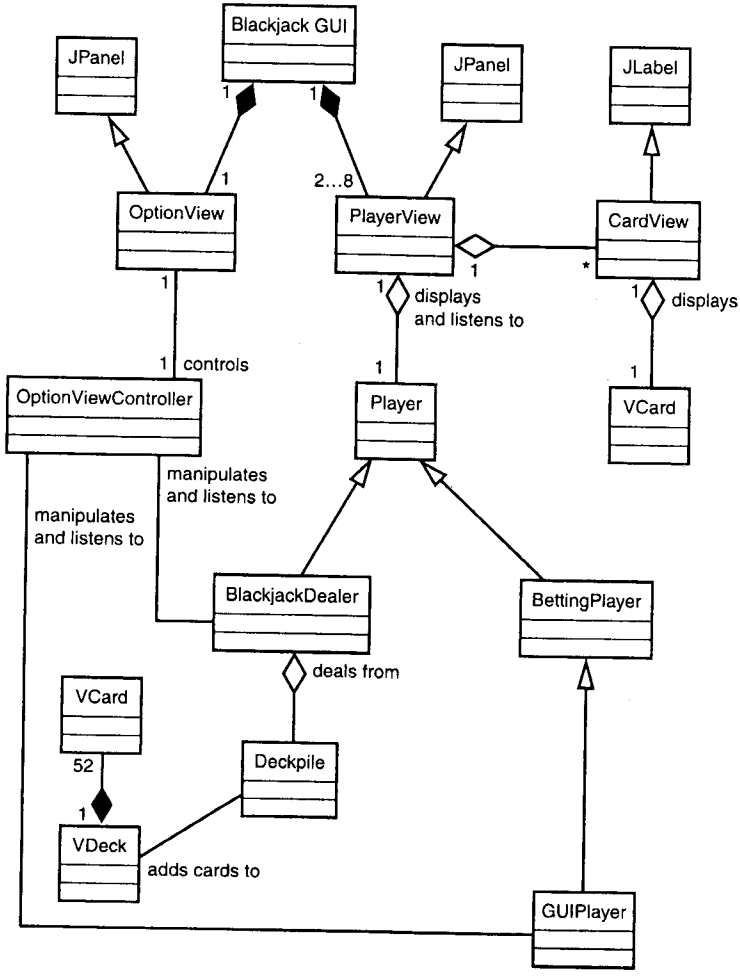
اکنون که دو نوع بازیکن انسان (GUI و CLUI) در اختیار داریم، می‌توانیم نام HumanPlayer را به CommandLinePlayer تغییر دهیم. این تغییر باید اکنون انجام شود.

دیاگرام کلاس GUI

اکنون که تمام کلاسهای جدید معرفی شده‌اند، می‌توان ساختار کلاس جدید را مدل کرد. مدل نمایش داده شده در شکل ۱۸ ـ ۸ روی ساختار تمرکز دارد.

پیاده‌سازی GUI بازی

در پیاده‌سازی GUI آسانترین روش کار از پایین به بالا است. لذا پیاده‌سازی باید به ترتیب زیر صورت پذیرد: OptionViewController، GUIPlayer، BlackjackGUI، VCard، VDeck، CardView، PlayerView و OptionView. اکنون با هم نکات مهم هر کلاس را بررسی می‌کنیم.



شکل ۱۸ ـ ۸
ساختار کلاس GUI

لیست ۱۸-۲ VCard.java

```
public class VCard extends Card {

    private String image;

    public VCard( Suit suit, Rank rank, String image ) {
        super( suit, rank );
        this.image = image;
    }

    public String getImage() {
        if( isFaceUp() ) {
            return image;
        } else {
            return "/bitmaps/empty_pile.xbm";
        }
    }

}
```

پیاده‌سازی VCard ,VDeck و CardView

پیاده‌سازی VDeck هم تقریباً به همان سادگی است. برای استفاده از VCard به جای Card باید روال buildCards()از Deck را جایگزین کنید. بنابراین باید اول آن را در Deck به صورت محافظت شده در بیاورید. روال در اصل اختصاصی بوده است. لیست ۱۸ ـ ۳ بخشی از پیاده‌سازی VDeck را نشان می‌دهد.

لیست ۱۸-۳ VDeck.java

```
public class VDeck extends Deck {

    protected void buildCards() {

        // This is ugly, but it is better than the alternative loops/if/elseif
        Card [] deck = new Card[52];
        setDeck( deck );

        deck[0] = new VCard( Suit.HEARTS, Rank.TWO, "/bitmaps/h2" );
        deck[1] = new VCard( Suit.HEARTS, Rank.THREE, "/bitmaps/h3" );
        deck[2] = new VCard( Suit.HEARTS, Rank.FOUR, "/bitmaps/h4" );
        deck[3] = new VCard( Suit.HEARTS, Rank.FIVE, "/bitmaps/h5" );
        deck[4] = new VCard( Suit.HEARTS, Rank.SIX, "/bitmaps/h6" );
        deck[5] = new VCard( Suit.HEARTS, Rank.SEVEN, "/bitmaps/h7" );
        deck[6] = new VCard( Suit.HEARTS, Rank.EIGHT, "/bitmaps/h8" );
        // rest cut for brevity
```

در GUI از گروهی تصویر بیت‌مپ (که در پوشه مربوطه از کد تهیه شده از اینترنت موجود هستند) استفاده خواهیم کرد. نام تصاویر از رسم‌الخط خاصی پیروی می‌کند. لذا می‌توان buildCard() را توسط حلقه پیاده‌سازی کرد. روش معمولی (بدون حلقه) اگرچه زشت است، اما خواناتر خواهد بود.

توجه

روش عادی مورد اشاره در بحث فوق مسلماً مناسبترین روش نیست. مسأله اساسی این است که هر روشی نقاط قوت و ضعف خود را دارد. VDeck مثالی از وقتی است که باید از بین دو شر، یکی را انتخاب کنید و با آن بسازید.

راه حل ارایه شده در بالا، به خاطر خطای موروثی در فراخوانی، دارای مشکل است. به علاوه در صورتی که تابع سازنده کوچکترین تغییری کند، مجبورید تمام فراخوانی‌ها را تغییر دهید.

راه دیگر استفاده از حلقه است. چنین راه حلی شما را مجبور می‌کند که روند خاصی را به عنوان پیش‌فرض ثابت در نظر بگیرید تا مثلاً بتوانید نام فایل‌ها را ایجاد کنید. اگر در روند مزبور کوچکترین تغییری رخ دهد، کد به صورتی اسرارآمیز بی‌ارزش خواهد شد. هر کسی که بخواهد با این کد کار کند، برای یافتن منبع خطا روزگار سختی را خواهد گذراند.

CardView بیت‌مپ‌های VCard را نمایش خواهد داد. این کار با استفاده از javax.swing.JLabel ممکن است. لیست ۱۸ ـ ۴ پیاده‌سازی CardView را نشان می‌دهد.

لیست ۱۸-٤ CardView.java

```
import javax.swing.*;
import java.awt.*;

public class CardView extends JLabel {

    private ImageIcon icon;

    public CardView( VCard card ) {
        getImage( card.getImage() );
        setIcon( icon );
        setBackground( Color.white );
        setOpaque( true );
    }

    private void getImage( String name ) {
        java.net.URL url = this.getClass().getResource( name );
        icon = new ImageIcon( url );
    }

}
```

CardView یک VCard می‌گیرد. مسیر بیت‌مپ را استخراج می‌کند، مسیر را به URL مبدل می‌کند، یک ImageIcon می‌سازد و تصویر را به خود اضافه می‌کند. تمام کاری که برای بار کردن و نمایش بیت مپ باید انجام دهید، همین است.

پیاده‌سازی PlayerView

این کلاس هر زیرکلاس Player را نمایش می‌دهد. برخلاف OptionView که آن را در بخش بعدی بررسی می‌کنیم، PlayerView فقط به نمایش Player می‌پردازد و هیچ تعاملی با کاربر ندارد. در نتیجه پیاده‌سازی آن نسبتاً ساده است. لیست ۱۸ ـ ۵ روالی که در زمان تغییر Player فراخوانی می‌شود را نشان می‌دهد.

لیست ۱۸-۵ به روز آوری کد PlayerView.java

```
public void playerChanged( Player p ) {
    border.setTitle( p.getName() );
    cards.removeAll();
    Hand hand = p.getHand();
    Iterator i = hand.getCards();
    while( i.hasNext() ) {
        VCard vcard = (VCard) i.next();
        JLabel card = new CardView( vcard );
        cards.add( card );
    }
    revalidate();
    repaint();
}
```

همانطور که می‌بینید روال playerChanged() به استخراج VCardهای Player می‌پردازد و سپس برای هر VCard یک CardView می‌سازد. در پایان، نمایش را به خودش اضافه می‌کند تا VCard نمایش داده شود.

پیاده‌سازی آمده در این قسمت بهترین پیاده‌سازی ممکن نیست، زیرا هر بار که بازیکن عوض می‌شود، برای هر VCard یک CardView ایجاد می‌کند. راه بهتر ذخیره موقت نماها است. از آنجایی که در پیاده‌سازی از اشیاء بهره گرفته‌ایم، هر زمانی که لازم باشد، می‌توانیم پیاده‌سازی را بهبود دهیم. کارایی افزوده شده در نتیجه افزودن امکان ذخیره موقت در اینجا آنقدر نیست که سربار انجام این کار را توجیه کند.

علاوه بر این PlayerView باید نتیجه بازی Player را هم نشان دهد. لیست ۱۸ ـ ۶ دو روال فراخوانی شده در انتهای بازی Player را نشان می‌دهد.

لیست ۱۸-۶ نمونه‌ای از متد PlayerListener مربوط به PlayerView

```
public void playerBusted( Player player ) {
    border.setTitle( player.getName() + " BUSTED! " );
    cards.repaint();
}

public void playerBlackjack( Player player ) {
    border.setTitle( player.getName() + " BLACKJACK! " );
    cards.repaint();
}
```

این روالها نوشته کنار لبه نما را هماهنگ با نتیجه بازی تغییر می‌دهند. PlayerListener روالهای بیشتری تعریف می‌کند، اما این روالها عیناً همین الگو را دنبال می‌کنند. اگر مشتاق دیدن فهرست کامل تغییرات هستید به سرس کد مراجعه کنید.

پیاده‌سازی OptionView و OptionViewController

OptionView از JPanel مشتق می‌شود و تعدادی دکمه به کد می‌افزاید. این کلاس گوش به زنگ مدل نمی‌ماند. بلکه این وظیفه به کنترل کننده آن سپرده شده است. کنترل کننده با توجه به شرایط مدل دکمه‌ها OptionView را فعال و غیرفعال می‌کند. هیچ یک از این کلاسها از نقطه نظر پیاده‌سازی نکته خاصی ندارند. در صورتی که به جزییات پیاده‌سازی علاقمند هستید به کد مراجعه کنید.

پیاده‌سازی GUIPlayer

این کلاس شاید جالب‌ترین کلاس در این تکرار باشد. وقتی یک GUI را پیاده‌سازی می‌کنید، باید توجه داشته باشید که اعمال کاربر پیش‌بینی شده نیستند و ممکن است هر زمانی پیش بیایند.

نوشتن یک بازیکن خط فرمان تقریباً آسان بود. تنها باید hit() یا bet() را طوری جایگزین می‌کردید که مقادیر را از خط فرمان بگیرند. از آنجایی که خط فرمان تا گرفتن ورودی مطلوب به مرحله بعدی نمی‌رود، نوشتن بازیکن بسیار ساده بود. اما نوشتن بازیکن GUI کمی مشکل‌تر است.

برخلاف بازیکن خط فرمان، GUIPlayer باید منتظر کلیک یک دکمه از طرف کاربر بماند. لذا کنترل اجرا در بیرون از بازیکن است.

با توجه به این حقیقت باید به GUI روالهایی برای فراخوانی GUIPlayer بیافزایید. لیست ۱۸ ـ ۷ روالهای شرط‌بندی که باید اضافه شوند را نشان می‌دهد.

لیست ۱۸-۷ شرط‌بندی GUIPlayer

```
// these bet methods will get called by the GUI controller
// for each: place the proper bet, change the state, let the
// dealer know that the player is done betting
public void place10Bet() {
    getBank().place10Bet();
    setCurrentState( getWaitingState() );
    dealer.doneBetting( this );
}

public void place50Bet() {
    getBank().place50Bet();
    setCurrentState( getWaitingState() );
    dealer.doneBetting( this );
}

public void place100Bet() {
    getBank().place100Bet();
    setCurrentState( getWaitingState() );
    dealer.doneBetting( this );
}
```

کاملاً واضح است که تمام روالها باید شرط را تعیین کرده و کاربر را به حالت مناسب ببرند. لیست ۱۸ ـ ۸ روالهای درخواست ورق و پاس کردن را نشان می‌دهد.

لیست ۱۸-۸ کشیدن و توقف GUIPlayer

```
// takeCard will get called by the GUI controller when the player
// decides to hit
public void takeCard() {
    dealer.hit( this );
}

// stand will get called by the GUI controller when the player chooses
// to stand, when standing change state, let the world know, and then
// tell the dealer
public void stand() {
    setCurrentState( getStandingState() );
    notifyStanding();
    getCurrentState().execute( dealer );
}
```

از آنجایی که حالت نمی‌تواند مستقیماً و بدون مقدمه ()hit یا ()bet را فراخوانی کند، باید حالتهای سفارشی Playing و Betting جدیدی به وجود آورید. لیست ۱۸ ـ ۹ روالهای تغییر یافته را نشان می‌دهد.

لیست ۱۸-۹ گیرنده‌های حالت جایگزین‌شده در GUIPlayer

```
protected PlayerState getPlayingState() {
    return new Playing();
}

protected PlayerState getBettingState() {
    return new Betting();
}
```

با جایگزینی این روالها، GUIPlayer می‌تواند حالتهای سفارشی خاص خود را ایجاد کند. لیست ۱۸ ـ ۱۰ حالت سفارشی Playing را نشان می‌دهد.

نکته روالهایی مانند ()getPlayingState و getBettingSatate روالهایی عامل هستند.

لیست ۱۸-۱۰ حالت بازی‌کردن سفارشی GUIPlayer

```
private class Playing implements PlayerState {

    public void handPlayable() {
        // do nothing
    }

    public void handBlackjack() {
        setCurrentState( getBlackjackState() );
        notifyBlackjack();
```

لیست ۱۸-۱۰ (ادامه)

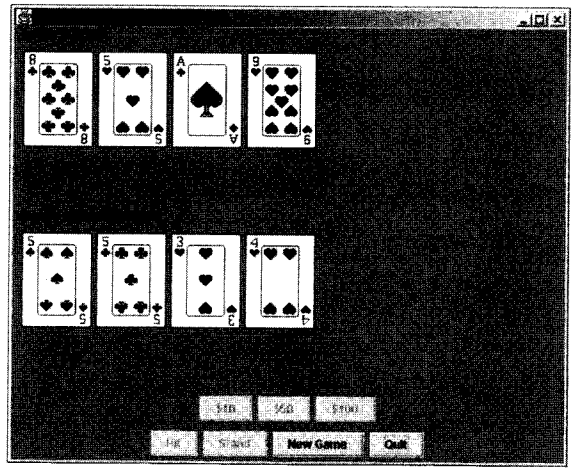
```
        getCurrentState().execute( dealer );
    }

    public void handBusted() {
        setCurrentState( getBustedState() );
        notifyBusted();
        getCurrentState().execute( dealer );
    }

    public void handChanged() {
        notifyChanged();
    }

    public void execute( Dealer dealer ) {
        // do nothing here, actions will come from the GUI which is
        // external to the state, but when events do come in be sure to
        // force state transition right away
    }
}
```

در زمان اجرا این حالت بازی تغییر یافته کار خاصی انجام نمی‌دهد. بلکه GUIPlayer منتظر تعامل از سوی GUI می‌ماند. می‌توان دید که حالت بازی هنوز در پاسخ به رویدادهای Hand منتقل می‌شود.

لیست ۱۸ ـ ۱۱ حالت سفارشی شرط‌بندی را نشان می‌دهد. این حالت کار خاصی انجام نمی‌دهد بلکه GUIPlayer منتظر کلیک دکمه‌ای در GUI می‌شود تا روال شرط‌بندی مناسب را فعال کند.

لیست ۱۸-۱۱ حالت شرط‌بندی سفارشی GUIPlayer

```
private class Betting implements PlayerState {
    public void handChanged() {
        // not possible in busted state
    }
    public void handPlayable() {
        // not possible in busted state
    }
    public void handBlackjack() {
        // not possible in busted state
    }
    public void handBusted() {
        // not possible in busted state
    }
    public void execute( Dealer dealer ) {
        // do nothing here, actions will come from the GUI which is
        // external to the state, since no events come in as part of
        // betting the state will need to be changed externally to this state
    }
}
```



شکل ۱۸ ـ ۹
GUI بازی Blackjack

نهایی کردن کار در BlackjackGUI

BlackjackGUI سیستم Blackjack را ایجاد کرده و آن را نمایش می‌دهد. لیست ۱۸ ـ ۱۲ روال setup() این کلاس را نشان می‌دهد.

لیست ۱۸-۱۲ متد setup() کلاس BlackjackGUI

```
private void setUp() {
    BlackjackDealer dealer = getDealer();
    PlayerView v1 = getPlayerView( dealer );

    GUIPlayer human = getHuman();
    PlayerView v2 = getPlayerView( human );

    PlayerView [] views = { v1, v2 };
    addPlayers( views );

    dealer.addPlayer( human );

    addOptionView( human, dealer );
}
```

روال setup() بازیکنان را ایجاد می‌کند، همه چیز را نمایش داده و اداره می‌کند. روال‌های دیگر اشیاء گوناگون گوناگونی را ایجاد می‌کنند. شکل ۱۸ ـ ۹ صفحه نهایی بازی را نشان می‌دهد.

خلاصه

امروز اعمال یک الگوی MVC به یک برنامه واقعی را مشاهده کردید. گاهی برای درک کامل یک الگو باید یک مثال را کاملاً بررسی کرد. درس امروز همچنین مشخص کرد که GUI یک کار تکمیلی نیست. سعی کنید، در طراحی و تحلیل آن دقت کافی را بکنید.

درس امروز بازی Blackjack را تکمیل کرد. در درس فردا طراحی و پیاده‌سازی یک GUI دیگر برای این بازی را بررسی خواهیم کرد.

پرسشها و پاسخها

قبلاً اشاره شد که نباید طرح GUI را به انتهای کار واگذار کرد، در حالی که در عمل در آخرین تکرار به آن پرداختیم. چرا؟

اصلاً چنین نیست! اگر یادتان باشد قبلاً درباره شکل کلی GUI و لزوم استفاده از MVC صحبت کردیم. تحلیل‌ها و طراحی‌های قبلی هم برای رسیدن به سطح طرح GUI لازم بودند.

اجزای مختلف MVC چه هستند؟

خود مدل سیستم است. در مورد BlackjackDealer، BettingPlayers و... لایه مدل را می‌سازند. طرح فقط یک کنترل کننده OptionViewcontroller را فراخوانده است. لذا چیز زیادی در مورد کنترل کننده‌ها مطرح نشد.

کارگاه

پرسشها

۱. **چگونه از وراثت و چندشکلی بودن در معرفی یک ورق بصری کمک می‌گیرید؟**

۲. **در بحث وراثت مطرح شد که تنها روالهایی باید محافظت شده معرفی شوند که زیرکلاسی بخواهد از آنها استفاده کند. در غیر این صورت حتی‌الامکان اختصاصی تعریف می‌شوند. اگر زمانی لازم شد، یک روال اختصاصی توسط زیرکلاسی جایگزین شود، باید آن را درست در همان زمان و نه زودتر مبدل به حفاظت شده نمود. مثالی از این مورد را درس امروز بیابید.**

تمرین‌ها

۱. کد درس امروز را تهیه کنید. کد به دو پوشه تقسیم شده: mvc_gui و exercise_2
mvc_gui حاوی کد GUI درس امروز است. exercise_2 حاوی فایلهایی است که در تمرین ۲ به آنها نیاز دارید است. کد mvc_gui را بررسی کنید. سعی کنید طرز کار آن را درک کنید و تمرین ۲ را تکمیل کنید.

۲. تمرین ۲ فصل ۱۷ را برای درس امروز تکرار کنید.

روز ۱۹

اعمال روشی متفاوت با MVC

دیروز تحلیل، طراحی و پیاده‌سازی GUI را برای بازی Blackjack انجام دادیم. امروز می‌خواهیم روشی متفاوت با MVC را برای طراحی و پیاده‌سازی رابط گرافیکی کاربر این بازی به کار ببندیم.

آنچه امروز خواهید آموخت:

- در مورد روشی متفاوت با الگوی طراحی MVC
- چگونه این روش متفاوت را به GUI بازی Blackjack اعمال کنیم.
- چه زمانی GUI را مبتنی بر MVC و چه زمانی مبتنی بر روشی دیگر قرار دهیم.

یک رابط گرافیکی دیگر برای Blackjack

در درس دیروز GUI بازی را بر اساس الگوی طراحی MVC ساختیم. MVC یک روش طراحی GUI است. در امروز GUI بازی را بر اساس روش دیگری طراحی و پیاده‌سازی خواهیم کرد.

روشی که امروز به کار خواهیم بست، تکنیک ویژه‌ای از الگوی طراحی کنترل انتزاعی (مجرد) نمایش، یا به طور خلاصه PAC است. همچون الگوی طراحی MVC، الگوی طراحی PAC، طراحی GUI را به سه قسمت تقسیم می‌کند:

- لایه نمایش (Presentation Layer) که نمایش سیستم را بر عهده دارد.

- لایه انتزاعی (مجرد) (Abstraction Layer) که نمایانگر سیستم است.
- لایه کنترل (Control Layer) که وظیفه اتصال همه اجزای لایه نمایش را برعهده دارد.

لایه‌های PAC

لایه انتزاعی PAC شبیه لایه مدل (Model Layer) در الگوی MVC است. لایه انتزاعی محل نگهداری تمام قابلیتهای اصلی سیستم (هسته سیستم) است. درواقع لایه نمایش این هسته را نمایش می‌دهد. لایه انتزاعی همچنین مسئول فراهم آوردن شرایطی برای دسترسی به اشیاء در سطح نمایش است.

در الگوی PAC قابلیتهای نمایش (View) و کنترل کننده (Controller) در MVC از یکدیگر جدا نشده‌اند. در عوض این دو جز با هم ادغام شده و درون لایه نمایش (Presentation) قرار گرفته‌اند. درواقع لایه نمایش مسئولیت نمایش و بکارگیری لایه انتزاعی را علاوه بر پاسخگویی به تعاملات کاربر بر عهده دارد.

از آنجا که لایه‌های نمایش و کنترل کننده MVC در لایه نمایش ادغام شده‌اند، لایه کنترل وظیفه‌ای کاملاً مجزا در الگوی PAC بر عهده دارد. در الگوی PAC، لایه کنترل وظیفه گردآوری و به خدمت گرفتن لایه‌های نمایش مختلف است. لایه کنترل همچون MVC موظف به دریافت تعاملات کاربر و پاسخگویی به آنها نیست.

فلسفه PAC

الگوی طراحی MVC از آنجا که تمام اجزا را در طراحی به طور کامل از یکدیگر جدا می‌سازد، جایگاه فوق‌العاده‌ای دارد. زمانی که از الگوی MVC استفاده می‌کنید، به راحتی می‌توانید لایه‌های نمایش جدید را به خدمت بگیرید و یا آنها را به سرعت عوض کنید، بدون آنکه به سیستم خدشه‌ای وارد شود. فصل سیزدهم «شیءگرایی و برنامه‌نویسی رابط کاربر» به طور مفصل موارد فوق را تشریح کرده است. با این حال آزادی عمل بیشتر به قیمت از دست دادن کپسوله‌سازی تمام می‌شود.

روش الگوی PAC به طور کلی با آنچه در بالا گفته شده متفاوت است. در PAC لایه‌های نمایش و انتزاعی از یکدیگر جدا نیستند. در عوض این دو لایه با یکدیگر چفت شده‌اند. این بدان معنا نیست که برای مثال بگوییم کلاس Player به طور مستقیم کلاس JComponent را توسعه می‌دهد. آنچه می‌توان گفت آن است که لایه انتزاعی، نمایش خود را می‌سازد. بنابراین کلاس Player و نمایشهای آن همچنان روشی متمایز و جدا از هم هستند.

برای دستیابی به نمایش متفاوت در لایه انتزاعی باید تعریف آن را به نحوی تغییر داد تا شیء نمایشی دیگری را بازگرداند. البته این امر تا حدودی مشکل است که بتوان نمایش این لایه را عوض کرد و یا مکانیزمی ارایه کرد تا دو نمایش از یک سیستم را فراهم کند. ساخت GUI اگرچه ساده است، در این حالت کنترل کننده از همه اعضای لایه انتزاعی می‌خواهد که خود را نمایش دهند و درواقع این امر موجب می‌شود که همه آنها به نحوی خود را بر روی صفحه نمایشگر نشان دهند. دیگر آنچنانکه در MVC دیدیم نمایش (View) و کنترل کننده‌ای (Controller) وجود ندارد که به یکدیگر متصل شوند.

چه زمانی از الگوی طراحی PAC استفاده کنیم

زمانی که نیاز نباشد برای یک سیستم از نمایشهای متفاوت (GUIهای مختلف) استفاده کنیم، می‌توانیم از

الگوی PAC بهره ببریم. در این حالت باید مطمئن شوید که سیستم تنها شامل یک رابط کاربری است. در این حالت است که PAC جایگزین بسیار خوبی برای MVC است.

الگوی PAC مزایای چندی دارد. از آنجا که لایه انتزاعی نمایش خود را به وجود می‌آورد، دیگر دلیلی برای تخریب قابلیت کپسوله‌سازی سیستم وجود ندارد. در عوض باید کلاسهای نمایشی را به عنوان کلاسهای درونی تعریف کنید. به عنوان کلاسهای درونی، این کلاسها می‌توانند به تمام اعضای کلاس والد در لایه انتزاعی دسترسی کامل داشته باشند. درواقع زمانی که نیاز به نمایش قسمتی باشد، کافی است مستقیماً سراغ حالت کلاس والد رفته و آن را نمایش دهد.

الگوی PAC ارتباط میان لایه انتزاعی و نمایش را بسیار ساده می‌کند. با تغییر لایه انتزاعی، سریعاً متدهای به‌روزرسانی (update) از کلاسهای مربوط به نمایش فراخوانی می‌شوند.

تحلیل رابط گرافیکی کاربر بازی Blackjack با استفاده از الگوی PAC

برای اعمال الگوی طراحی PAC به رابط گرافیکی کاربر بازی Blackjack می‌توانید از تحلیلی که در روز گذشته ارایه شد، استفاده کنید. چیزی از تحلیل تغییر پیدا نمی‌کند چرا که تصمیم گرفته‌اید تنها به جای استفاده از الگوی طراحی MVC از الگوی طراحی PAC استفاده کنید!

طراحی رابط گرافیکی کاربر بازی Blackjack با استفاده از الگوی PAC

برای بازی Blackjack تنها یک رابط اصلی وجود دارد: GUI. درواقع نمی‌خواهید رابط کاربری این بازی را به صورت HTML و برای وب عرضه کنید (اگرچه می‌توانید آن را به سادگی تبدیل به یک Applet کنید) و یا آنکه آن را برای PDA آماده کنید. بنابراین برای این بازی تنها یک رابط کاربری لازم است.

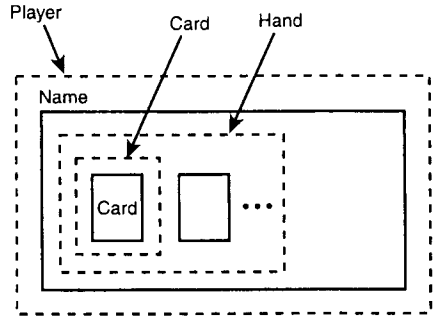
ذکر این نکته ضروری است که هیچ فشاری بر روی شما نیست تا مکانیزم گیرندگی که هم اکنون در سیستم جاسازی شده است را حذف کنید. همچنان نیز می‌توان رابط کاربری خط فرمان (Command Line) داشت. در حقیقت با استفاده از زیرکلاسها می‌توان تعاریف کلاسهای اولیه را تحت تأثیر قرار داد. زمانی که نیاز به GUI دارید کافی است از زیرکلاسهایی استفاده کنید که از GUI پشتیبانی می‌کنند و هر زمان که به رابط کاربری خط فرمان نیاز داشته باشید می‌توانید از کلاسهای قدیمی استفاده کنید. انتخاب الگوی MVC و یا PAC باعث نمی‌شود که از کلاسهای دیگر استفاده نکنید!

آنچنانکه در فصل ۱۸ «تکرار چهارم Blackjack: اضافه کردن GUI» دیدید از طراحی و پیاده‌سازی که در فصل ۱۷، «تکرار سوم: اضافه کردن قابلیت شرط‌بندی» ارایه شده بود، استفاده شد. تنها چیزی که نیازی دارید تشخیص آن است که چه کلاسهایی نیازمند اشیاء نمایشی مختص به خود هستند. بعد از تشخیص این کلاسها، باید لایه انتزاعی را طراحی کنید. پس از طراحی لایه انتزاعی نوبت به طراحی لایه کنترل می‌شود.

تشخیص اجزای لایه نمایش

برای تشخیص اجزای لایه نمایش، بهترین کمک آن است که شمایی از GUI را ترسیم کنید. در این حالت باید قسمتهای نمایشی را با کلاسهای مربوطه پیوند دهید (به جای آنکه هریک از آنها را با یک نمایش (View) جداگانه پیوند دهید)

شکل ۱۹ ـ ۱
یک قسمت از صفحه نمایش



شکل ۱۹ ـ ۱ یک قسمت از GUI را ایزوله کرده است. با تکه تکه کردن قسمتهای روی صفحه نمایش، می‌توان بعضی از اجزای لایه نمایش را تشخیص داد.

با تقسیم صفحه نمایش، می‌توان فهمید که Card، Hand و Player نیازمند اشیاء (یا کلاسهایی) برای نمایش خود هستند.

شکل ۱۹ ـ ۲ باقی قسمتهای صفحه نمایش را تقسیم می‌کند.

همه دکمه‌ها مربوط به کلاس HumanPlayer می‌شود بنابراین کلاسی که مسئول نمایش آن است باید شامل دکمه‌ها باشد. کلاس GUIPlayer همان طراحی دارد که در فصل ۱۸ ساخته شد. بنابراین از تکرار طراحی صرفنظر شده و پیشنهاد می‌گردد خواننده طراحی این کلاس را در همان فصل مطالعه کند. تنها تفاوت GUIPlayer فصل هجدهم و این فصل در این است که این کلاس باید قابلیتهای نمایش خود را نیز فراهم کند.

طراحی اجزای لایه انتزاعی

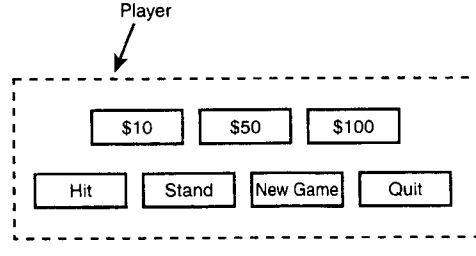
از بخش قبل دیدید که تشخیص ما منجر به شناخت کلاسهای Card و Hand و زیرکلاسهای مختلف Player شد که باید برای خودشان مکانیزمی ارایه دهند تا آنها را نمایش دهد.

برای هریک از کلاسها نیاز به ساخت یک زیرکلاس مجرد دارید. به عبارت دیگر کلاسهای Hand، BlackjackDealer، BettingPlayer و Card را باید توسعه دهید. به علاوه آنکه باید کلاس GUIPlayer را آنگونه که در فصل ۱۸ گفته شد، بسازید. در ضمن کلاس GUIPlayer فوق باید قابلیت نمایش خودش را نیز داشته باشد.

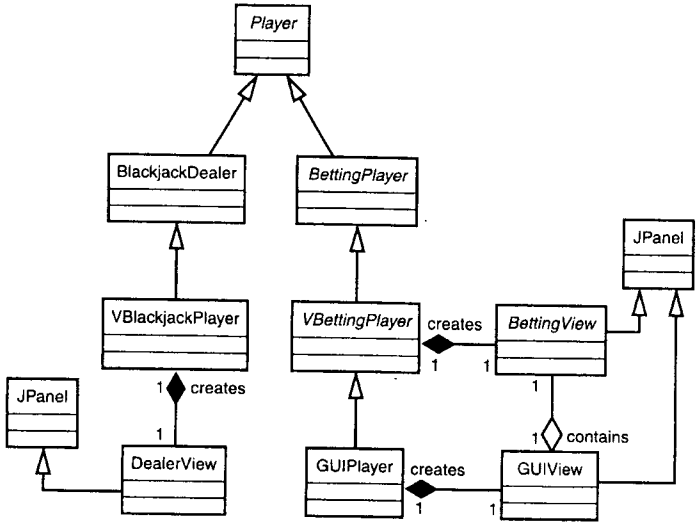
شکل ۱۹ ـ ۳ سلسله مراتب وراثت از کلاس Player را نشان می‌دهد.

برای پیاده‌سازی زیرکلاسها، زیرکلاس VBlackjackDealer را از کلاس BlackjackDealer و VBettingPlayer را از کلاس BettinPlayer می‌سازیم. در ضمن هریک از آنها اشیاء مربوط به نمایش خود را ساخته و برمی‌گردانند.

شکل ۱۹ ـ ۲
دکمه‌های GUI



شکل ۱۹ ـ ۳
سلسله مراتب کلاس Player



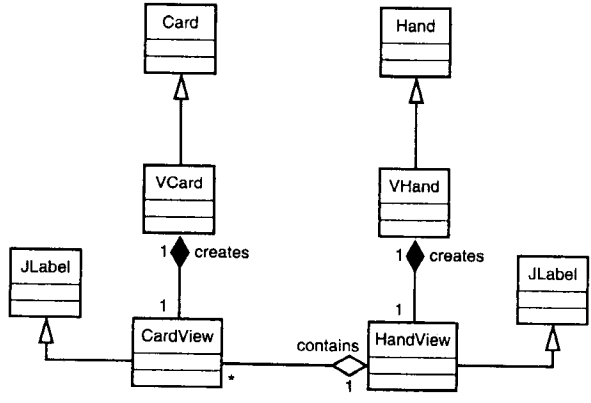
شکل ۱۹ ـ ۴ سلسله مراتب کلاسهای Hand و Card نتیجه را نشان می‌دهند.

زیرکلاسهای VHand و VCard موظف به نمایش Card و Hand هستند. همچون طراحی دیروز، نیازمند به کلاس VDeck برای نمایش کلاس مربوط به Deck نیز هستیم.

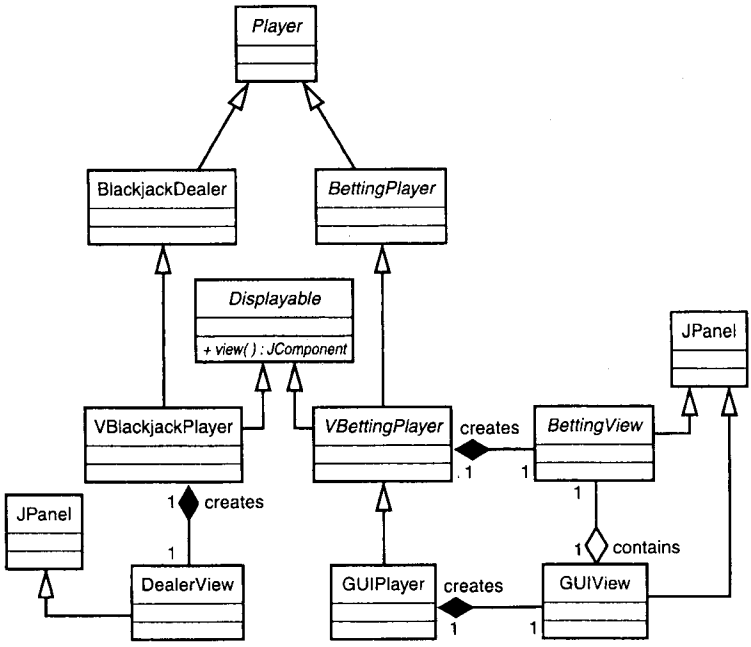
طراحی کنترل

کنترل کلاس نسبتاً ساده است. نمونه ایجاد شده از کلاس کنترل، VBlackjackDealer را به همراه کلاسهای مربوط به دیگر بازیکنان را دریافت کرده و از هریک از آنها درخواست می‌کند تا شیء مربوط به نمایش خود را ارایه کنند. سپس شیء نمایشی هریک را گرفته و آنها را نمایش می‌دهد. بنابراین نیاز به مکانیزمی هست تا از لایه انتزاعی درخواست اشیاء نمایشی کند. ساده‌ترین راه تعریف یک رابط (interface) است. فرض کنید این رابط Displayable نام داشته باشد. بنابراین Displayable تنها یک متد دارد: ()public JComponent view که شیء نمایشی را دریافت می‌کند. هریک از کلاسهای انتزاعی که احتیاج به نمایش داشته باشند باید این متد را پیاده‌سازی کنند.

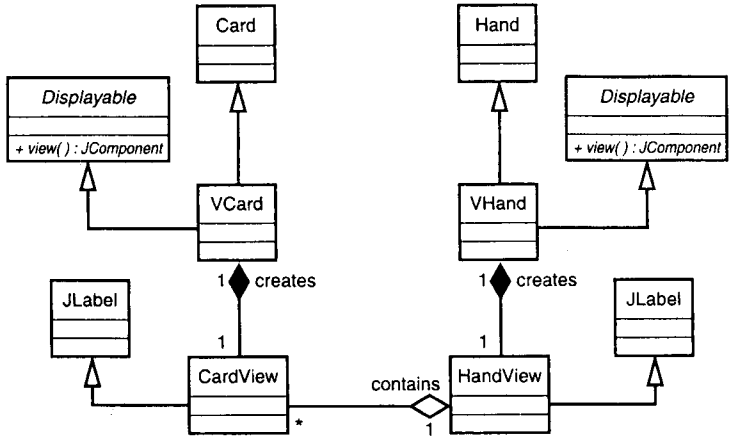
شکل ۱۹ ـ ۴
سلسله مراتب کلاس
Card و Hand



شکل ۱۹ ـ ۵
سلسله مراتب تغییر
یافته Player



شکل ۱۹ ـ ۶
سلسله مراتب تغییر
یافته برای Hand و Card



تصاویر ۱۹ ـ ۵ و ۱۹ ـ ۶ سلسله مراتب تغییر یافته را نشان می‌دهد. کلاسهای انتزاعی هم اکنون رابط Displayable را محقق می‌سازند.

استفاده از الگوی Factory جهت پیشگیری از خطاهای مشترک

تنها یک مشکل برای سلسله مراتب نشان داده شده وجود دارد: هیچ چیزی نمی‌تواند شما را از ساخت Deskpile از کلاسهای Card قدیمی که نمایش داده نمی‌شوند، جلوگیری کند. اگرچه محدودیتی برای شماایجاد نمی‌شود با این حال ممکن است به خاطر مخلوط کردن کلاسهای با GUI و کلاسهای بدون GUI با خطاهای در زمان اجرا (Run-time error) مواجه شوید.

فصل ۱۲ شما را با الگوی طراحی Factory آشنا کرد. یک دلیل استفاده از این الگو آن است که مجموعه‌ای از اشیاء از یکدیگر استفاده می‌کنند و این امر موجب پیشگیری از هرگونه خطایی بـه دلیـل اتـصال اشـیاء ناسازگار با یکدیگر می‌شود.

با استفاده از الگوی Factory می‌توان مطمئن بود که اشیاء به طور صحیح از یکدیگر استفاده می‌کنند. بنابراین باید عاملی (factory) ساخت که VBlackjackDealer و GUIPlayer که با آرگومانهای درست ایجاد شده‌اند، را برگرداند. زمانی که کنترل می‌رود تا شیئی از نوع بازیکن و یا واسط را دریافت کند، این کار را از طریق الگوی عامل انجام می‌دهد. این امر موجب می‌شود که همه کارها به طور صحیح و بدون نقص صورت پذیرد.

پیاده‌سازی رابط گرافیکی کاربر بازی Blackjack با استفاده از الگوی PAC

از درس گذشته به خاطر دارید که پیاده‌سازی رابط گرافیکی از پایین به بالا عموماً ساده‌تر است. با توجه به ایـن نکـته پـیاده‌سازی کـلاسها را بـه تـرتیب زیـر انـجام خـواهـیم داد: VCard، VHand، VBettingPlayer، VBlackjackGUI و در انتها GUIPlayer.

پیاده‌سازی VCard و VHand

کلاس VCard از کلاس Card مشتق شده و وظیفه نمایش خودش را از طریق کلاس درونی CardView برعهده دارد. لیست ۱۹ ـ ۱ پیاده‌سازی VCard را نشان می‌دهد.

لیست ۱۹-۱ VCard.java

```
public class VCard extends Card implements Displayable {

   private String image;
   private CardView view;

   public VCard( Suit suit, Rank rank, String image ) {
      super( suit, rank );
      this.image = image;
      view = new CardView( getImage() );
   }

   public void setFaceUp( boolean up ) {
      super.setFaceUp( up );
      view.changed();
   }

   public JComponent view() {
      return view;
   }

   private String getImage() {
      if( isFaceUp() ) {
```

لیست ۱۹-۱ (ادامه)

```
            return image;
        } else {
            return "/bitmaps/empty_pile.xbm";
        }
    }

    private class CardView extends JLabel {

        public CardView( String image ) {
            setImage( image );
            setBackground( Color.white );
            setOpaque( true );
        }

        public void changed() {
            setImage( getImage() );
        }

        private void setImage( String image ) {
            java.net.URL url = this.getClass().getResource( image );
            ImageIcon icon = new ImageIcon( url );
            setIcon( icon );
        }

    }

}
```

این نوع پیاده‌سازی کلاس VCard شبیه پیاده‌سازی قبلی است به جز آنکه کلاس CardView در پیاده‌سازی قبلی وجود نداشت. به محض ایجاد کلاس VCard و ساخت شیء از آن، کلاس فوق نمایش از خودش را نیز می‌سازد.

ذکر این نکته نیز ضروری است که خاصیت جدید image به طور کامل در داخل خود کلاس VCard کپسوله‌سازی شده است. برای شیء خارجی که بخواهد تصویر را نشان بدهد باید از VCard درخواست نمایش کند.

در هر حال هر زمان که کارتی از دور خارج شود (و کلاً با هر تغییر) کلاس VCard به طور خودکار از لایه‌نمایش خود می‌خواهد تا خود را به‌روز (update) کند. این کار از طریق فراخوانی متد changed() صورت می‌گیرد. برخلاف الگوی MVC تمام کنترل داخل خود لایه انتزاعی وجود دارد.

کلاس VHand نیز شبیه VCard است. به محض ساخت نمونه‌ای از کلاس VHand، کلاس فوق نمایش از خودش را می‌سازد. لیست ۱۹ ـ ۲ پیاده‌سازی کلاس VHand را نشان می‌دهد.

لیست ۱۹-۲ VHand.java

```
public class VHand extends Hand implements Displayable {

    private HandView view = new HandView();
    public JComponent view() {
```

لیست ۱۹-۲ (ادامه)

```
        return view;
    }

    // you need to override addCard and reset so that when the hand changes, the
    // change propigates to the view
    public void addCard( Card card ) {
        super.addCard( card );
        view.changed();
    }

    public void reset() {
        super.reset();
        view.changed();
    }

    private class HandView extends JPanel {
        public HandView() {
            super( new FlowLayout( FlowLayout.LEFT ) );
            setBackground( new Color( 35, 142, 35 ) );
        }
        public void changed() {
            removeAll();
            Iterator i = getCards();
            while( i.hasNext() ) {
                VCard card = (VCard) i.next();
                add( card.view() );
            }
            revalidate();
        }
    }

}
```

همچون VCard، کلاس VHand نیز به محض بروز هرگونه تغییری از لایه نمایش خود می‌خواهد که خود رابه‌روز کند.

پیاده‌سازی VBettingPlayer

ایده‌ای که در پشت کلاس VBettingPlayer نهفته است همانی است که در کلاسهای VHand و VCard ارایه شده است. لیست ۱۹ ـ ۳ پیاده‌سازی این کلاس را نشان می‌دهد.

لیست ۱۹-۳ VBettingPlayer.java

```
public abstract class VBettingPlayer extends BettingPlayer implements Displayable {
```

لیست ۱۹-۳ (ادامه)

```
private BettingView view;

public VBettingPlayer( String name, VHand hand, Bank bank ) {
    super( name, hand, bank );
}

public JComponent view() {
    if( view == null ) {
        view = new BettingView( (VHand) getHand() );
        addListener( view );
    }
    return view;
}

private class BettingView extends JPanel implements PlayerListener {

    private TitledBorder border;

    public BettingView( VHand hand ) {
        super( new FlowLayout( FlowLayout.LEFT ) );
        buildGUI( hand.view() );
    }

    public void playerChanged( Player player ) {
        String name = VBettingPlayer.this.getName();
        border.setTitle( name );
        repaint();
    }

    public void playerBusted( Player player ) {
        String name = VBettingPlayer.this.getName();
        border.setTitle( name + " BUSTED!" );
        repaint();
    }

  // the rest of the PlayerListener methods have been snipped fot brevity
    // they all follow the same patter, please see source for full listing

    private void buildGUI( JComponent hand ) {
        border = new TitledBorder( VBettingPlayer.this.getName() );
        setBorder( border );
        setBackground( new Color( 35, 142, 35 ) );
        border.setTitleColor( Color.black );
        add( hand );
```

لیست ۱۹-۳ (ادامه)

```
        }

    }

}
```

زمانی که شیء از کلاس VBettingPlayer ایجاد شد، کلاس فوق لایه نمایشی خود را ساخته و از آن به عنوان یک گیرنده (listener) استفاده می‌کند. با تغییر بازیکن، بلافاصله لایه نمایش می‌فهمد که باید خود را به‌روزکند. چیز قابل توجه متد buildGUI() است. این متد موظف به ساخت و برپا کردن نمایش است.

نکته‌ای که باید بدان توجه کرد آن است که به جای آن که هر کارت از دسته کارتها را گرفته و نمایشی از آن را بسازیم، BettingView لایه نمایشی VHand را گرفته و آن را داخل خودش قرار می‌دهد. کلاس VHand نیز نحوه نمایش کارتها را مدیریت می‌کند. تنها کاری که BettingView باید انجام دهد آن است که لایه نمایش را گرفته و آن را به‌روز نگهدارد.

پیاده‌سازی VBlackjackDealer

VBlackjackDealer نیز دقیقاً مثل VBettingPlayer کار می‌کند. لیست ۱۹ ـ ۴ پیاده‌سازی آن را نشان می‌دهد.

لیست ۱۹-٤ VBlackjackDealer.java

```
public class VBlackjackDealer extends BlackjackDealer implements Displayable {

    private DealerView view;

    public VBlackjackDealer( String name, VHand hand, Deckpile cards ) {
        super( name, hand, cards );
    }

    public JComponent view() {
        if( view == null ) {
            view = new DealerView( (VHand) getHand() );
            addListener( view );
        }
        return view;
    }

    private class DealerView extends JPanel implements PlayerListener {

        private TitledBorder border;

        public DealerView( VHand hand ) {
            super( new FlowLayout( FlowLayout.LEFT ) );
            String name = VBlackjackDealer.this.getName();
            border = new TitledBorder( name );
            setBorder( border );
            setBackground( new Color( 35, 142, 35 ) );
```

لیست ۱۹-٤ (ادامه)

```
            border.setTitleColor( Color.black );
            add( hand.view() );
            repaint();
        }

        public void playerChanged( Player p ) {
            String name = VBlackjackDealer.this.getName();
            border.setTitle( name );
            repaint();
        }

        public void playerBusted( Player p ) {
            String name = VBlackjackDealer.this.getName();
            border.setTitle( name + " BUSTED!" );
            repaint();
        }

    // the rest snipped for brevity

    }

}
```

کلاس DealerView تغییرات VBlackjackDealer را رصد می‌کند و به محض تغییر، مدل نمایش خود را به‌روز می‌کند. VHand نیز مراقب است تا نمایش کارتها به‌روز باقی بماند.

پیاده‌سازی GUIPlayer

تمام قسمتهایی از GUIPlayer که نیازی به نمایش ندارند، همان است که در روز گذشته آورده شده‌اند. همچون دیگر کلاسهای لایه انتزاعی، GUIPlayer نیز کلاسی درونی برای نمایش خود فراهم کرده است.

این کلاس، کلاسهای OptionView و OptionViewController را با هم ادغام کرده است. کدهای مربوط به نمایش خیلی تغییر پیدا نکرده‌اند. سرس کد کامل را می‌توانید از سایت www.samspublishing.com بیابید.

ترکیب همه کلاسها با هم به همراه کنترل

قبل از آنکه اقدام به ایجاد کنترل نماییم، باید نسبت به ساخت عامل بازیکن (Player factory) اقدام کنیم. لیست ۱۹ ـ ۵ پیاده‌سازی VPlayerFactory را نشان می‌دهد.

لیست ۱۹-۵ VPlayerFactory.java

```
public class VPlayerFactory {

    private VBlackjackDealer dealer;
    private GUIPlayer human;
    private Deckpile pile;

    public VBlackjackDealer getDealer() {
```

لیست ۱۹-۵ (ادامه)

```
        // only create and return one
        if( dealer == null ) {
            VHand dealer_hand = getHand();
            Deckpile cards = getCards();
            dealer = new VBlackjackDealer( "Dealer", dealer_hand, cards );
        }
        return dealer;
    }

    public GUIPlayer getHuman() {
        // only create and return one
        if( human == null ) {
            VHand human_hand = getHand();
            Bank bank = new Bank( 1000 );
            human = new GUIPlayer( "Human", human_hand, bank, getDealer() );
        }
        return human;
    }

    public Deckpile getCards() {
        // only create and return one
        if( pile == null ) {
            pile = new Deckpile();
            for( int i = 0; i < 4; i ++ ) {
                pile.shuffle();
                Deck deck = new VDeck();
                deck.addToStack( pile );
                pile.shuffle();
            }
        }
        return pile;
    }

    private VHand getHand() {
        return new VHand();
    }

}
```

کلاس VPlayerFactory به ما اطمینان می‌دهد که کلاسهای VBlackjackDealer و GUIPlayer به طرز صحیحی ایجاد شده‌اند. لیست ۱۹ ـ ۶ متد setup() از کلاس BlackjackGUI را نشان می‌دهد. این متد درواقع نمایانگر کنترل است.

لیست ۱۹-۶ setup() از BlackjackGUI

```
    private void setUp() {
        VBlackjackDealer dealer = factory.getDealer();

        GUIPlayer human = factory.getHuman();
```

لیست ۱۹-۶ (ادامه)

```
        dealer.addPlayer( human );

        players.add( dealer.view() );
        players.add( human.view() );
        getContentPane().add( players, BorderLayout.CENTER );
    }
```

متد setup() تک تک بازیکنان را گرفته و لایه نمایش آنها را به خود اضافه می‌کند و سپس واسط را به بازیکنان پیوند می‌دهد. لیست فوق را با لیست ۱۹ ـ ۷ (متد setup() از درس دیروز) مقایسه کنید.

لیست ۱۹-۷ متد setup() از BlackjackGUI به روش MVC

```
    private void setUp() {
        BlackjackDealer dealer = getDealer();
        PlayerView v1 = getPlayerView( dealer );

        GUIPlayer human = getHuman();
        PlayerView v2 = getPlayerView( human );

        PlayerView [] views = { v1, v2 };
        addPlayers( views );

        dealer.addPlayer( human );

        addOptionView( human, dealer );
    }
```

آنگونه که دیده شد درخواست از لایه انتزاعی برای نمایش خود به مراتب ساده‌تر از ایجاد و سپس اتصال نمایشهای مختلف ارایه شده در مدل MVC است.

خلاصه

امروز روش دیگری نسبت به مدل MVC را آموختید. اگر سیستم شما نسبتاً پایدار است و تنها از یک رابط کاربری استفاده می‌کند روش PAC روشی به مراتب بهتر نسبت به روش MVC است.

پرسشها و پاسخها

اگر PAC انتخاب بهتری است چرا باید پیاده‌سازی را نیز فراگیریم؟

PAC تنها یک روش دیگر است و این بدان معنا است که همواره انتخاب بهتر نیست. انتخاب هر روش به تصمیم‌گیری در طراحی برمی‌گردد. یک دلیل فراگیری MVC استفاده از این روش به کرات در صنعت است. کمتر پیش می‌آید که روش PAC مورد استفاده قرار گیرد.

هر دو روش را به خاطر بسپارید. تنها موردی که نباید آن را انجام دهید نوشتن کدهای اصلی (منطق برنامه) در اجزای GUI است. برای مثال کلاس BettingPlayer هیچگاه نباید از JComponent توسعه یابد. هر

دو روش MVC و PAC باعث می‌شوند که منطق برنامه را از GUI آن جدا کنید. الگوهای مختلف روشهای مختلف برای پیاده‌سازی برنامه را ارایه می‌دهند و استفاده از آنها کاملاً بستگی به طراحی شما و تیم طراحی دارد.

کارگاه

سؤالاتی که مطرح می‌شود تنها برای فهم بیشتر شما از مطالب ارایه شده در درس آمده‌اند.

پرسشها

۱. **سه لایه الگوی طراحی PAC کدامند؟**
۲. **توضیح مختصری از هریک از لایه‌های الگوی طراحی PAC ارایه کنید.**
۳. **چگونه می‌توان از وراثت برای جداسازی GUI از باقی کلاسهای سیستم استفاده کرد؟**
٤. **قبل از استفاده از الگوی PAC، سیستم شما باید چه مشخصاتی داشته باشد؟**
۵. **اگر بخواهید از الگوی PAC استفاده کنیم، چگونه می‌توانیم رابط کاربری خط فرمان برای برنامه ایجاد کنیم؟**
۶. **از الگوی Factory برای چه منظوری در این درس استفاده کردیم؟**

تمرین‌ها

۱. سرس کد درس امروز را از اینترنت آورده، آن را کامپایل و سپس اجرا کنید. سپس سعی کنید منطق به کار گرفته شده در آن را خوب بفهمید. فهم کامل برنامه نیازمند وقت و حوصله است
۲. تمرین شماره ۲ از فصل هفدهم از شما خواسته بود تا قابلیت شرط‌بندی دوبرابر (Double Down) را به برنامه اضافه کنید. این کار را برای درس امروز نیز انجام دهید. قابلیت فوق را به سرس کدی که در تمرین ۱ آورده‌اید، اضافه کنید.

روز ۲۰

Object Orinted Programming

کمی تفریح با بازی Blackjack

در روزهای گذشته با یک پروژه OOP سروکله زدید. در درس امروز می‌خواهیم دوباره به بازی برگشته و اندکی با آن تفریح کنیم. این کار را با اضافه کردن چند بازیکن غیرانسانی با استفاده از قابلیت چندشکلی انجام خواهیم داد. خواهید دید چگونه می‌توان از قابلیتهای شیءگرایی (OO) برای پیاده‌سازی شبیه‌سازها (Simulators) استفاده نمود.

آنچه امروز خواهید آموخت:

- استفاده از قابلیت چندشکلی برای اضافه کردن تعدادی بازیکن به بازی Blackjack
- استفاده از OO برای ایجاد شبیه‌سازها

تفریح با قابلیت چندشکلی

بازی Blackjack اجازه می‌دهد تا هفت بازیکن بتوانند در کنار یکدیگر بازی کنند. تاکنون بازی که ساخته‌اید تنها شامل بازیکنان انسانی و یک واسط بوده است. خوشبختانه پلی مورفیسم اجازه می‌دهد که بازیکنان غیرانسانی را نیز به بازی اضافه کنیم.

ایجاد یک بازیکن

برای ساخت یک بازیکن جدید غیرانسانی، کافی است تنها کلاسی بسازیم که

از BettingPlayer مشتق شده‌باشد. این کلاس تنها باید دو متد مجرد زیر را پیاده‌سازی کند:

```
public boolean hit();
public void bet();
```

رفتاری راکه قرار است دو متد فوق پیاده‌سازی کنند، درواقع نحوه بازی کردن را زمانی که دور وی فرا رسیده است، نشان می‌دهد. هیچ نیازی به تغییر وضعیت و یا جایگزینی متد دیگری نیست. وضعیت حال می‌داند چگونه از متدهایی که آنها را پیاده‌سازی کرده‌اید، استفاده کند.

پس از اتمام کلاس جدید لازم است BlackjackGUI را تغییر دهیم تا از بازیکنان جدید در بازی استفاده کند.

کلاس SafePlayer

اجازه دهید بازیکن جدیدی را ایجاد کنیم: SafePlayer. این بازیکن هیچگاه کارت نمی‌کشد و کمترین مقدار ممکن را برای شرط‌بندی می‌گذارد. لیست ۲۰ ـ ۱ تعریف این کلاس را نشان می‌دهد.

لیست ۲۰-۱ SafePlayer.java

```
public class SafePlayer extends BettingPlayer {

    public SafePlayer( String name, Hand hand, Bank bank ) {
        super( name, hand, bank );
    }

    public boolean hit() {
        return false;
    }

    public void bet() {
        getBank().place10Bet();
    }
}
```

توجه داشته باشید که خروجی متد hit() همواره مقدار false است. این کار بدین جهت است که SafePlayer هیچگاه کارت نمی‌کشد. همچنین این بازیکن همواره متد place10Bet() را صدا می‌زند.

اضافه کردن SafePlayer به GUI

اضافه کردن بازیکن جدید به بازی کار ساده‌ای است. ابتدا متد زیر را به BlackjackGUI اضافه کنید:

لیست ۲۰-۲ getSafePlayer()

```
    private Player getSafePlayer() {
        // return as many as called for
        Hand safe_hand = new Hand();
        Bank safe_bank = new Bank( 1000 );
        return new SafePlayer( "Safe", safe_hand, safe_bank );
    }
```

getSafePlayer متد عاملی است که از آن طریق تمام اشیاء از نوع SafePlayer مقداردهی می‌شوند. پس از این متد، کافی است متد setup() را نیز تغییر دهیم تا بازیکنهای جدید به بازی اضافه شوند. لیست ۲۰ ـ ۳ متد setup() تغییر یافته را نشان می‌دهد.

لیست ۲۰-۳ متد setup() تغییر یافته

```
private void setUp() {
    BlackjackDealer dealer = getDealer();
    PlayerView v1 = getPlayerView( dealer );

    GUIPlayer human = getHuman();
    PlayerView v2 = getPlayerView( human );

    Player safe = getSafePlayer();
    PlayerView v3 = getPlayerView( safe );

    PlayerView [] views = { v1, v2, v3 };
    addPlayers( views );

    dealer.addPlayer( human );
    dealer.addPlayer( safe );

    addOptionView( human, dealer );
}
```

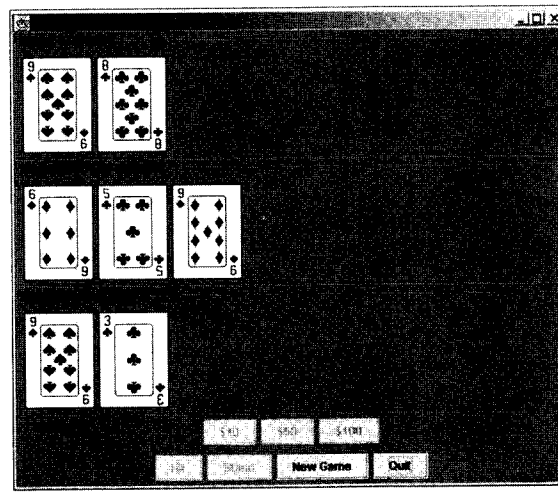
ممکن است بخواهید تا متد اصلی GUI را تغییر دهید تا پنجره‌ای بزرگتر را ایجاد نماید به نحوی که بازیکن جدید نیز در آن نمایش داده شود. شکل ۲۰ ـ ۱ بازیکن جدید را آنگونه که در GUI ظاهر می‌شود، نشان می‌دهد. در اینجا بازیکن به عنوان بازیکن دوم تلقی می‌شود. یعنی بازی خود را پس از بازیکن انسانی انجام می‌دهد. می‌توان تغییری اندک در برنامه داد تا بازیکن غیرانسانی در ابتدا بازی کند.

می‌توان به هر اندازه که بخواهید بازیکنان جدید از نوع BettingPlayer به بازی اضافه کنید. با افزایش تعداد بازیکنان ممکن است بخواهید که به کاربر این اختیار را بدهید تا ترکیب بازیکنان را مشخص کند. برای این کار می‌توانید از یک جعبه گفتگو (Dialog Box) استفاده کنید. می‌توانید متد setup() را به صورت محافظت شده (protected) تعریف کنید تا زیرکلاسها بتوانند این متد را بازنویسی کنند. با تعریف این متد به صورت محافظت شده می‌توان بازیهایی ساخت که ترکیب بازیکنان آنها مختلف است.

OOP و شبیه‌سازی

همانگونه که اشاره شد زمانی که یک سیستم OOP می‌نویسید درواقع یک شبیه‌سازی واقعی از یک مسأله حقیقی را ایجاد می‌کنید. در این هفته سیستمی ساخته‌اید که بازی Blackjack را شبیه‌سازی کرده است.

با ساخت انواع مختلفی از بازیکنان و اضافه کردن آنها به سیستم، بازی ساخته‌اید که بدون دخالت انسانی می‌تواند بازی کند: درواقع یک شبیه‌ساز واقعی از بازی Balckjack. این شبیه‌ساز می‌تواند به دلایل زیر مفید باشد: اول آنکه می‌توانید بازیکنانی بسازید که استراتژیهای مختلفی را در بازی دنبال کنند و ببینید که کدام استراتژی بهتر عمل می‌کند. دوم آنکه ممکن است بخواهید تحقیقی بر روی قابلیتهای هوش مصنوعی



شکل ۲۰ ـ ۱
رابط گرافیکی کاربر
به همراه سه بازیکن

داشته باشید. مثلاً بخواهید از شبکه‌های عصبی (neural networks) برای یادگیری بهینه بازی استفاده کنید. می‌توانید از شبیه‌ساز Blackjack برای این منظور استفاده کنید.

در خلال این درس و تمرین‌های ارایه شده در انتهای درس انواع مختلفی از بازیکنان را خواهید ساخت.

بازیکنان Blackjack

بازیکن SafePlayer را در صفحات قبل مشاهده نمودید. در کنار بازیکن فوق، بازیکنان زیر را خواهیم‌ساخت:

- FlipPlayer: بازیکنی که دائم بین حالت کشیدن و توقف کردن تغییر می‌کند.
- OneHitPlayer: بازیکنی که همواره در هر دور، کارت می‌کشد.
- SmartPlayer: بازیکنی که به دسته‌ای از امتیازاتش که از ۱۱ بیشتر است، تکیه می‌کند.

توجه

موارد کاربردی بازیکنان فوق در درس گذشته ارایه نشده است. به عنوان یک تمرین موارد کاربردی بازیکنان فوق را ایجاد کنید.

پیاده‌سازی FlipPlayer

پیاده‌سازی این بازیکن اندکی از پیاده‌سازی بازیکن SafePlayer پیچیده‌تر است. لیست ۲۰ - ۴ پیاده‌سازی این‌بازیکن را نشان می‌دهد.

لیست ۲۰-٤ FlipPlayer.java

```
public class FlipPlayer extends BettingPlayer {

    private boolean hit = false;
    private boolean should_hit_once = false;

    public FlipPlayer( String name, Hand hand, Bank bank ) {
```

لیست ۲۰-٤ (ادامه)

```
        super( name, hand, bank );
    }

    public boolean hit() {
        if( should_hit_once && !hit ) {
            hit = true;
            return true;
        }
        return false;
    }

    public void reset() {
        super.reset();
        hit = false;
        should_hit_once = !should_hit_once;
    }

    public void bet() {
        getBank().place10Bet();
    }
}
```

کلاس FlipPlayer نیاز به دو متغیر (پرچم یا flag) از نوع boolean دارد. یکی از آنها به بازیکن می‌گوید که در همه دورها نسبت به کشیدن کارت اقدام کند و دیگری مشخص می‌کند که بازیکن در آن دور کارتها را بر زده است یا خیر. از طریق این دو متغیر می‌توان مطمئن بود که در همه دورها بازیکن اقدام به بازی کرده است.

برای آنکه منطق بولی فوق بتواند به کار گرفته شود باید متد reset() را جایگزین کرد تا متغیر should_hit_once تغییر یابد. جایگزینی reset() این اطمینان را می‌دهد که بازیکن در هر بازی و دوری و تنها یک بار اقدام به بازی می‌کند.

پیاده‌سازی OneHitPlayer

پیاده‌سازی OneHitPlayer بسیار شبیه FlipPlayer است. برخلاف FlipPlayer، OneHitPlayer در هر بازی تنها یکبار کارت می‌کشد. لیست ۲۰ - ۵ پیاده‌سازی OneHitPlayer را نشان می‌دهد.

لیست ۲۰-۵ OneHitPlayer.java

```
public class OneHitPlayer extends BettingPlayer {

    private boolean has_hit = false;

    public OneHitPlayer( String name, Hand hand, Bank bank ) {
        super( name, hand, bank );
    }
```

لیست ۲۰-۵ (ادامه)

```
    public boolean hit() {
        if( !has_hit ) {
            has_hit = true;
            return true;
        }
        return false;
    }

    public void reset() {
        super.reset();
        has_hit = false;
    }

    public void bet() {
        getBank().place10Bet();
    }
}
```

دوباره نیاز به جایگزینی متد ()reset است تا از آن طریق متغیر has_hit را بتوان پاک نمود. از طریق این متغیر مطمئن می‌شویم که در هر دور بازیکن تنها یک بار بازی کرده است.

پیاده‌سازی SmartPlayer

کلاس SmartPlayer پیاده‌سازی بسیار ساده‌ای دارد. لیست ۲۰ - ۶ این پیاده‌سازی را نشان می‌دهد.

لیست ۲۰-۶ SmartPlayer.java

```
public class SmartPlayer extends BettingPlayer {

    public SmartPlayer( String name, Hand hand, Bank bank ) {
        super( name, hand, bank );
    }

    public boolean hit() {
        if( getHand().total() > 11 ) {
            return false;
        }
        return true;
    }

    public void bet() {
        getBank().place10Bet();
    }
}
```

کلاس SmartPlayer جمع ارزش کارتها را به دست می‌آورد. اگر این مجموع از ۱۱ بیشتر باشد، متوقف می‌شود و اگر کمتر از یازده باشد، اقدام به بر زدن کارتها می‌کند.

ساخت شبیه‌ساز

برای تبدیل بازی به یک شبیه‌ساز (simulator) خیلی ساده، باید تابع main() در کلاس Blackjack را تغییر داد. در اینجا کلاس فوق را به BlackjackSim تغییر نام می‌دهیم. لیست ۲۰ - ۷ کلاس ذکر شده را نشان می‌دهد.

لیست ۲۰-۷ BlackjackSim.java

```
public class BlackjackSim {

   public static void main( String [] args ) {

      Console.INSTANCE.printMessage( "How many times should the simulator play?" );
      String response = Console.INSTANCE.readInput( "invalid" );
      int loops = Integer.parseInt( response );

      Deckpile cards = new Deckpile();
      for( int i = 0; i < 4; i ++ ) {
         cards.shuffle();
         Deck deck = new Deck();
         deck.addToStack( cards );
         cards.shuffle();
      }

      // create a dealer
      Hand dealer_hand = new Hand();
      BlackjackDealer dealer = new BlackjackDealer( "Dealer", dealer_hand, cards );

      // create a OneHitPlayer
      Bank one_bank = new Bank( 1000 );
      Hand one_hand = new Hand();
      Player oplayer = new OneHitPlayer( "OneHit", one_hand, one_bank );

      // create a SmartPlayer
      Bank smart_bank = new Bank( 1000 );
      Hand smart_hand = new Hand();
      Player smplayer = new SmartPlayer( "Smart", smart_hand, smart_bank );

      // create a SafePlayer
      Bank safe_bank = new Bank( 1000 );
      Hand safe_hand = new Hand();
      Player splayer = new SafePlayer( "Safe", safe_hand, safe_bank );

      // create a FlipPlayer
```

لیست ۲۰-۷ (ادامه)

```
        Bank flip_bank = new Bank( 1000 );
        Hand flip_hand = new Hand();
        Player fplayer = new FlipPlayer( "Flip", flip_hand, flip_bank );

        // hook all of the players together
        dealer.addListener( Console.INSTANCE );
        oplayer.addListener( Console.INSTANCE );
        dealer.addPlayer( oplayer );
        splayer.addListener( Console.INSTANCE );
        dealer.addPlayer( splayer );

        smplayer.addListener( Console.INSTANCE );
        dealer.addPlayer( smplayer );
        fplayer.addListener( Console.INSTANCE );
        dealer.addPlayer( fplayer );

        int counter = 0;
        while( counter < loops ) {
            dealer.newGame();
            counter ++;
        }
    }

}
```

در ابتدا شبیه‌ساز از شما می‌خواهد تا تعداد دفعاتی را که باید بازی کند، مشخص کنید. با دریافت این اطلاعات، اشیاء از نوع BlackjackDealer، OneHitPlayer، SmartPlayer و FlipPlayer ایجاد می‌شوند. سپس این بازیکنان به واسط مرتبط می‌شوند.

پس از طی مراحل فوق، به تعداد دفعات ذکر شده برای بازی، بازی انجام می‌گیرد.

نتایج

پس از اجرای بازی (۱۰۰۰ بار بازی در هر اجرا)، به سادگی می‌توان دید که ترتیب جایگیری بازیکنان به چه صورت است. در اینجا نتایج بازی از بالاترین عنوان به پایینترین عنوان نشان داده شده است.

```
SmartPlayer
SafePlayer
FlipPlayer
OneHitPlayer
```

به سادگی دیده می‌شود که SmartPlayer در همه بازیها بیش از همه پولهای شرط‌بندی را می‌برد. SafePlayer اگرچه رتبه دوم را احراز کرده است، با این حال او نیز مقداری از پولهایش را از دست می‌دهد.

توجه هیچکدام از بازیکنان در انتهای بازی پول بیشتری را از آنچه که در ابتدا برای شرط‌بندی گذاشته‌اند،

دریافت نمی‌کند. تفاوت موجود در رتبه‌بندی بازیکنان تنها بر اساس مقدار پولی است که از دست داده‌اند.

خلاصه

در درس امروز مطالب مهمی را یاد گرفتید. هرگز استراتژیهای ذکر شده در امروز را دنبال نکنید، چرا که ممکن است همه پولهایتان را از دست بدهید!

همچنین ملاحظه کردید که چگونه می‌توان با استفاده از قابلیت چندشکلی نرم‌افزارهای متناسب با نیازهای آینده نوشت. می‌توان بازیکنانی را به سیستم معرفی نمود بدون آنکه هسته سیستم را تغییر داد. این در حالی است که این بازیکنان در مراحل ابتدایی ساخت سیستم، در نظر گرفته نشده‌اند.

پرسشها و پاسخها

آیا دلیل خاصی داشت که از GUI به عنوان اساس شبیه‌ساز، استفاده نکردید؟

می‌توانستم به جای استفاده از خط فرمان از GUI استفاده کنم. اما عموماً شبیه‌سازها رابط کاربری ندارند و تنها اقدام به انتشار یکسری از آمار و ارقام می‌کنند.

البته محدودیتهای عملی نیز در استفاده از GUI نیز وجود دارد. مثلاً آنکه GUI برای تازه کردن اطلاعات نمایش داده شده نیاز به زمان دارد. نمایش گرافیکی ۱۰۰۰ باراجرای بازی به مراتب زمان بیشتری را می‌طلبد. در ضمن بعضی از نسخه‌های swing باعث نشتی حافظه (memory leaks) می‌شود که باعث کندی اجرا و از دست دادن مقادیر زیادی از حافظه در هنگام اجرای بازی می‌شود.

می‌توان تنها به عنوان آزمایش از GUI برای راه‌اندازی شبیه‌ساز استفاده کرد.

کارگاه

پرسشهایی که در این قسمت مطرح می‌شوند تنها برای فهم بیشتر شما از مطالب ارایه شده‌اند.

پرسشها

۱. **قابلیت چندشکلی چگونه باعث می‌شود که بازیکنان غیرانسانی را برای بازی کردن وارد بازی نمود؟**

۲. **چه استراتژی را برای شرط‌بندی شبیه‌سازی نکردیم؟**

تمرین‌ها

۱. سرس کد درس امروز را از سایت www.samspublishing.com آورده و آن را به دقت بررسی کنید. کد به چهار قسمت مجزا تقسیم شده است که در دایرکتوریهای gui، simulation، exercise_2 و exercise_3 قرار دارند.

gui شامل رابط گرافیکی است که یک بازیکن انسانی و بازیکنی از نوع SafePlayerU دارد.

simulation شامل کدهای مربوط به شبیه‌سازی بازی است و بازیکنان OneHitPlayer، FlipPlayer، SafePlayer و SmartPlayer در آن بازی می‌کنند.

exercise_2 و exercise_3 کدهای مربوط به تمرین‌های ۲ و ۳ را در بر دارند.

کدهای مربوط به gui و simulation را مطالعه کرده و سعی کنید منطق به کار رفته در آنها را متوجه شوید. سپس به سراغ تمرین‌های ۲ و ۳ بروید.

۲. تغییراتی در متد hit() تعریف شده در کلاس Player انجام گرفته است. این تغییر Dealer را به عنوان پارامتر ورودی معرفی کرده است و متد جدیدی (getUpCard) به کلاس Dealer اضافه شده است. در بازی واقعی Blackjack بازیکنان می‌توانند کارتی از واسط را که رو به بالا است، ببینند. ازاین اطلاعات می‌توان برای انجام حرکات هوشمندانه‌تر استفاده کرد. برای این تمرین یک یا دو بازیکن جدید ایجاد کنید که مبنای بر زدن کارتها بر اساس کارت واسط باشد. دو پیشنهاد برای این منظور ارایه شده است: KnowledgeablePlayer و OptimalPlayer.

۳. سرس کد ارایه شده برای این تمرین شامل مراحلی است که برای دو برابر کردن (دوبل کردن) میزان شرط‌بندی لازم است. برای مثال متد جدید doubledown کلاس dealer را به عنوان آرگومان ورودی دریافت می‌کند. متد getUpCard نیز به کلاس dealer اضافه شده است تا از آن طریق بازیکنان بتوانند کارت واسط را که رو به بالاست ببینند. برای این تمرین یک یا دو بازیکن جدید ایجاد کنید، به نحوی که بتوان مبنای تصمیم‌گیری آنها برای بر زدن کارتها و یا دوبل کردن پول شرط‌بندی را بر اساس کل موجودی پولها و کارت واسط قرار داد. پس از پیاده‌سازی این بازیکنان آنها را به شبیه‌ساز اضافه کرده و نتایج را ملاحظه کنید. برای این منظور می‌توانید از دو پیشنهاد ارایه شده در سرس کد استفاده کنید. این دو پیشنهاد عبارتند از: KnowledgeablePlayer و OptionPlayer.

روز ۲۱

آخرین قدم

تبریک! بالاخره به آخرین درس از این کتاب رسیدید. راه درازی پیمودید. دیگر پایه و اساس آنچه که برای فراگیری برنامه‌نویسی شیءگرا لازم است دارید.

آنچه امروز خواهید آموخت:

- بازسازی بازی Blackjack برای استفاده مجدد در دیگر سیستمها
- مزایایی که OOP برای بازی Blackjack به ارمغان آورده است.
- حقایق موجود در صنعت که نمی‌توان از راه حلهای شیءگرایی استفاده کرد.

نهایی کردن

در خلال سه هفته گذشته مطالب فراوانی را آموختید. با مبانی شیءگرایی شروع کرده و تا جایی پیش رفته‌اید که پروژه‌های خود را بر اساس OOP انجام می‌دهید. دیگر باید به خوبی درک کنید OOP چه چیزی را بازگو می‌کند.

قبل از آنکه کتاب را به اتمام برسانید، سه مبحث را باید بدانید:

- بازسازی طراحی بازی Blackjack برای استفاده مجدد دیگر سیستمها
- مزایای OOP برای Blackjack
- حقایق مربوط به صنعت نرم‌افزار و OOP

بازسازی طراحی بازی Blackjack برای استفاده مجدد در دیگر سیستمها

نکته کوچکی در طراحی بازی Blackjack باقی مانده است که اکنون فرصت خوبی است تا به آن بپردازیم. این نکته بازی Blackjack را تحت تأثیر قرار نخواهد داد ولی در صورتی که از طراحی آن به صورت نادرستی استفاده شود می‌تواند بر روی دیگر سیستمهای شیءگرا تأثیر منفی داشته باشد.

در روز ۱۵ دیدید که دو راه حل مختلف وجود دارد. در راه حل اول واسط پس از اتمام بازی هر بازیکن سراغ بازیکن بعدی رفته و از وی می‌خواهد که بازی را ادامه دهد و این کار را حلقه‌وار انجام می‌دهد. پس از آن روش شیءگرای بهتری ارایه شد.

به جای آنکه واسط سراغ هر بازیکن رفته و از وی درخواست کند که بازی را ادامه دهد، واسط شیءگرا با یک بازیکن شروع کرده و صبر می‌کند تا همان بازیکن وی را از اتمام بازی خبردار کند.

این کار روشی روشنتر نسبت به راه حل قبلی ارایه می‌کند چراکه باید صبر کنید تا بازیکن به واسط بگوید که کارش تمام شده است، تنها در این صورت است که واسط به بازی ادامه می‌دهد. همچنین مشخص می‌شود که تنها این نوع طراحی است که باعث می‌شود GUI به درستی کار کند.

مشکل طراحی

تنها یک مشکل کوچک با این نوع طراحی وجود دارد. اجازه دهید ترتیب فراخوانی متدها را ردیابی (Trace) کنیم (فرض کنید تنها یک بازیکن و واسط وجود دارد)

لیست ۲۱ ـ ۱ ردیابی فراخوانی متدها در بازی را نشان می‌دهد. با بازگشت متد (اتمام متد) پشته (Stack) متوقف می‌شود.

لیست ۲۱-۱ ردیابی فراخوانی متدها در بازی Blackjack

```
BlackjackSim.main
 BlackjackDealer.newGame
  Player.play
   BlackjackDealer$DealerCollec ingBets.execute
    Player.play
     BettingPlayer$Betting.execute
      BlackjackDealer.doneBettin g
       Player.play
        BlackjackDealer$DealerCollectingBets.execute
         BlackjackDealer$DealerDealing.execute
          BlackjackDealer$DealerWaiting.execute
           Player.play
            Player$Playing.execute
             Player$Standing.execute
              BlackjackDealer.standing
               Player.play
                BlackjackDealer$DealerWaiting.execute
                 Player$Playing.execute
                  BlackjackDealer$DealerStanding
```

مشکل در اینجاست: هیچکدام از متدها تا زمانی که دور واسط به اتمام نرسد، باز نمی‌گردند (به اتمام

نمی‌رسند). درواقع متدها به صورت بازگشتی همدیگر را فراخوانی می‌کنند. برای مثال متد notifyChanged در لیست ۲۱ ـ ۲ تا زمانی که بازی جاری به اتمام نرسد اجرا نمی‌گردد.

لیست ۲۱-۲ متدی که تا زمانی که بازی جاری به اتمام نرسد فراخوانی نمی‌شود

```
public void execute( Dealer dealer ) {
    if( hit( dealer ) ) {
        dealer.hit( Player.this );
    } else {
        setCurrentState( getStandingState() );
        notifyStanding();
    }
    current_state.execute( dealer );
    // transition

    // will not get called until stack unwinds!!!!
    notifyChanged();
}
```

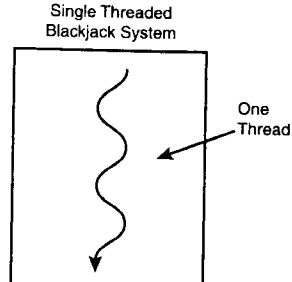
در بازی Blackjack این امر مشکل حادی نیست. به دلیل آن که تعداد محدودی بازیکن (حداکثر هفت نفر) وجود دارد و فراخوانی متدها کم است. بنابراین باید در هنگام نوشتن کدهایی شبیه آنچه که در لیست ۲۱-۲ آمده است، دقت به خرج دهید.

حال فرض کنید شبیه‌سازیی (simulator) حاوی صدها بلکه هزاران شیء را که از طراحی بازی Blackjack پیروی می‌کنند، داریم. اگر این اشیاء به طور بازگشتی (recursively) یکدیگر را فراخوانی کنند، به سرعت با مشکل کمبود حافظه مواجه خواهید شد. چرا که با فراخوانی هر متد، قسمتهایی از حافظه اشغال می‌شود. بنابراین اگر بر طبق این طراحی پیش بروید سیستم شما هیچگاه به طور درست اجرا نخواهد شد و یا آنکه نیازمند مقدار زیادی حافظه خواهد بود.

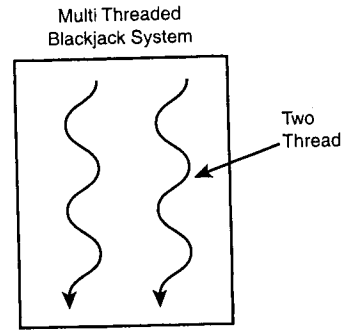
یک راه حل جدید: رشته‌ها (Threads)

خوشبختانه یک راه حل وجود دارد. رشته‌ها (Threads). اگر چه بحث مفصل در مورد Thread از حوصله این کتاب خارج است، با این حال خواهید دید چگونه می‌توان با استفاده از این روش مشکل فراخوانی متدها را سریعاً رفع کرد.

یک رشته، یک مسیر اجرای برنامه است. بنابراین سیستم Blackjack یک مسیر اجرا دارد. شکل ۲۱ ـ ۱ کمک می‌کند مفهوم تک‌رشته (Single Thread) را تصور کنید.



شکل ۲۱ ـ ۱
سیستم Blackjack تک رشته‌ای



شکل ۲۱ ـ ۲
سیستم Blackjack شامل
چند رشته اجرایی

از آنجایی که سیستم Blackjack تنها یک رشته اجرایی دارد، این تک رشته موظف است همه کارها را انجام دهد.

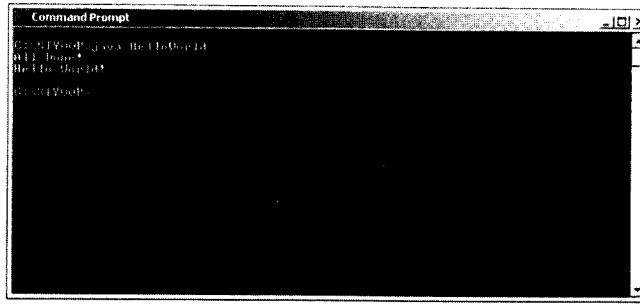
می‌توان از قابلیت چندرشته‌ای (Multi Threading) استفاده کرد و چندین مسیر اجرایی برای برنامه ساخت. با ساخت چند رشته اجرایی، برنامه می‌تواند کارهای مختلفی را در زمان واحد انجام دهد. شکل ۲۱ـ۲ کمک می‌کند تا مفهوم سیستم Blackjack که از طریق دو رشته اجرایی، اجرا می‌شود، را درک کنید.

از طریق این روش سیستم Blackjack می‌تواند به متدها اجازه دهد که به طور صحیحی پایان بپذیرند. برای مثال برنامه ساده زیر را که پیغام Hello world! را چاپ می‌کند در نظر بگیرید.

لیست ۲۱-۳ برنامه Hello world! همراه با قابلیت چندرشته‌ای

```
public class HelloWorld {

public void sayHello() {
        System.out.println( "Hello World!" );
    }

public static void main( String [] args ) {
       final HelloWorld hw = new HelloWorld();

       Runnable runnable = new Runnable() {
          public void run() {
             hw.sayHello();
               }
        };

       Thread thread = new Thread( runnable );
       thread.start();

       System.out.println( "All Done!" );

   }

}
```

HelloWorld به تنهایی کلاس ساده‌ای است که تنها یک متد دارد: sayHello. متد sayHello تنها پیغامی را در خط فرمان چاپ می‌کند.



شکل ۲۱ ـ ۳
خروجی برنامه Helloworld

()main جایی است که مورد علاقه است. در ابتدا نمونه‌ای از کلاس HelloWorld در ()main ایجاد می‌شود. سپس کلاس ناشناس Runable در این متد تعریف می‌شود. این کلاس نیز تنها یک متد دارد: run. متد runاست که به رشته اجرایی می‌گوید چه کاری انجام دهد. در این مورد متد run به نمونه ایجاد شده از کلاس HelloWorld می‌گوید که پیغامش را چاپ کند.

پس از ساخت کلاس Runnable، متد main یک رشته اجرایی می‌سازد. زمانی که یک رشته می‌سازید، باید Runnable را به عنوان آرگومان به آن پاس کنید. متد run از کلاس Runnable به رشته می‌گوید چه چیزی را اجرا کند. پس از اجرای رشته متد ()main پیغام خودش را چاپ می‌کند.

ممکن است از نتیجه به دست آمده شگفت‌زده شوید. شکل ۲۱ ـ ۳ خروجی برنامه HelloWorld را نشان می‌دهد. با اجرای برنامه خواهید دید که ابتدا پیغام "All Done!" ظاهر می‌شود و سپس عبارت "Hello World!" چاپ می‌گردد. درواقع فراخوانی Thread.start اجرای باقی برنامه را دچار توقف (block) نمی‌کند. چرا که متد start، رشته اجرایی جدیدی را می‌سازد و بلافاصله به متن اصلی برنامه برمی‌گردد. پس از فراخوانی start برنامه شما دارای دو رشته در حال اجراست. مواقع بسیاری پیش می‌آید که main پیغام خود را سریعتر از آنکه رشته وضعیت فراخوانی sayHello را داشته باشد، چاپ می‌کند.

با توجه به نکات فوق می‌توان از قابلیت چندرشته‌ای در بازی Blackjack استفاده کرد چرا که صدا زدن متد start اجرای باقی قسمتهای برنامه را متوقف نمی‌کند. لیست ۲۱ ـ ۴ حالت انتظار (waiting state) جدیدی را برای BlackjackDealer در بر دارد که بازی هر بازیکن در رشته مختص به خود بازیکن صورت می‌پذیرد.

لیست ۲۱-٤ استفاده از قابلیت چندرشته‌ای در بازی Blackjack

```
private class DealerWaiting implements PlayerState {
    public void handChanged() {
        // not possible in waiting state
    }
    public void handPlayable() {
        // not possible in waiting state
    }
    public void handBlackjack() {
        // not possible in waiting state
    }
```

لیست ۲۱-٤ (ادامه)

```
    public void handBusted() {
        // not possible in waiting state
    }
    public void execute( final Dealer dealer ) {
        if( !waiting_players.isEmpty() ) {
            final Player player = (Player) waiting_players.get( 0 );
            waiting_players.remove( player );
  Runnable runnable = new Runnable() {
              public void run() {
                    player.play( dealer );
              }
            };
            Thread thread = new Thread( runnable );
            thread.start();
        } else {
            setCurrentState( getPlayingState() );
            exposeHand();
            getCurrentState().execute( dealer );
            // transition and execute
        }
    }
}
```

با شروع بازی هر بازیکن در رشته مخصوص خودش، متد execute بلافاصله ادامه می‌یابد (درواقع اجرای آن متوقف نمی‌شود) این امر باعث حل مشکلاتی می‌شود که در فراخوانی متدهای بازگشتی عمدتاً به وجود می‌آید.

نکته

استفاده از قابلیت چندرشته‌ای تنها یک راه حل برای مشکل بازگشتی است. در اینجا خواستم با ارایه راه حلی مبتنی بر این قابلیت یک دید کلی برای استفاده از چندرشته‌ای به شما منتقل کنم.

کد منبع مربوط به نسخه چندرشته‌ای از GUI بازی Blackjack را می‌توانید از سایت www.samspublishing.com دریافت کنید.

مزایایی که OOP برای بازی Blackjack به ارمغان آورده است

در هفته اول برخی از اهداف و مزایای OOP را بیان کردیم. برای یادآوری مجدد، گفتیم که OOP سعی دارد تا نرم‌افزارهایی را ارایه کند که:

۱. طبیعی
۲. پایدار و قابل اطمینان
۳. قابل استفاده مجدد
٤. قابل نگهداری
۵. قابل توسعه
۶. متناسب با زمان حال

باشند. OOP تمام موارد فوق را در سیستم بازی Blackjack پیاده‌سازی کرده است:

- طبیعی: سیستم ارایه شده برای بازی Blackjack به طور طبیعی این بازی را مدل کرده است.
- پایدار و قابل اطمینان: سیستم Blackjack از پایداری و قابلیت اطمینان خوبی برخوردار است. این امر را از طریق تستهای مختلف و همچنین به خاطر کپسوله‌سازی خوب می‌توان فهمید. از آنجا که دانش (Knowledge) و مسئولیت‌پذیری (Responsiblity) از یکدیگر جدا (ایزوله) شده‌اند، می‌توان بدون آنکه بر روی سیستم تأثیر منفی گذاشت، قابلیتهای آن را بهبود بخشید.
- قابل استفاده مجدد: سیستم Blackjack دارای قابلیت استفاده مجدد است. از کلاسهای تعریف شده برای کارتها، دسته کارتها و... (Card، Deck،...) می‌توان در دیگر بازیهای کارتی به خوبی استفاده کرد.
- قابل نگهداری: سیستم Blackjack دارای قابلیت نگهداری بسیار خوبی است. از آنجا که دانش (داده‌ها) و مسئولیت‌پذیری در اشیاء ساخته شده در بازی از یکدیگر جدا شده‌اند، می‌توان به راحتی از سیستم نگهداری کرد و با خواست کاربر قابلیتهای آن را بهبود بخشید.
- قابل توسعه: سیستم Blackjack به راحتی قابل توسعه است. با استفاده از وراثت می‌توان انواع جدیدی از کارتها، دسته کارتها، بازیکنان و غیره تعریف کرد و سیستم را توسعه داد.
- متناسب با زمان حال: سیستم Blackjack سیستمی است که متناسب با زمان حال ساخته شده است.

حقایق مربوط به صفت نرم‌افزار و OOP

دروس ارایه شده در این کتاب فرض کرده‌اند که شما پروژه‌های OOP خود را از ابتدا شروع کرده‌اید. زمانی که پروژه از ابتدا شروع می‌شود نیازی به اتصال دیگر سیستمهای غیرشیءگرا (non – OO) نیست. دیگر نیازی به استفاده از کتابخانه‌های رویه‌ای نیست. حال آنکه پروژه‌های OOP مستقل، به ندرت یافت می‌شوند. اکثر اوقات نیاز است که با اجزای غیرشیءگرا در تعامل باشید. برای مثال بانکهای اطلاعاتی رابطه‌ای را در نظر بگیرید. بانکهای رابطه‌ای به طور مؤثر شیءگرا نیستند و پایگاههای داده شیءگرا همچنان به ندرت مورد استفاده قرار می‌گیرند.

Java نیز به خودی خود زبان شیءگرای کاملی نیست و این باعث می‌شود که گاهی اوقات از کدنویسی غیرشیءگرا استفاده کنید. در این گونه مواقع بهتر است که اجزای غیرشیءگرا را تبدیل به اجزای شیءگرا کنید. برای مثال زمانی که با پایگاههای داده‌ای رابطه‌ای سروکار دارید، به جای آنکه مستقیماً سراغ پایگاه داده‌ای رفته و یک پرس‌وجو (Query) را اجرا کنید، بهتر است برای این منظور کلاسی نوشته و از آن بخواهید که کارها را برایتان انجام دهد.

البته نمی‌توان تمام اجزاهای غیرشیءگرا را تبدیل به اجزای شیءگرا کرد. درواقع زمان بسیار زیادی می‌طلبد تا یک سیستم غیرشیءگرا را تبدیل به نوع شیءگرا (البته اگر بشود این کار را کرد) نمود. بنابراین خودتان را برای این گونه معضلات آماده کنید.

خلاصه

تمام شد! در سه هفته کوتاه این کتاب پایه محکمی از OOP را به شما آموخت. باقی همه بستگی به شما دارد. تمام دانش لازم را برای اعمال اصول و قواعد OOP به پروژه‌های خود را به دست آورده‌اید. موفق باشید!

پرسشها و پاسخها

چرا برای گفتن قابلیت چندرشته‌ای تا این درس ما را منتظر نگهداشتید؟

در صورت ارایه این قابلیت قبل از موعد آن ممکن بود شما را دچار دردسر کند. باید به مشکل گفته شده در این رابطه برخورد می‌کردید تا اهمیت چندرشته‌ای و نحوه استفاده از آن را به خوبی فرا می‌گرفتید.

چندرشته‌ای خود مبحث بسیار پیشرفته‌ای است. اگرچه در پیاده‌سازی بازی Blackjack از یک جنبه ساده آن استفاده شده است، در دیگر برنامه‌ها بسیار پیچیده می‌باشد.

چه چیزی استفاده از قابلیت چندرشته‌ای را مشکل می‌کند؟

در صورتیکه چندرشته اجرایی از داده‌های مشترک استفاده می‌کنند، یکی رشته می‌تواند داده‌ها را تغییر داده و به طور اتفاقی باعث می‌شود رشته دیگر (که از داده‌ها اطلاعاتی را می‌خواند) دچار مشکل شود. در اینگونه مواقع باید رشته‌ها را با یکدیگر همزمان (Synchron) نمود. این مبحث خود فصل جداگانه‌ای را می‌طلبد.

کارگاه

سؤالات مطرح شده در این قسمت برای فهم بیشتر شما از مطالب ارایه شده آورده شده‌اند.

پرسشها

۱. **چگونه قابلیت چندرشته‌ای مشکل فراخوانی متدهای بازگشتی را حل می‌کند؟**

تمرین‌ها

۱. سرس کد مربوط به درس امروز را از سایت انتشارات Sams بیاورید. سرس کد به چهار دایرکتوری تقسیم شده است:

threaded_hello_world، threaded_mvc_gui، threaded_pac_gui و threaded_simulator. هریک از چهار کد فوق را به دقت مطالعه کنید.

۲. مطالعه شما در مورد OOP با این کتاب پایان نمی‌پذیرد. فهرستی از مباحثی را که باید در مورد آنها چیزهای بیشتری را فراگیرید نوشته و مطالب را بر اساس اهمیت آنها رتبه‌بندی کنید. سپس در وب به دنبال آنها گشته و مطالعه را آغاز کنید!

پاسخ به تمرین‌ها

روز ۱

پاسخ پرسش‌ها

۱. به عنوان یک قاعدهٔ نرم‌افزاری، برنامه‌نویسی رویه‌ای یک برنامه را به داده‌ها و رویه‌هایی تجزیه می‌کند طوری که بتوان داده‌های آن را دستکاری کرد و در آنها تغییر به وجود آورد. برنامه‌نویسی رویه‌ای دارای طبیعت سلسله مراتبی است. زمانی که لیستی از رویه‌ها صدا زده می‌شوند پشت سر هم اجرا می‌شوند و این باعث می‌شود که جریان برنامه‌نویسی رویه‌ای به کار افتد.

۲. برنامه‌نویسی رویه‌ای به برنامه یک ساختار کلی شامل داده و رویه می‌دهد. رویه‌ها در سازماندهی کار به کار می‌آیند. به جای نوشتن حجم وسیعی از یک بلوک می‌توان این رویه را به چند ریز رویه شکست و کار را آسانتر کرد. همچنین این کار قابلیت استفادهٔ مجدد از رویه را ایجاد می‌کند و می‌توان کتابخانه‌ای از رویه‌های دارای این قابلیت ایجاد کرد.

۳. برنامه‌نویسی ماژولار (پیمانه‌ای) میان داده‌ها و رویه‌ها پیوند محکمی برقرار می‌کند تا بتوان داده را به واحدهایی که به عنوان ماژول شناخته می‌شوند تبدیل کرد. ماژولها کارهای داخلی یک برنامه و

نمایش داده را مخفی می‌کنند. با وجود این بیشتر زبانهای برنامه‌نویسی ماژولی این امکان را فراهم می‌آورند که از ماژولهای آنها در محیط رویه‌ای استفاده شود.

٤. برنامه‌نویسی ماژولار پیاده‌سازی برنامه را مخفی می‌کند و به این ترتیب از دستکاری نامناسب داده‌ها جلوگیری می‌شود. ماژولها ساختار سطح بالاتری را به برنامه تحمیل می‌کنند و اجازه می‌دهند به جای فکر کردن به داده‌ها و رویه‌ها، بتوانید به سطح رفتاری و مفهومی برنامه فکر کنید.

۵. هم برنامه‌نویسی ماژولی و هم برنامه‌نویسی رویه‌ای قابلیت استفادهٔ مجدد را محدود کرده‌اند. اگر چه می‌توان از رویه‌ها دوباره استفاده کرد. ولی رویه‌ها به شدت به داده هایشان وابسته هستند. سراسری بودن داده‌ها در برنامه‌نویسی رویه‌ای قابلیت استفادهٔ مجدد را مشکل می‌کند. رویه‌ها وابستگی‌هایی دارند که تعیین آنها مشکل است ماژولها به آسانی و سهولت قابل استفادهٔ مجدد هستند. استفادهٔ از یک ماژول در هر برنامه‌ای امکانپذیر است. اما ماژولها در این زمینه دارای محدودیتی هستند و این محدودیت به این ترتیب است که از آنها می‌توان فقط به صورت مستقیم در برنامه‌ها استفاده کرد. یعنی نمی‌توان از یک ماژول به عنوان پایه برای ایجاد یک ماژول جدید استفاده کرد.

۶. برنامه‌نویسی شیءگرا (OOP) یک روش نرم‌افزاری است که برنامه را برحسب اشیای دنیای حقیقی مدل می‌کند. برنامه‌نویسی شیءگرا برنامه را به تعدادی شیء که با یکدیگر رابطهٔ متقابل دارند می‌شکند. این شکل از برنامه‌نویسی بر اساس برنامه‌نویسی ماژولار ساخته شده است و با پشتیبانی از کپسوله‌سازی همچنین به کارگیری قابلیت استفادهٔ مجدد (Reuse) از طریق وراثت تکمیل شده است.

۷. شش فایدهٔ برنامه‌نویسی شیءگرا عبارت است از:

- برنامه‌های طبیعی
- قابل اعتماد
- قابل استفادهٔ مجدد
- قابل نگهداری
- قابل توسعه
- قابل حصول در زمان قابل قبول

۸. برنامه‌نویسی شیءگرا یک روش برنامه‌نویسی طبیعی است. که در آن به جای مدل کردن مسایل بر حسب داده یا رویه، برنامه‌ها بر حسب مسایل و مشکلاتی که وجود دارند مدل می‌شوند. چنین روشی برنامه‌نویس را در فکر کردن بر حسب مسایل و تمرکز روی آنچه که سعی در انجام آن دارد یاری می‌کند. در این روش نیازی نیست که روی جزییات پیاده‌سازی تمرکز شود.

۹. کلاس، همهٔ رفتارها و نسبت‌های مشترک میان یک گروه از اشیاء را تعریف می‌کند. از تعریف کلاس می‌توان برای ایجاد اشیایی از این گروه استفاده کرد. شیء نمونه‌ای از یک کلاس است. برنامه‌ها به دستکاری این اشیاء می‌پردازند. ایفا کنندهٔ نقش‌های اصلی در واقع اشیاء هستند. فراخوانی کارکردهای یک شیء از طریق رابط عمومی آن انجام می‌شود. دیگر اشیاء ممکن است با رابط شیء هر رفتاری را در پیش گیرند.

۱۰. اشیاء با فرستادن پیغام به یکدیگر با هم رابطه برقرار می‌کنند. فراخوانی یک پیغام مترادف است با اینکه رویه یا متدی برای فراخوانی ساخته شود.

۱۱. تابع سازنده برای ایجاد یک نمونهٔ شیء به کار می‌رود. با استفاده از تابع سازندهٔ شیء مقداردهی اولیه می‌شود و برای استفاده در برنامه آماده می‌شود.
۱۲. یک دست‌یابنده دسترسی به داده‌های داخلی یک شیء را ممکن می‌سازد.
۱۳. یک تغییر دهنده متدی است که حالت داخلی شیء را تغییر می‌دهد.
۱٤. هر نمونه شیء می‌تواند رجوعی به خودش داشته باشد. که این رجوع دسترسی به متغیرها و رفتارهای داخلی آن شیء را ممکن می‌سازد.

روز ۲

پاسخ پرسش‌ها

۱. کپسوله سازی فرایندی طبیعی است که اجازه می‌دهد نرم‌افزار را بر حسب مساله مدل کنید نه بر حسب جزییات و پیاده‌سازی برنامه.
کپسوله سازی قابلیت اعتماد به نرم‌افزار را بالا می‌برد. کپسوله سازی فرآیندهای داخلی یک تکه از نرم‌افزارها را مخفی می‌کند و دسترسی مناسب به آن را تضمین می‌نماید. کپسوله سازی باعث می‌شود که وظایف از هم جدا شده و ارزیابی شوند. زمانی که یک تکه از نرم‌افزار به درستی کار می‌کند با اطمینان می‌توان از آن دوباره استفاده کرد.
کپسوله سازی قابلیت استفادهٔ مجدد را برای نرم‌افزارها ایجاد می‌کند. از آنجایی که هر قسمت نرم‌افزار به صورت مستقل عمل می‌کند، می‌توان از آن در جاهای مختلفی مجدداً استفاده کرد.
کپسوله سازی باعث می‌شود که نرم‌افزار قابل نگهداری شود چرا که هر قسمت از بخشهای دیگر مستقل است. ایجاد تغییر در یک قسمت صدمه‌ای به قسمت‌های دیگر نمی‌زند. بنابراین نگهداری و توسعهٔ برنامه به آسانی قابل انجام است.
کپسوله سازی باعث ماژولار (پیمانه‌ای) شدن نرم‌افزار می‌شود. ایجاد تغییر در یک قسمت از برنامه صدمه‌ای به کد نوشته شده در قسمت دیگر وارد نمی‌کند. پیمانه‌ای بودن برنامه باعث می‌شود که اشکال موجود در یک قسمت به خوبی رفع شود بدون آنکه به بقیهٔ کد لطمه‌ای وارد شود.
کپسوله سازی باعث می‌شود کد شما به اندازه و مفید باشد چرا که قسمت‌های غیر ضروری از برنامه حذف می‌شوند. اغلب اوقات وابستگیکهای مخفی که در کد وجود دارند باعث پدید آمدن اشکالاتی در برنامه می‌شود که رفع کردن آنها بسیار سخت است.
۲. تجرید فرایندی است که به ساده‌سازی مسایل سخت می‌پردازد. وقتی می‌خواهید مسأله‌ای را حل کنید نیازی نیست که خودتان را با جزییات مسأله زیاد درگیر کنید. به جای آن می‌توانید با آدرس‌هایی جزییات حل مسایل را فرموله کنید و مسأله را به آسانی حل کنید.
Desktop گرافیکی کامپیوتر مثال ساده‌ای از تجرید است که جزییات فایل سیستم را به طور کامل از دید شما مخفی نگاه می‌دارد.

۳. پیاده‌سازی تعریف می‌کند که چگونه یک قسمت از کد سرویسی را فراهم می‌کند. در پیاده‌سازی جزییات داخلی هستند که مورد توجه قرار می‌گیرد.

٤. رابط معین می‌کند که با یک تکه از برنامه چه کاری می‌توان انجام داد. رابط پیاده‌سازیهای سطوح زیرین برنامه را کاملاً مخفی می‌کند.

۵. رابط معین می‌کند که یک قسمت از برنامه چه کاری انجام می‌دهد، پیاده‌سازی بیان می‌کند که چگونه این کار انجام می‌شود.

۶. بدون تقسیم بندی واضح، مسؤولیتها در هم می‌شوند و این مسأله منجر به پدید آمدن دو مشکل می‌شود. اول اینکه، کدی که باید متمرکز باشد، غیر متمرکز می‌شود. وظایف و مسؤولیتهایی که غیر متمرکز شده باید دوباره تکرار شود یا دوباره پیاده‌سازی شود. البته در هر جایی که مورد نیاز باشد. به عقب برگردید به مثال کلاس BadItem که قبلاً بیان شد.

به آسانی می‌توان دید که هر کاربر نیاز دارد که کد را برای محاسبهٔ جمع کلی یک آیتم دوباره پیاده‌سازی کند. هر وقت که منطق برنامه دوباره نوشته شود باعث می‌شود که با خطاهای بیشتری مواجه شوید. این کار باعث می‌شود که استفادهٔ نامناسبی از کد شود چرا که وظیفهٔ بین چند قسمت پخش می‌شود.

۷. یک نوع داده، یک عنصر زبان است که تعدادی واحد محاسبه و رفتار را پیاده می‌کند. اگر خطوط کد در حکم جملات باشند، انواع داده‌های کلمات آن هستند. انواع داده‌ای اغلب به صورت مستقل عمل می‌کنند، خودبسنده هستند و به عنوان واحدهای کوچک شناخته می‌شوند.

۸. ADT مجموعه‌ای از داده و مجموعه‌ای از عملیاتی است که روی آن داده انجام می‌گیرد. ADTها ما را قادر می‌سازند که انواع جدیدی را با مخفی کردن داده‌های داخلی و قرار گرفتن در پشت یک رابط خوب تعریف شده بسازیم.

۹. روشهایی که برای مخفی کردن پیاده‌سازی مسأله و کد وابسته به آن وجود دارد. روش آسان، استفاده از کپسوله سازی است. با وجود آنکه روش، روش آسانی است اما دستیابی به کپسوله سازی کار آمد و مفید یا تصادفی نیست.

در اینجا چند نکته برای دستیابی به این کپسوله سازی مطرح می‌شود.

- به ADT فقط از طریق یک رابط دسترسی داشته باشید و اجازه ندهید که ساختارهای داخلی بخشی از رابط عمومی شوند.
- دسترسی به ساختارهای داخلی داده را فراهم نکنید و همهٔ دسترسی‌ها را مجرد کنید.
- دسترسی غیر توجیهی به ساختارهای داخلی داده از طریق اشاره گرهای بازگشتی تصادفی فراهم نکنید.
- فرضهایی در رابطه با انواع داده‌ای دیگری که استفاده می‌کنید نداشته باشید تا زمانی که رفتار یک نوع داده‌ای در رابط کلاس یا در مستندات آن ظاهر نشده به آن نوع داده‌ای اعتماد نکنید.
- هنگامی که دو نوع داده‌ای بسیار نزدیک و وابسته به یکدیگر می‌نویسید دقت کنید. به خودتان اجازه ندهید که فرضها و وابستگی هایی را به صورت تصادفی ایجاد کنید.

۱۰. باید از چند دام حذر کرد:

در این کار دچار پریشانی نشوید. مسایلی که با آنها روبرو می‌شوید را در ابتدا حل کنید. حل کردن این مسایل اولین کار شما می‌باشد. به تجرید به عنوان یک امتیاز مثبت، نگاه کنید نه به عنوان هدف نهایی. در

غیر اینصورت امکان دارد زمان را از دست بدهید و تجرید شما ناصحیح باشد. اما اگر طبق مراحل بالا پیش روید زمان کافی در اختیار دارید حتی اگر به تجرید مناسبی دست پیدا نکردید، زمان برای جبران در اختیار دارید.

تجرید می‌تواند خطرناک باشد. حتی اگر تعدادی از عناصر را مجرد کرده باشید ممکن است که در هر جایی کار نکنند. نوشتن کلاسی که نیازهای هر کاربر را پاسخگو باشد بسیار سخت است.

در هر کلاس بیشتر از آنچه که برای حل مسأله نیاز است، مسؤولیت قرار ندهید و به فکر حل کردن همهٔ مسایل نباشید. ابتدا مسأله را حل کنید و سپس در جستجوی راهی برای تجرید آنچه که انجام داده‌اید باشید.

حل تمرین‌ها

۱. یک پشتهٔ ممکن ADT:

```
public interface Stack {
    public void push( Object obj );
    public Object pop();
    public boolean isEmpty();
    public Object peek();
}
```

۲. برای یک پشته (Stack)، بهترین پیاده‌سازی به صورت یک لیست منفرد از عناصر است که می‌تواند با اشاره‌گر به عنصر ابتدایی پیوند برقرار کند. وقتی که یک عنصر به درون پشته وارد یا از آن خارج می‌شود می‌توان از اشاره‌گر مزبور برای پیدا کردن اولین عنصر استفاده کرد.

۳. برگردید به عقب به پاسخ به تمرین ۱ و پیاده‌سازی در تمرین ۲، می‌بینید که رابط استفاده شده در آنجا کفایت می‌کند پس رابط فوایدی را ایجاد می‌کند که رابطی که به صورت مناسبی تعریف شود دارای این فواید خواهد بود. در اینجا فهرست کوتاهی از فواید آنها ارایه می‌شود:

- رابط پشته را به عنوان یک نوع داده‌ای تعریف می‌کند. با مطالعهٔ رابط می‌توان دقیقاً فهمید که پشته چه کاری انجام می‌دهد.
- رابط نمایش داخلی پشته را کاملاً مخفی می‌کند.
- رابط وظایف پشته را به طور واضح تعریف می‌کند.

روز ۳

پاسخ پرسش‌ها

۱. حساب بانکی دو تابع تغییر دهنده (Mutator) دارد: depositFunds و withdrawFunds(). حساب بانکی یک تابع دست یابنده (Accessor) دارد: getBalance()

۲. دو نوع تابع سازنده وجود دارد: آنهایی که دارای آرگومان هستند و آنهایی که دارای آرگومان نیستند. حساب بانکی از کارگاه ۲ دارای دو نوع تابع سازنده است.

۳. (اختیاری) دسترسی عمومی برای نوع قابل قبول است، زیرا متغیرها ثابت هستند. داشتن دسترسی عمومی به ثابتها در کپسوله سازی اختلال ایجاد نمی‌کند، زیرا پیاده‌سازی را برای استفادهٔ خارجی در معرض دید قرار نمی‌دهد.

بنابراین، استفاده از ثوابت Boolean برای درست (true) و نادرست (false) در مصرف حافظه صرفه جویی می‌کند. به آسانی می‌توان این نوع از ثوابت سراسری را در قسمت‌های مختلف برنامه به اشتراک گذاشت.

٤. نمونه‌های ایجاد شده از کلاس Card تغییرپذیر هستند. تعریف ۵۲ ثابت از کلاس Card بسیار مفید خواهد بود. یعنی برای هر کارت یک ثابت تعریف کنیم. اما نیازی نیست که متغیرهای مختلفی را برای کلاس تعریف کنیم حتی اگر متغیرهای متعددی از کلاس Deck تعریف شده باشد. تقسیم متغیرهای کلاس ورق در میان‌اشیاء مختلف از نوع Deck کاملاً بی خطر خواهد بود.

۵. وقتی کلاسی برای یک شیء طراحی می‌کنید، باید از خود بپرسید که چه چیزی باعث شده است که از این شیء یک کلاس ساخته شود. مخصوصاً برگردید به بحث اینکه چگونه کلاسها، اشیاء مربوط به خود را طبقه بندی می‌کنند.

کارتها بر چه اساس طبقه بندی می‌شوند؟ کارتها بر اساس مقدارشان، نوع خالشان و نحوهٔ نمایششان طبقه بندی می‌شوند. مقدار یا مجموعهٔ یک کارت وجه تمایز یک کارت از انواع دیگر آن نمی‌باشد و آن کارت هنوز به عنوان کارت پوکر (Poker) تلقی می‌شود. کارتهای پوکر ممکن است مقادیر مختلفی داشته باشند. ولی درست مثل یک پستاندار قهوه‌ای که هنوز پستاندار است یک کارت با ۱۰ دل روی آن هنوز یک کارت است.

اگر می‌بینید که رفتار یک شیء به طور اساسی با تغییر مقدار یکی از خواص شیء تغییر می‌کند فرصت دارید که کلاسهایی مجزا ایجاد کنید. هر کلاس برای یکی از مقادیری که ممکن است خاصیت شیء مورد نظر داشته باشد. به بیان واضحتر، مقدار کارت رفتار کارت را به هیچ وجه تغییر نمی‌دهد. پس نیاز به ایجاد کلاسهای مختلف نمی‌باشد.

۶. تقسیم بندی مناسب وظایف، طراحی کلاسهای Dealer و Deck را ماژولار (پیمانه‌ای) می‌کند. به جای یک کلاس بزرگ، کارتهای پوکر به سه کلاس تقسیم بندی می‌شوند، هر کلاس موظف است که کار خودش را انجام دهد و پیاده‌سازی خود را از دیگر کلاسها مخفی نگاه دارد. در نتیجه، این کلاسها به راحتی پیاده‌سازی خود را تغییر می‌دهند بدون اینکه برای دیگران سختی و مرارتی در پی داشته باشد.

با کلاسهای جدا از هم می‌توان از کلاس Card جدای از کلاسهای Dealer (واسط) و Deck (نیز بازی) مجدداً استفاده کرد.

حل تمرین‌ها

۱. روش حل تمرین ۱:

```
public class DoubleKey {

    private Object key1, key2;
```

```
// a no args constructor
public DoubleKey() {
    key1 = "key1";
    key2 = "key2";
}

// a constructor with arguments
// should check for and handle null case
public DoubleKey( Object key1, Object key2 ) {
    this.key1 = key1;
    this.key2 = key2;
}

// accessor
public Object getKey1() {
    return key1;
}

// mutator
// should check for and handle null case
public void setKey1( Object key1 ) {
    this.key1 = key1;
}

// accessor
public Object getKey2() {
    return key2;
}

// mutator
// should check for and handle null case
public void setKey2( Object key2 ) {
    this.key2 = key2;
}

// the following two methods are required in order to properly work as a key
// if passed to a HashMap or Hashtable
public boolean equals( Object obj ) {

    if( this == obj ) {
        return true;
    }

    if( this.getClass() == obj.getClass() ) {
        DoubleKey dk = ( DoubleKey ) obj;
        if( dk.getKey1().equals( getKey1() ) && dk.getKey2().equals( getKey2() ) )
        {
            return true;
        }
    }
    return false;
```

```
    }

    public int hashCode() {
        return key1.hashCode() + key2.hashCode();
    }

}
```

۲. یک روش حل تمرین ۲:

```
public class Deck {

    private java.util.LinkedList deck;

    public Deck() {
        buildCards();
    }

    public String display() {
        int num_cards = deck.size();
        String display = "";
        int counter = 0;
        for( int i = 0; i < num_cards; i ++ ) {
            Card card = ( Card ) deck.get( i );
            display = display + card.display() + " ";
            counter++;
            if( counter == 13 ) {
                counter = 0;
                display = display + "\n";
            }
        }
        return display;
    }

    public Card get( int index ) {
        if( index < deck.size() ) {
            return (Card) deck.get( index );
        }
        return null;
    }

    public void replace( int index, Card card ) {
        deck.set( index, card );
    }

    public int size() {
        return deck.size();
    }

    public Card removeFromFront() {
        if( deck.size() > 0 ) {
```

```
            Card card = (Card) deck.removeFirst();
            return card;
        }
        return null;
    }

    public void returnToBack( Card card ) {
        deck.add( card );
    }

    private void buildCards() {

        deck = new java.util.LinkedList();

        deck.add( new Card( Card.CLUBS, Card.TWO   ) );
        deck.add( new Card( Card.CLUBS, Card.THREE ) );
        deck.add( new Card( Card.CLUBS, Card.FOUR  ) );
        deck.add( new Card( Card.CLUBS, Card.FIVE  ) );
        // full definition cliped for brevity
        // see sourse for full listing
    }

}
```

روز ۴

پاسخ پرسش‌ها

۱. اگر بخواهید از کد نوشته شده به طور مستقیم در جای دیگر مجدداً استفاده کنید، نیاز است که کد از جای خود بریده شود (Cut)، و در محل مورد نظر چسبانده شود (Paste). این کار به چند دلیل مناسب نیست. زیرا کد مزبور در چند جای مختلف دستکاری می‌شود و چندین اشکال ممکن است که پیش آید. کدی که قابل وراثت نباشد کد استاتیک نامیده می‌شود و این کد قابل توسعه نیست. بنابراین کد استاتیک از انواع داده‌ای محدود است و نمی‌تواند انواع داده‌ای را به اشتراک گذارد بنابراین استفاده مجدد از کدهای استاتیک به طور مستقیم در برنامه ما را از بسیاری مزایا محروم می‌کند.

۲. وراثت یک مکانیزم درون ساخت برای استفادهٔ مجدد مطمئن و بی‌خطر و توسعهٔ تعریف‌های موجود یک کلاس می‌باشد. وراثت برقراری رابطهٔ «همانی (Is-a)» را میان کلاسها ممکن می‌سازد.

۳. سه شکل وراثت عبارتند از:
   وراثت برای استفادهٔ مجدد از پیاده‌سازی
   وراثت برای ایجاد تفاوت
   وراثت برای جانشینی

٤. استفاده از وراثت در پیاده‌سازی می‌تواند باعث شود که برنامه‌نویس چشم بسته کاری را انجام دهد.

استفادهٔ مجدد از پیاده‌سازی نباید تنها هدف وراثت باشد. بلکه جانشینی باید به عنوان اولین اولویت معرفی شود. استفادهٔ کورکورانه از وراثت، سلسله مراتبی از کلاسها را ایجاد می‌کند که عملاً با آنها کاری نمی‌توان انجام داد و فاقد توانایی هستند.

۵. برنامه‌نویسی با استفاده از تفاوتها یکی از شکلهای وراثت است. و به این معنا است که وقتی از وراثت استفاده می‌شود بر روی خواصی تکیه می‌شود که در کلاس جدید یا کلاس قدیمی فرق دارد. و این کار باعث کوچکتر شدن و پله‌ای شدن کلاسها می‌شود و مدیریت کلاسهای کوچکتر آسانتر است.

۶. سه نوع از خواص و متدها عبارتند از:

جایگزین‌شده (Overridden)

جدید (New)

بازگشتی (Recursine)

خاصیت یا متد جایگزین‌شده خاصیتی است که در شیء والد تعریف می‌شود و در فرزند دوباره پیاده‌سازی می‌شود تا فرزند همان رفتار را نشان دهد.

متد یا خاصیت جدید، خاصیتی است که در فرزند ظاهر می‌شود اما در والد آن وجود ندارد.

متد یا خاصیت بازگشتی در والد تعریف می‌شود اما در فرزند دوباره تعریف نمی‌شود. فرزند به آسانی آن متد یا خاصیت را به ارث می‌برد. وقتی در فرزند این متد و یا خاصیت فراخوانی می‌شود فراخوانی سلسله مراتبی را طی می‌کند تا به جایی برسد که به آن پاسخ داده شود.

۷. برنامه‌نویسی با استفاده از تفاوتها کلاسهای کوچکتری را تعریف می‌کند که در نتیجه رفتارها و فعالیتها نیز کوچکتر می‌شوند. کلاسهای کوچکتر دارای اشکالات کمتری هستند، رفع اشکال آنها آسانتر است و نگهداری و فهم آنها نیز ساده‌تر می‌باشد.

برنامه‌نویسی با استفاده از تفاوتها شما را قادر می‌سازد که به صورت پله‌ای برنامه‌نویسی کنید. بنابراین طراحیتان زمان کمتری را در بر می‌گیرد.

۸. AllPrermission، BasicPermission و UnresolvedPermission همه فرزندهای Permission هستند. SecurityPermission نیز از اخلاف Permission است.

Permission کلاس اصلی و ریشه است. AllPermission و UnresolvedPermission و SecurityPermission کلاسهای برگ هستند و بنابراین فرزند ندارند. آری Permission نیای SecurityPermission است.

۹. وراثت برای جانشینی، فرایند تعریف ارتباطات جانشینی است. فرایند جانشینی اجازهٔ جانشین کردن نسبی را برای یک نیا می‌دهد بنابراین برای این کار نیازی به استفاده از هیچ متد جدیدی نیست.

۱۰. وراثت می‌تواند با ایجاد دسترسی سهوی یک زیرکلاس به ساختار داخلی یک کلاس والد، کپسوله‌سازی را نابود کند. تخریب سهوی کپسوله سازی دامی است که می‌تواند شما را گیر بیاندازد. وراثت دسترسی کاملاً آزادانه‌ای برای فرزند نسبت به والد خود ایجاد می‌کند. در نتیجه اگر تمهیدات مناسبی اندیشیده شود، فرزند می‌تواند به پیاده‌سازی کلاس والد خود دسترسی مستقیم داشته باشد. دسترسی مستقیم به پیاده‌سازی کلاس خطرناک‌ترین ارتباطی است که میان دو شیء می‌تواند وجود داشته باشد. بسیاری از دامهای این‌چنینی هنوز وجود دارند. برای اجتناب از چنین تخریبی می‌توان

پیاده‌سازی داخلی کلاس والد را از نوع اختصاصی (Private) تعریف کرد. برای اینکه فرزند بتواند به داده‌ها و متدهای کلاس والد دسترسی داشته باشد و در عین حال کپسوله‌سازی تخریب نشود می‌توان داده‌ها را از نوع محافظت شده (Protected) تعریف کرد. اما بیشتر اوقات متدها یا رابطی که با فرزند در ارتباط است از نوع عمومی (Public) هستند.

حل تمرین‌ها

۱. هر زیرکلاس به پیاده‌سازی داخلی کلاس Point دسترسی خواهد داشت. این دسترسی نامحدود کپسوله‌سازی را تخریب می‌کند و کلاس را دچار مشکلاتی می‌کند که در پاسخ پرسش ۱۰ به آن پرداخته شده است.

برای جبران این وضعیت باید x و y را از نوع اختصاصی (Private) تعریف کنیم. توجه داشته باشید که کلاس Point از روی java.awt.Point مدل شده است.

روز ۵

پاسخ پرسش‌ها

۱. در کلاس CheckingAccount، متد public double withdrawFunds(double amount) مثالی از یک متد دوباره تعریف شده می‌باشد. کلاس CheckingAccount تابع withdrawFunds را جایگزین می‌کند تا بتواند از این طریق تعدادی تبادلات مالی را ذخیره و نگهداری کند. در کلاس BankAccount متد public double getBalance() مثالی از یک متد بازگشتی است. این متد در والد ظاهر می‌شود، ولی در هیچ‌یک از زیرکلاسها دوباره تعریف نمی‌شود. بلکه از طریق زیرکلاسها فراخوانی می‌شود و در نهایت در کلاس SavingsAccount متد public double getInterestRate مثالی از یک متد جدید است. این متد فقط در کلاس SavingAccount ظاهر می‌شود اما در والد ظاهر نمی‌شود.

۲. می‌توان از یک کلاس مجرد برای ایجاد وراثت با برنامه استفاده کرد. کلاس مجرد به زیرکلاسها سرنخی ارایه می‌دهد تا مشخص شود که چه متدی نیاز است در آنها دوباره تعریف شود. کلاسهای مجرد استفادهٔ مناسب از کلاس پایه را توسط زیرکلاسها تضمین می‌نماید.

۳. کارگاه ۳ "دارا بودن" را نشان می‌دهد. میز بازی دارای کارتهایی می‌باشد. کلاس Daublekey از کارگاه ۲ دارای دو رشته است.

٤. کارگاهها کپسوله‌سازی را با مخفی کردن همهٔ عنصرهای داده‌ای نشان دادند. اگر نگاهی به راه حل‌ها بیندازید، می‌بینید که همهٔ داده‌ها اختصاصی (Private) هستند. برای مثال کلاس BankAccount متغیر balance را از نوع اختصاصی تعریف می‌کند. در عوض، هر کلاسی دسترسی به نمایش داده را از طریق یک رابط خوب‌تعریف شده فراهم می‌سازد.

۵. کلاس SavingsAccount مثالی از تخصیص است. این کلاس والد خود، کلاس Bank Account، را از

طریق اضافه کردن متدهایی برای اعمال کردن، قرار دادن و گزارش گرفتن از نرخ بهرهٔ حساب بانکی ویژه می‌سازد.

۶. کارگاه ۳ وراثت را برای استفادهٔ مجدد از رفتارهای پایه‌ای تعریف شدهٔ کلاس BankAccount به کار می‌برد. کلاس BankAccount یک پیاده‌سازی عمومی برای برداشتن از حساب، واریز به حساب و گزارش گرفتن از موجودی تعریف می‌کند. زیرکلاسهای حساب بانکی از طریق وراثت این پیاده‌سازی را به ارث می‌برند.

کارگاه ۴ با عنوان کردن نمونه‌ای از وراثت برای پیاده‌سازی شروع می‌شود و با استفاده از ترکیب برای دستیابی به فرم بهتری از استفادهٔ مجدد، خاتمه می‌یابد.

روز ۶

پاسخ پرسش‌ها

۱. ضمنی
پارامتری
جایگزینی
سربارگذاری

۲. چندشکلی‌بودن ضمنی باعث می‌شود که یک شیء رفتار شیء دیگر را داشته باشد. در نتیجه هر شیء رفتارهای مختلفی را از خود نشان می‌دهد.

۳. چندشکلی‌بودن پارامتری و سربارگذاری باعث می‌شود که یک مدل مفهومی از شیء ایجاد شود. در این روش بجای نگرانی در مورد انواع پارامترها، می‌توانید کد را به صورت جامع‌تر بنویسید. یعنی اینکه متدها را بر اساس اینکه چه کاری انجام می‌دهند، مدل کنید.

٤. هر رابط دارای تعدادی پیاده‌سازی است. با برنامه ریزی رابط، دیگر نیازی به وابسته شدن به یک پیاده‌سازی خاص نیست. در نتیجه برنامه می‌تواند به طور اتوماتیک از هر پیاده‌سازی که مناسب باشد استفاده کند. این استقلال از پیاده‌سازی، اجازهٔ استفاده از پیاده‌سازیهای مختلف برای تغییر رفتار برنامه را ممکن می‌سازد.

۵. وقتی متدی جایگزین می‌شود چندشکلی‌بودن این اطمینان را می‌دهد که متد مناسب فراخوانی شود.

۶. چندشکلی‌بودن ویژه نام دیگر سربارگذاری می‌باشد.

۷. سربارگذاری امکان تعریف چندین بارهٔ یک متد را فراهم می‌کند. هر تعریف در تعداد و انواع آرگومانها با دیگری فرق دارد. به هنگام فراخوانی یک متد نیازی به انجام هیچ کاری برای اطمینان از اینکه متد مناسبی فراخوانی شده است، نمی‌باشد.

سربارگذاری اجازهٔ مدل کردن متد را به صورت مفهومی فراهم می‌سازد. طبیعت چندشکلی سربارگذاری عملیات لازم برای پشتیبانی آرگومانهای خاص را صورت می‌دهد.

۸. چندشکلی‌بودن پارامتری باعث می‌شود بتوانید متدها و انواع داده‌ای بسیار جامعی را با به تعویق

انداختن تعریف انواع داده‌ای تا زمان اجرا بنویسید.
این نوع چندشکلی‌بودن باعث می‌شود کد نوشته شده واقعاً طبیعی باشد زیرا باعث می‌شود بتوانید متدها و انواع داده‌ای مفهومی و بسیار جامعی بنویسید. بنابراین متدها و انواع داده‌ای از نگاه مفهومی با توجه به اینکه چه کاری به طور خاص انجام می‌دهند و چه کاری انجام نمی‌دهند، نوشته می‌شوند. برای مثال اگر یک متد مقایسه به صورت compare([T]a,[T]b) نوشته شود، به مفهوم بالاتری که همان مقایسهٔ دو شیء از نوع [T] با یکدیگر می‌باشد پرداخته می‌شود. آرگومانهای نوع [T] با استفاده از ساختاری مثل > یا متد ()compare با یکدیگر مقایسه می‌شوند و نکتهٔ مهم اینست که یک متد به آسانی نوشته می‌شود ولی می‌تواند اشیاء مختلفی را با هم مقایسه کند.

۹. چندشکلی‌بودن معمولاً کارایی سیستم را پایین می‌آورد. چراکه تعدادی از اشکال و پیاده‌سازیهای چندشکلی‌بودن نیاز به بررسی کردن و جستجو دارند. و این ارزیابی کردن است که باعث ایجاد تأخیر در سیستم می‌شود و هزینه‌بر است. البته این هزینه در مقایسه با زبانهایی است که از انواع داده‌ای ایستا استفاده می‌کنند. چندشکلی‌بودن باعث فریب برنامه‌نویس می‌شود تا سلسله مراتب وراثت را از بین ببرد. اما هرگز نباید برای افزایش رفتار چندشکلی، کارایی وراثت را از طریق از بین بردن برنامه‌نویسی سلسله مراتبی پایین آورد. وقتی با یک زیرکلاس به صورت کلاس پایه رفتار شود دسترسی به هر گونه رفتار یا فعالیت اضافه شده توسط کلاس پایه از بین می‌رود. بنابراین هنگامی که یک زیرکلاس جدید ایجاد می‌کنید باید مطمئن شوید که رابط کلاس پایه در برقراری ارتباط با متدهای زیرکلاس جدید که با کلاس پایه کار می‌کنند کافی است.

۱۰. سلسله مراتب مؤثر و کار آمد در برنامه مستقیماً روی پلی مورفیسم ضمنی اثر می‌گذارد. برای استفاده از قابلیت افزودن که توسط چندشکلی‌بودن زیرکلاس ارایه می‌شود، باید سلسله مراتب مناسبی داشته باشید. کپسوله‌سازی شیء را از وابسته شدن به یک پیاده‌سازی خاص جلوگیری می‌کند. بدون کپسوله‌سازی هر شیء به راحتی به پیاده‌سازی داخلی شیء دیگری وابسته می‌شود.
یک چنین پیوند محکمی میان اشیاء جانشینی را نه تنها بسیار سخت، بلکه غیر ممکن می‌سازد.

حل تمرین‌ها

۱. برنامه‌ای را در ذهن خود مجسم کنید که به هنگام اجرا وضعیت خود را روی صفحه نمایش نشان می‌دهند. در جا را، می‌توان به آسانی از تابع ()System.write.println به این منظور استفاده کرد. حال اگر بخواهید این پیغامها را درون یک فایل بنویسید چه می‌کنید؟ یا اگر بخواهید این پیغامها به کامپیوتر دیگری با رابط گرافیکی (UI) فرستاده شود چه می‌کنید؟ آشکار است که باید کد خود را جهت برآورده ساختن این نیازها تغییر دهید.
حال اگر نیاز باشد که دو کار در یک زمان صورت پذیرد چه می‌کنید ؟ بجای اینکه خودتان انتخاب کنید که کدام کار انجام پذیرد به کاربر اجازه می‌دهید از طریق آرگومانهای خط زمان بتواند کار مورد نیاز خود را انتخاب کند.
بدون چندشکلی‌بودن باید برای هر نوع کاری که می‌خواهیم بکنیم فرزندی را برنامه ریزی کنیم ولی با چندشکلی‌بودن می‌توان به آسانی کلاسی تعریف کرد به نام log که دارای یک متد Write باشد.

زیرکلاسها می‌توانند معین کنند که پیغامها کجا ثبت شده‌اند. می‌توان در هر زمانی زیرکلاسهایی را به برنامه اضافه کرد. برنامه به طور اتوماتیک می‌داند که چگونه از زیرکلاسهای جدید به هنگامی که رابطه کلاس log را برنامه ریزی می‌کنید استفاده کند. بنابراین می‌توان در هر زمان وضعیت موجود را به رفتار جدیدی برای کلاس تغییر داد.

۲. `int i = 2 + 3.0`

بسته به تعریف +، این عبارت ممکن است تحمیل باشد. در اینجا، عبارت بیان شده به معنی جمع کردن یک عدد صحیح و یک عدد حقیقی است. حاصل این عبارت در یک متغیر صحیح قرار خواهد گرفت. بسته به زبان برنامه‌نویسی، کامپایلر ممکن است عدد صحیح 2 را به عنوان عدد حقیقی در نظر گرفته بر روی آن عملیاتی انجام دهد و نتیجه را به عدد صحیح برگرداند.

عبارت مذکور بسیار جالب است زیرا نمونه‌ای از سربارگذاری را نشان می‌دهد. + در موارد زیر ممکن است باعث سربارگذاری شود:

```
+(real,real)
+(integer,integer)
+(integer,real)
```

در هر مورد چندشکلی‌بودن ویژه‌ای خواهید داشت زیرا بجای یک متد چندشکلی‌بودن، تعدادی متد یک‌شکلی تحمیلی دارید.

۳. سربارگذاری

به java.util.SimpleTimezone توجه کنید. SimpleTimeZone دو متد سربارگذار شدهٔ زیر را تعریف می‌کند: SetEndRule و SetStartRule. بنابراین این متدها بسته به تعداد و نوع ورودیهایشان پاسخ متفاوت می‌دهند.

چندشکلی‌بودن ضمنی

java.io.Writer را در نظر بگیرید. کلاس مجرد Writer متدهایی را برای نوشتن داده تعریف می‌کند جاوا همزمان با آنان تعدادی زیرکلاس Writer تعریف می‌کند که عبارتند از: BufferedWriter، CharArrayWriter، FilterWriter، OutputStreamWriter، PipedWriter، PrintWriter و StringWriter. هر زیرکلاسی متدهایی را برای نوشتن (Write) (متد سربار گذاری شده)، خالی کردن (Flush) و بستن (Close) دارد.

وقتی برنامه می‌نویسید، باید متدها و اشیاء را به گونه‌ای بنویسید که به عنوان نمونه‌هایی از نوع کلاس Write عمل کنند. از این طریق بسته به اینکه داده چگونه نوشته شود می‌توان از زیرکلاسهای مختلف استفاده کرد. با این شیوهٔ برنامه‌نویسی کلاس Writer رفتارهای مختلفی متناسب با پیاده‌سازی مزبور ارایه می‌دهد.

روز ۷

پاسخ پرسش‌ها

۱. متد ()observe از کلاس PsychiatristObject مثالی از یک متد سربارگذاری شده می‌باشد.

۲. متد ()observe مشکل استفاده از سربارگذاری را به خوبی تشریح می‌کند. هر زمانی که زیرکلاس جدیدی اضافه شود نیاز به اضافه کردن متد سربارگذاری شدهٔ دیگری پیش می‌آید. وقتی تعداد متدها افزایش یابد، نیاز به یافتن راهی برای اضافه کردن یک تابع عمومی برای رفتار یکنواخت با اشیاء و حذف متدهای جایگزین‌شده به وجود می‌آید.

۳. اضافه کردن یک رفتار جدید به سلسله مراتب چندشکلی در طی دو مرحله صورت می‌پذیرد. مرحلهٔ اجراء، ایجاد یک کلاس جدید و مرحلهٔ دوم تغییر برنامه به منظور ایجاد متغیری از کلاس ایجاد شده، می‌باشد. نیازی به تغییر دیگری نیست مگر اینکه لازم باشد از خواص ویژهٔ کلاس جدید استفاده شود.

٤. متد ()examine از کلاس PsychiatristObject مثالی از چندشکلی‌بودن ضمنی است. این متد می‌تواند در هر زیرکلاس MoodyObject کار کند.

۵. با کار اضافه روی داده‌ها، می‌توان از دستورات شرطی پرهیز کرد. اگر داده شیء نبود آن را تبدیل به شیء کنید و اگر شیء بود متدی برای برآورده ساختن رفتارهای مورد نیاز ایجاد کنید. وقتی که این کار انجام شد، می‌توان به جای اینکه کاری روی داده انجام داد، انجام کار را از شیء درخواست کرد.

۶. چندشکلی‌بودن ضمنی اجازه می‌دهد که یک متد با آرگومانهای از نوع یک کلاس یا از نوع زیرکلاسهای آن کار کند. برای هر زیرکلاس نیازی به تعریف متد متفاوتی نمی‌باشد. در چنین شرایطی وجود یک متد تعداد متدهای مورد نیاز در صورت عدم استفاهد از این خاصیت را کاهش می‌دهد. همچنین افزودن خواص جدید ساده‌تر خواهد شد.

۷. در برنامه‌نویسی شیءگرا داده را از شیء درخواست نمی‌کنیم بلکه از آن می‌خواهیم که کاری روی داده انجام دهد.

۸. دستورات شرطی باعث شکسته شدن ارتباطات طرح شده در شمارهٔ ۷ می‌شود. شکسته شدن ارتباطات باعث درهم‌شدن وظایف می‌شود. زیرا هر کاربری می‌خواهد بداند ذات داده چیست و چگونه می‌توان آن را دستکاری کرد.

۹. اگر می‌بینید که با اضافه کردن یک نوع داده‌ای جدید نیاز است که تعدادی شرط و شروط در برنامه به‌روز شود در این صورت دستورات شرطی مسأله‌ساز هستند و اگر مجبور می‌شوید که یک دستور شرطی را در چند جای مختلف بنویسید (پارامتری را فراخوانی کنید که حاوی دستور شرطی است)، در این صورت هم دستورات شرطی مسأله‌ساز هستند.

۱۰. چندشکلی‌بودن باعث می‌شود که با یک زیرکلاس همانند کلاس والد آن رفتار شود. گرچه چندشکلی‌بودن باعث می‌شود رفتار واقعی کلاس مورد نظر مورد استفاده قرار گیرد. اما به نظر می‌رسد که کلاس والد رفتارهای مختلفی از خود نشان می‌دهد.

روز ۸

پاسخ پرسش‌ها

۱. UML زبان مدل‌سازی یکپارچه می‌باشد. UML یک زبان مدل‌سازی استاندارد صنعتی است.

۲. یک متدولوژی چگونگی طراحی نرم‌افزار را تعریف می‌کند و یک زبان مدل‌سازی به نمایش گرافیکی طراحی می‌پردازد. یک متدولوژی اغلب همراه با یک زبان مدل‌سازی متناسب با آن موجود است.

۳. کارگاه، یک رابطهٔ وابستگی را نشان می‌دهد.

٤. در مورد کلاس MoodyObject می‌توان دو عبارت بر زبان آورد. این کلاس دارای متدی به نام queryMood() می‌باشد و همچنین کلاسی مجرد است. استفاده از حروف کج (Italic) برای نام کلاس، نشان می‌دهد که کلاس مجرد است.

۵. رابطهٔ کلاسهای Employee/Payroll از کارگاه ۱ مثالی از یک وابستگی است. متد payEmployees() از کلاس Payroll به رابط عمومی کلاس Employee وابسته است.

۶. هر کدام از این نشانه‌هااطلاعات مفاهیمی را دربر دارند. + به معنی نوع داده‌ای عمومی (Public)، # به معنی نوع داده‌ای محافظت شده (Protected) و به معنای نوع اختصاصی (Private) هستند.

۷. کلاس Queue و عناصرش مثالی از این رابطه هستند.

۸. کلاس Deck دارای کارتهای زیادی است. اگر بخواهید این کلاس را از بین برید باید کارتهای آن را هم از بین ببرید. بنابراین کلاس Deck مثالی از رابطهٔ ترکیب است.

۹. برای تجرید یک کلاس یا متد کافیست نام آن را ایتالیک کنید.

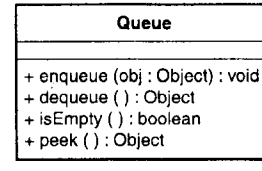
۱۰. هدف نهایی مدل‌سازی انجام طراحی است. در نتیجه نباید از هر علامت موجود برای مدل‌سازی استفاده کنید بلکه باید حداقل علایم ممکن را به کار بگیرید تا به بهترین و ساده‌ترین طرح دست پیدا کنید.

۱۱. یک رابطه (Association) ارتباطات ساختاریی را میان اشیاء مدل می‌کند. اجتماع و ترکیب زیرگروه‌هایی از رابطهٔ میان دو یا چند شیء هستند که ارتباطات «جزیی/کلی» را مدل می‌کنند. اجتماع ارتباط ساختاری میان جفتها است. ترکیب یک ارتباط ساختاری را هر جایی که جزء به کل وابسته است به نمایش می‌گذارد. جزء نمی‌تواند جدای از کل وجود داشته باشد.

۱۲. روابط را زمانی مدل کنید که می‌خواهید نقش‌هایی را میان اشیاء مدل‌سازی کنید. از اجتماع و ترکیب زمانی که می‌خواهید طراحی ساختاری انجام دهید استفاده کنید.

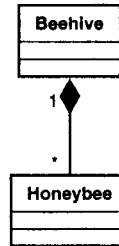
حل تمرین‌ها

۱.



شکل الف - ۱
کلاس Queue

۲.



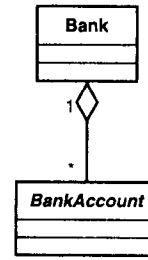
شکل الف - ۲
ارتباط ترکیبی میان دو
کلاس Honeybee/Hive

۳.

شکل الف – ۳

ارتباط اجتماع میان دو

کلاس Bank/BankAccount

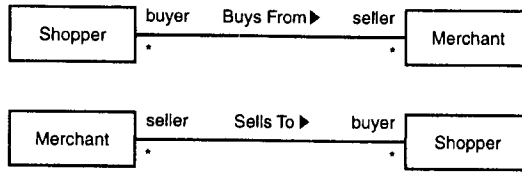


٤.

شکل الف – ۴

رابطهٔ میان دو

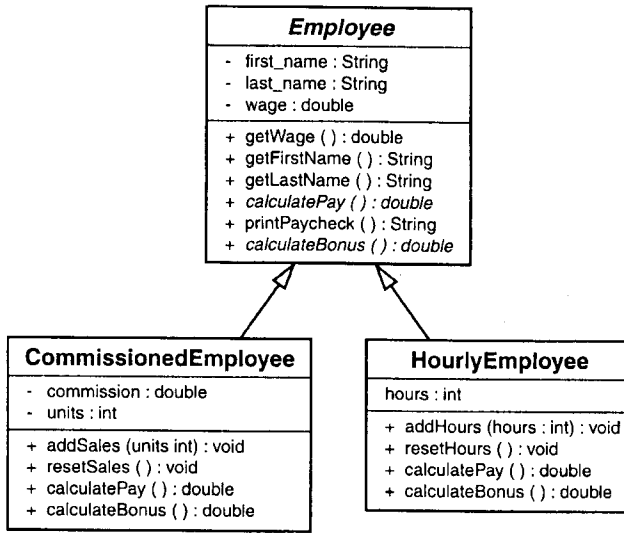
کلاس Shopper/Merchant



۵.

شکل الف – ۵

سلسله مراتب Employee

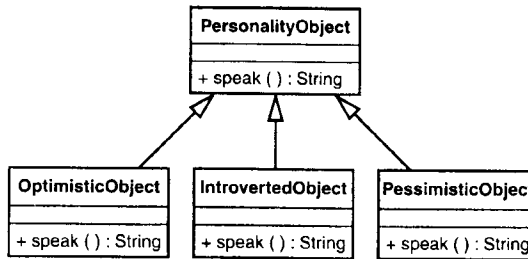


۶.

شکل الف – ۶

سلسله مراتب در

کلاس PersonalityObject



روز ۹

پاسخ پرسش‌ها

۱. یک فرآیند نرم‌افزار مراحل مختلف توسعهٔ نرم‌افزار را طرح می‌کند.

۲. یک فرایند تکراری فرایندی است که باعث می‌شود که به طور مداوم برگشت به عقب انجام گرفته و بر روی نتیجهٔ تکرارهای گذشته دوباره عملیات انجام شود. فرایند تکراری با رجوع مکرر به مراحل تکرار قبلی روند توسعهٔ نرم‌افزار را بهبود می‌بخشد.
توسعه در اینجا به این معنی است که هر تکرار باعث افزایش کارایی نرم‌افزار به میزان کمی می‌شود. نه آنقدر که غیر قابل توجه باشد و نه آنقدر زیاد که هزینه‌بر باشد.

۳. در پایان تحلیل شیءگرا درک خوبی از احتیاجات و ضروریات سیستم و نیز دامنهٔ عمل آن به دست خواهد آمد.

٤. نیازهای سیستم مشخص می‌کنند که کاربران می‌خواهند چه کارهایی با سیستم انجام دهند و چه وظایفی از آنها انتظار می‌رود که انجام دهند. پاسخی از سیستم انتظار دارند.
احتیاجات، خواصی هستند که سیستم برای حل مسایل باید آنها را دارا باشد.

۵. مورد کاربردی رابطه‌ای را میان کاربر سیستم و سیستم تعریف می‌کند و نشان می‌دهد که چگونه کاربر سیستم را از دید کاربری مورد استفاده قرار خواهد داد.

۶. برای تعریف موارد کاربری باید مراحل زیر را پیمود.
   ۱. تعیین عامل‌ها
   ۲. ایجاد فهرست مقدماتی موارد کاربردی
   ۳. نامگذاری و اصلاح موارد کاربردی
   ۴. تعریف سلسله رخدادهای هر مورد کاربردی
   ۵. مدل کردن موارد کاربردی

۷. عامل برای ارتباط برقرار کردن با سیستم به کار می‌رود.

۸. می‌توان برای یافتن عامل‌ها از پاشخ پرسش‌های زیر بهره برد:

- چه کسی اولین بار از سیستم استفاده می‌کند؟
- آیا سیستم دیگری وجود دارد که از این سیستم استفاده کند؟ برای مثال، آیا کاربران غیر انسانی از این سیستم استفاده می‌کنند؟
- آیا سیستم مذکور با هیچ سیستم دیگری تعامل دارد؟ برای مثال آیا بانک اطلاعاتی وجود دارد که نیاز به جمع آوری آن داشته باشید.
- آیا سیستم باید به تأثیرات غیرکاربری هم پاسخ دهد؟ برای مثال آیا نیاز است که سیستم در یک روز خاص از ماه کار خاصی انجام دهد؟ این تأثیرات معمولاً با وجود دید کاربری توسط برنامه‌نویس پشتیبانی نمی‌شوند.

۹. یک مورد کاربردی می‌تواند شامل مورد کاربردی دیگری باشد و از آن استفاده کند یا آن را توسعه دهد.

یک مورد کاربردی همچنین ممکن است متغییر مورد کاربردی دیگری باشد.

۱۰. متغیر یک مورد کاربردی نمونهٔ ویژه‌ای از مورد کاربردی عمومی‌تری است.

۱۱. سناریو سلسله مراتب یا جریانی از رخدادهای میان کاربر و سیستم است.

۱۲. می‌توانید موارد کاربردی را از طریق نمودارهای تعاملی و نمودارهای فعالیت مدل کنید. دو نوع نمودار تعاملی وجود دارند: نمودارهای تسلسلی و همکاری

۱۳. نمودارهای تسلسلی، ترتیبی از رخدادها را مدل می‌کنند. نمودار همکاری تعاملات میان عامل‌های یک مورد کاربردی را می‌کند. هر دو نوع نمودار، نمودارهای تعامل هستند اما هر کدام از آنها دیدگاه متفاوتی نسبت به سیستم دارند. از نمودارهای تسلسلی می‌توان در مواردی که نیاز به دنبال کردن رخدادها است استفاده کرد و از نمودار اشتراک همکاری زمانی که می‌خواهید ارتباطات را پر رنگ کنید می‌توان بهره گرفت. نمودار فعالیت در مدل‌سازی فرآیندهای موازی کمک می‌کند. از نمودارهای فعالیت زمانی می‌توان استفاده کرد که بخواهید فرایندی را مدل کنید که با فرایندهای دیگر در داخل یک سناریوی مورد کاربردی به طور موازی در حال اجرا باشد.

۱۴. مدل دامنه می‌تواند به عنوان پایه یا اسکلت مدل شیء عمل کند. می‌توان از این مدل برای شروع استفاده کرد و آن را ساخت.

مدلها یک دامنهٔ عادی و توانایی درک مسایل را در اختیار شما قرار می‌دهند.

۱۵. موارد کاربردی در درک سیستم احتیاجات آن و استفاده‌هایش کمک زیادی می‌کنند.

موارد کاربردی در طراحی تکرارهای شیء کمک می‌کند.

و در نهایت، می‌توان از موارد کاربردی برای تعریف مدل دامنه استفاده کرد.

حل تمرین‌ها

۱. تعدادی از موارد کاربردی دیگر:

- حذف آیتم: یک کاربر می‌تواند آیتمی را از سبد خرید حذف کند.
- حذف کاربر: یک مدیر سایت می‌تواند دسترسی حسابهای(account) غیر فعال را حذف کند.
- پاداش به کاربر: سیستم می‌تواند به اغلب مشتریان از طریق دادن تخفیف‌های آنی پاداش دهد.

۲. کاربر آیتمی را از سبد خرید انتخاب می‌کند و آیتم انتخاب شده را از این کارت حذف می‌کند.

- حذف آیتم
  ۱. کاربر مهمان آیتمی را از سبد خرید انتخاب می‌کند.
  ۲. کاربر مهمان از سبد می‌خواهد تا آیتم را حذف کند.

- شرایط قبلی
  - سبد شامل آیتمی است که باید حذف شود.
- شرایط بعدی
  - آیتم از سبد حذف می‌شود.

- راه دیگر: لغو عملیات خرید

کاربر ممکن است تعاملات را پس از مرحلهٔ اول منتفی کند.

۳. دو مورد کاربردی زیر دو نوع مختلف از موارد کاربردی جستجو در محصولات کاتالوگ است.

- کاربران مهمان می‌توان در محصولات کاتالوگ جستجو کنند.
- کاربران مهمان می‌توانند آیتم خاصی را جستجو کنند.

دو مورد کاربردی زیر موارد مختلفی از عضویت (Sign up) هستند.

- کاربران ثبت نام شده می‌توانند جهت دسترسی به آخرین اطلاعات سایت عضو شوند.
- کاربران ثبت نام شده می‌توانند در mailing listهای مختلفی عضو شوند.

٤. اشیاء دامنهٔ دیگری هم ممکن است وجود داشته باشند، که در اینجا به فهرست کردن برخی از آنها می‌پردازیم. مدیریت سایت، فهرست محصولات ویژه و فهرست انتخاب‌های مورد نظر.

روز ۱۰

پاسخ پرسش‌ها

۱. سه مزیت در یک طراحی رسمی وجود دارد. اینگونه طراحی در تشخیص اینکه چه اشیایی در برنامه ظاهر می‌شوند و چگونگی ارتباط این اشیاء با یکدیگر مفید است. طراحی کمک می‌کند تا بسیاری از موارد طراحی که در طول پیاده‌سازی به آنها برخورد می‌شود حل و فصل شود. طراحی قبل از اینکه کد نوشته شود ساده‌تر است.

در نهایت، طراحی باعث می‌شود مطمئن شوید که همهٔ برنامه‌نویسان روی یک موضوع یکسان و با اطلاعات و فرضهای مشابه کار می‌کنند. در غیر اینصورت ریسک انجام‌ایجاد تکه‌های نامتناسب توسط برنامه‌نویسان بالا می‌رود.

۲. طراحی شیءگرا (OOD) فرایند ساخت یک مدل حل مسأله است. به بیان دیگر، OOD فرآیند تبدیل یک راه حل به تعدادی اشیاء تشکیل دهنده است.

۳. مدل شیءگرا به طراحی اشیاء که در حل مسأله ظاهر می‌شوند می‌پردازد. مدل نهایی شیءگرا ممکن است شامل اشیایی باشد که در دامنه یافت نشوند. مدل شیءگرا وظایف مختلف اشیاء، ارتباطات و ساختار آنها را توصیف می‌کند.

٤. دیدن تمامی راه‌های طراحی ممکن قبل از پرداختن به آن غیرممکن است و همواره هم به زمانی که صرف ان می‌شود نمی‌ارزد. قسمتی از طراحی را هم می‌توان به زمان پیاده‌سازی واگذار کرد و البته از و می به انجام طراحی کامل نیست. بالاخره باید زمانی کدنویسی را شروع کنید.

۵. بخش‌های مهم، آن قسمت هایی از سیستم هستند که به طور کامل رفتار یا ساختار سیستم را تغییر می‌دهند. اینها، قسمت‌هایی هستند که واقعاً در حل مسأله اهمیت دارند. تغییر در یک تکه مهم معماری ساختار حل برنامه را تغیر خواهد داد.

۶. پنج مرحلهٔ پایه‌ای OOD عبارتند از:

۱. ایجاد فهرست اولیه‌ای از اشیاء
۲. تعریف دوبارهٔ وظایف اشیاء با کمک کارتهای CRC
۳. برنامه‌ریزی نقاط تعامل
۴. جزیی کردن ارتباطات میان اشیاء
۵. ساخت مدل

۷. با دامنه شروع کنید تا فهرست اولیه‌ای از اشیاء مورد نیاز را ایجاد کنید. هر شیء دامنه و عامل باید در مدل شیءگرا، دارای کلاسی باشد.
باید برای هریک از سیستم‌های دیگر، رابطهای سخت‌افزاری، گزارشها، نمایشها و ابزارهای سیستم کلاسی در نظر گرفته شود.

۸. یک طرح کامل شامل وظایف هر شیء، ساختار و ارتباطات میان اشیاء خواهد بود. طرح نشان می‌دهد که چگونه اجزاء در کنار یکدیگر قرار می‌گیرند.

۹. کارتهای CRC به تشخیص وظایف و همکاریهای یک کلاس کمک می‌کند.

۱۰. همکاری ارتباط واگذاری میان دو شیء است. می‌توان به این نوع همکاری به صورت یک ارتباط Client/Server میان دو شیء نگاه کرد.

۱۱. عملاً، وظایف تبدیل به متدهایی می‌شوند و رابطه‌ها به ساختار تبدیل می‌شوند. ولی با وجود این، درک همه جانبه‌ای از وظایف به تقسیم آنها میان اشیاء به طور کار آمد و مؤثر کمک می‌کند. همواره باید از داشتن مجموعه‌ای کوچک از اشیاء خیلی بزرگ احتراز کنید. از طریق طراحی مطمئن می‌شوید که وظایف اشیاء بخش شده‌اند.

۱۲. کارتهای CRC کارتهای ۶×۴ هستند که با کاوش موارد کاربردی، به کشف وظایف و همکاریهای میان اشیاء کمک می‌کنند.

۱۳. گاهی در سایز کارتهای CRC، با محدودیت مواجه می‌شوید. اگر متوجه شوید که تعریف کلاس روی یک کارت جا نمی‌شود، بدانید که به آن کلاس بیش از حد وظیفه واگذار کرده‌اید.

۱٤. از کارتهای CRC باید در مراحل اولیه برنامه‌ریزی استفاده کرد، مخصوصاً اگر در برنامه‌نویسی شیءگرا تازه کار باشید. کارتهای CRC در پروژه‌های کوچک یا بخش کوچکی از پروژه‌های بزرگتر استفاده می‌شوند.
از کارتهای CRC فقط برای مشخص کردن وظایف و همکاریها استفاده کنید. سعی نکنید که از کارتهای CRC جهت توصیف ارتباطات پیچیده استفاده کنید.

۱۵. کارتهای CRC برای پروژه‌های بزرگ یا گروههای توسعه یافته چندان کارآمد نیستند. زیاد بودن کلاسها می‌تواند کارتهای CRC را بلا استفاده کند. از طرفی دیگر زیاد شدن تعداد برنامه‌نویسان هم همین نتیجه را ایجاد می‌کند.

۱۶. "نقطهٔ تعامل" جایی است که یک شیء از دیگری استفاده می‌کند.

۱۷. در نقطه تعامل باید به انتقال داده، تغییرات آینده، رابطها و استفاده از عامل‌ها توجه کرد.

۱۸. "عامل" شیئی است که میان دو یا چندا شیء میانجیگری می‌کند تا به اهدافی دست یابد.

۱۹. برنامه‌نویس به توصیف وابستگی‌ها، روابط و تعمیم‌ها می‌پردازد. جزیی کردن این رابطه ها مرحلهٔ مهم از کار است. زیرا مشخص می‌کند چگونه اشیاء در کنار یکدیگر قرار می‌گیرند. همچنین ساختار داخلی اشیاء مختلف را توصیف می‌کند.

۲۰. ممکن است نمودار کلاس، نمودار فعالیت و نمودار تعاملی برای مدلسازی طراحی ایجاد شود. به علاوه UML (زبان مدلسازی یکپارچه)، نمودارهای شیء و نمودارهای حالت را تعریف می‌کند.

حل تمرین‌ها

۱. کلاس ShoppingCart وظیفهٔ نگهداری خریدها را برعهده دارد. مثلاً می‌تواند آیتمی را به خودش اضافه کند، یا از خودش حذف کند یا به یک شیء خارجی اجازه دهد که آیتمی را بدون حذف کردن انتخاب کند.

روز ۱۱

پاسخ پرسش‌ها

۱. کلاس Adapter رابط یک شیء را به آنچه که برنامه بدان نیازمند است، تبدیل می‌کند. در واقع یک کلاس Adapter شامل یک شیء دیگر است و پیغامها را از رابط جدید گرفته و به رابط شیء موجود در خود ارسال می‌کند.

۲. الگوی تکرارکننده مکانیزمی برای شمارش عناصر درون یک کلکسیون ارایه می‌کند.

۳. از الگوی تکرار کننده برای مواقعی که بخواهیم جزییات پیاده‌سازی شمارش و دسترسی به عناصر یک کلکسیون را از دید کاربر پنهان کنیم، استفاده می‌کنیم.

٤. از الگوی وفقی (Adapter) برای تبدیل رابط یک شیء استفاده می‌کنیم.

۵. از الگوی وفقی زمانی استفاده می‌کنیم که بخواهیم شیئی را با رابط ناسازگار به خدمت بگیریم. برای این مثال می‌توان از تکنیکهای مختلفی استفاده کرد تا در برابر تغییرات توابع (API) نیز کلاس وفقی در امان بماند.

۶. الگوی رابط (Proxy) دسترسی غیر مستقیم به شیء را ممکن می‌سازد. در واقع از این طریق دسترسی غیر مستقیم به اشیاء مختلف را می‌توان فراهم کرد.

۷. از الگوی رابط برای مواقعی که بخواهیم به شئی دسترسی داشته باشیم و نتوان ارجاع مستقیمی به آن داشت، استفاده می‌کنیم. مثالهای رایج عبارتند از: منابع موجود در کامپیوترهای دور، بهینه سازی، شمارش ارجاعات به اشیاء و محافظت کلی از اشیاء.

۸. در این مورد می‌توان از الگوی وقفی برای ساخت رابطی مستقل از آنچه که توسط Sun، IBM و Apache آماده شده است بهره برد. با ایجاد این رابط می‌توان از دیگر توابع API سازندگان مختلف نیز مستقل ماند. در اینصورت با ارتقاء نسخه‌های کتابخانه‌ای یا انتخاب کتابخانه‌ای دیگر، مشکلی پیش نخواهد آمد.

۹. برای این مورد می‌توان از الگوی واسط استفاده کرد تا هویت شیء یا اشیاء مخفی بماند. بسته به موقعیت سرویس گیرندگان (Clients) می‌توان از واسط شبکه‌ای یا یک واسط محلی استفاده کرد. به این ترتیب باقی اجزای برنامه متوجه این اختلاف نخواهد شد. بنابراین تمام اشیاء برنامه می‌توانند از یک رابط Proxy استفاده کنند بدون آنکه نگران پیاده‌سازی درونی آن باشند.

۱۰. الگوی واسط رابط درونی خود را تغییر نمی‌دهد ولی آزاد است تا خواص یا قیدهای جدیدی به رابط به آن اضافه کند.

حل تمرین‌ها

لیست ۱۱-۱۳ ShoppingCart.java

```
public class ShoppingCart {

    java.util.LinkedList items = new java.util.LinkedList();

    /**
     * adds an item to the cart
     * @param item the item to add
     */
    public void addItem( Item item ) {
        items.add( item );
    }

    /**
     * removes the given item from the cart
     * @param item the item to remove
     */
    public void removeItem( Item item ) {
        items.remove( item );
    }

    /**
     * @return int the number of items in the cart
     */
    public int getNumberItems() {
        return items.size();
    }

    /**
     * retrieves the indexed item
     * @param index the item's index
     * @retun Item the item at index
     */
    public Item getItem( int index ) {
        return (Item) items.get( index );
    }
```

لیست ۱۱-۱۳ (ادامه)

```
    public Iterator iterator() {
        // ArrayList has an iterator() method that returns an iterator
        // however, for demonstration purposes it helps to see a simple iterator
        return new CartIterator( items );
    }
}
```

لیست ۱۱-۱٤ CartIterator.java

```
public class CartIterator implements Iterator {

    private Object [] items;
    private int index;

    public CartIterator( java.util.LinkedList items ) {
        this.items = items.toArray();
    }

    public boolean isDone() {
        if( index >= items.length ) {
            return true;
        }
        return false;
    }

    public Object currentItem() {
        if( !isDone() ) {
            return items[index];
        }
        return null;
    }

    public void next() {
        index++;
    }
    public void first() {
        index = 0;
    }

}
```

۲. با ایجاد وفق دهنده mutable، می‌توان از یگ الگو برای پوشش دادن بسیاری از اشیاء استفاده کرد و در اینصورت نیازی نیست برای هر یک از اشیاء یک کلاس خاصی ساخت. در اینجاست از حافظه به نحو بهتری استفاده می‌شود و برنامه از لزوم تعریف کلاسهای بی‌شمار خلاصی می‌یابد.

لیست ۱۱-۱۵ MutableAdapter.java

```
public class MutableAdapter extends MoodyObject {

    private Pet pet;
```

لیست ۱۱-۱۵ MutableAdapter.java

```
    public MutableAdapter( Pet pet ) {
        setPet( pet );
    }

    protected String getMood() {
        // only implementing because required to by
        // MoodyObject, since also override queryMood
        // we don't really need it
        return pet.speak();
    }

    public void queryMood() {
        System.out.println( getMood() );
    }

    public void setPet( Pet pet ) {
        this.pet = pet;
    }
}
```

روز ۱۲

پاسخ پرسش‌ها

۱. یک کلاس پوشش دهنده (Wrapper Class) رابط یک شیء را به آنچه که برنامه به آن نیاز دارد، تبدیل می‌کند. در واقع کلاس پوشش‌دهنده یک شیء را دربر می‌گیرد و پیغامها را از رابط جدید به سمت رابط شیء موجود هدایت می‌کند.

۲. الگوی مجرد عامل مکانیزمی فراهم می‌کند تا بتوان بدون نیاز به دانستن دقیق اینکه کدام کلاس خلف در نظر گرفته شده، متغیر تعریف کرد. یعنی می‌توان کلاسهای خلف مختلفی را در سیستم مورد استفاده قرار گرفت.

۳. از الگوی عامل مجرد زمانی استفاده می‌شود که بخواهیم جزییات ایجاد اشیاء مختلف را پنهان کنیم و یا اینکه بخواهیم تعدادی از اشیاء با یکدیگر در ارتباط باشند و با همدیگر به کار گرفته شوند.

٤. از الگوی تک‌برگ برای مواقعی که بخواهیم اطمینان داشته باشیم که شیئی تنها یکبار تعریف شده است، استفاده می‌کنیم.

۵. از الگوی تک‌برگ برای اینکه بخواهیم تنها یک شیء از یک کلاس خاص ایجاد شود، استفاده می‌کنیم.

۶. استفاده از ثابتهای ساده و ابتدایی روشی مناسب برای برنامه‌نویسی شیءگرا نیست. چراکه باید برای هر یک از این ثابتها معانی قراردادی در نظر گرفت.

از طریق الگوی شمارشی نوع حفاظت شده می‌توان این مشکل را برطرف کرد. در این حالت ثابتها بدل به اشیاء سطح بالاتری خواهند شد. در اینصورت می‌توان مسؤولیت‌پذیری را نیز به این اشیاء تزریق کرد.

۷. زمانی از الگوی شمارشی نوع حفاظت شده استفاده می‌کنیم که ثابتهای عمومی در برنامه وجود دارند که می‌توان آنها را به اشیایی از نوع شمارشی تبدیل کرد.

۸. خیر، الگوها تضمینی از طراحی صحیح ارایه نمی‌کنند چرا که ممکن است از الگوی در جای نامناسب استفاده شود. در ضمن استفاده از الگوها به این معنی نخواهد بود که باقی طرح موجود درست است. بسیاری طرحهای خوب وجود دارند که از الگوها استفاده نکرده‌اند.

حل تمرین‌ها

۱.

لیست ۱۲-۱۹ Bank.java

```
public class Bank {

    private java.util.Hashtable accounts = new java.util.Hashtable();

    private static Bank instance;

    protected Bank() {}

    public static Bank getInstance() {
        if( instance == null ) {
            instance = new Bank();
        }
        return instance;
    }

    public void addAccount( String name, BankAccount account ) {
        accounts.put( name, account );
    }

    public double totalHoldings() {
        double total = 0.0;

        java.util.Enumeration enum = accounts.elements();
        while( enum.hasMoreElements() ) {
            BankAccount account = (BankAccount) enum.nextElement();
            total += account.getBalance();
        }
        return total;
    }

    public int totalAccounts() {
        return accounts.size();
    }

    public void deposit( String name, double ammount ) {
        BankAccount account = retrieveAccount( name );
        if( account != null ) {
```

لیست ۱۲ـ۱۹ (ادامه)

```
            account.depositFunds( ammount );
        }
    }

    public double balance( String name ) {
        BankAccount account = retrieveAccount( name );
        if( account != null ) {
            return account.getBalance();
        }
        return 0.0;
    }

    private BankAccount retrieveAccount( String name ) {
        return (BankAccount) accounts.get( name );
    }
}
```

۲.

لیست ۱۲ـ۲۰ Level.java

```
public final class Level {

    public final static Level NOISE   = new Level( 0, "NOISE" );
    public final static Level INFO    = new Level( 1, "INFO" );
    public final static Level WARNING = new Level( 2, "WARNING" );
    public final static Level ERROR   = new Level( 3, "ERROR" );

    private int    level;
    private String name;

    private Level( int level, String name ) {
        this.level = level;
        this.name  = name;
    }

    public int getLevel() {
        return level;
    }

    public String getName() {
        return name;
    }
}
```

لیست ۱۲ـ۲۱ Error.Java

```
public class Error {

    private Level level;
```

لیست ۱۲-۲۱ (ادامه)

```
    public Error( Level level ) {
        this.level = level;
    }

    public Level getLevel() {
        return level;
    }

    public String toString() {
        return level.getName();
    }
}
```

۳. راه حل شامل یک عامل حساب بانکی مجرد، که به صورت یک رابط نوشته شده (اگر چه می‌تواند یک کلاس مجرد باشد) و یک عامل حساب بانکی یکپارچه می‌باشد. عامل برای ایجاد هر نوع حساب بانکی دارای یک متد می‌باشد.

این عامل جزییات نمونه‌سازی را مخفی می‌کند، نه لزوماً زیرکلاس یک شیء را.

لیست ۱۲-۲۲ AbstractAccountFactory.java

```
public interface AbstractAccountFactory {

    public CheckingAccount createCheckingAccount( double initDeposit, int trans, double fee );

    public OverdraftAccount createOverdraftAccount( double initDeposit, double rate );

    public RewardsAccount createRewardsAccount( double initDeposit, double interest, double min );

    public SavingsAccount createSavingsAccount( double initBalance, double interestRate );

public TimedMaturityAccount createTimedMaturityAccount( double initBalance, double interestRate,
double feeRate );

}
```

لیست ۱۲-۲۳ ConcreteAccountFactory.java

```
public class ConcreteAccountFactory implements AbstractAccountFactory {

    public CheckingAccount createCheckingAccount( double initDeposit, int trans, double fee ) {
        return new CheckingAccount( initDeposit, trans, fee );
    }

    public OverdraftAccount createOverdraftAccount( double initDeposit, double rate ) {
        return new OverdraftAccount( initDeposit, rate );
    }
```

لیست ۱۲ـ۲۳ (ادامه)

```
    public RewardsAccount createRewardsAccount( double initDeposit, double interest, double min ) {
        return new RewardsAccount( initDeposit, interest, min );
    }

    public SavingsAccount createSavingsAccount( double initBalance, double interestRate ) {
        return new SavingsAccount( initBalance, interestRate );
    }

public TimedMaturityAccount createTimedMaturityAccount( double initBalance, double
interestRate, double feeRate ) {
        return new TimedMaturityAccount( initBalance, interestRate, feeRate );
    }

}
```

روز ۱۳

پاسخ پرسش‌ها

۱. تحلیل، طراحی و پیاده‌سازی یک رابط کاربری (UI) با بقیهٔ سیستم متفاوت نیست. تمامی مراحل برنامه‌نویسی رابط کاربری مشابه نوشتن سایر اجزای نرم‌افزاری است. باید به رابط کاربری به همان اندازهٔ اجزای دیگر توجه شود. باید مطمئن شوید که از توجه به این بخش غفلت نکرده‌اید.

۲. باید رابطهای کاربری را از یکدیگر جدا کرد تا سیستم و رابط کاربری به یکدیگر پیوند نخورد. چرا که زمانی که رابط کاربری با هسته و قسمت اصلی سیستم پیوند بخورد ایجاد تغییرات به مشکل برخواهد خورد.
همچنین به اشتراک گذاشتن سیستم میان رابطهای کاربری یا انواع آن زمانی که این بخش با سیستم پیوند می‌خورد غیر ممکن است.

۳. سه بخش الگوی MVC عبارتند از مدل (Model)، نما (View) و کنترل کننده (Controller).

٤. الگوی PAC و مدل Document/View دو جایگزین ممکن الگوی MVC هستند.

۵. مدل لایه‌ای از مجموعهٔ سه عنصری MVC می‌باشد که رفتار هستهٔ و حالت سیستم را مدیریت می‌کند. کنترل کننده از مدل برای انگیزش رفتار سیستم بهره می‌گیرد. نما هم از مدل برای برگرداندن اطلاعات حالت جهت نمایش استفاده می‌کند.
مدل همچنین مکانیزمی جهت تغییر اعلان فراهم می‌کند. کنترل کننده از این مکانیزم برای همگام شدن با تغییرات حالت در مدل استفاده می‌کند.

۶. نما نیز عنصری از اعضای سه گانهٔ MVC است، که مسؤول نمایش مدل به کاربر است.

۷. کنترل کننده مسؤول تغییر و ترجمهٔ رخدادهایی است که توسط کاربر ایجاد می‌شوند. این بخش برای پاسخگویی به این رخدادها، رفتار مدل یا نما را بر می‌انگیزد.

۸. هر سیستم ممکن است دارای چندین مدل باشد. هر مدل ممکن است چندین نما داشته باشد. هر نما

ممکن است یک کنترل کننده داشته باشد و هر کنترل کننده ممکن است فقط یک نما را کنترل کند.

۹. عدم کارایی ممکن است در مدل، نما و کنترل کننده رخ دهد. مدل باید از رخداد اعلان‌های تغییر حالت غیر ضروری جلوگیری کند. نما و کنترل کننده باید داده را هر زمان که ممکن باشد، ذخیره (Cache) کنند.

۱۰. در الگوی MVC فرض بر این است که یک مدل پایدار و یک نمایش در حال تغییر موجود است.

۱۱. خلاصه‌ای کوتاه شده از تاریخچه و انگیزهٔ موجود در پس الگوی MVC در کتاب "برنامه‌نویسی کاربردی در Smalltalk-80: چگونه از مدل-نما-کنترل‌کننده (MVC) استفاده کنیم؟" نوشتهٔ دکتر استیو باربک آمده است. می‌توان یک کپی از این کتاب را در سایت زیر یافت:

http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

حال، نکتهٔ این پرسش چیست؟ پاسخ به این پرسش دید مهمی را دربارهٔ انگیزهٔ موجود در پس الگوی MVC ارایه می‌دهد. با خواندن تاریخچهٔ آمده در این کتاب، می‌توان فهمید که الگوی MVC ابتدا به صورت قسمتی از زبان برنامه‌نویسی Smalltalk طرح ریزی شده است و امروزه تقریباً در هر زبانی مورد استفاده دارد. از این نکته می‌توان اصل مهمی را نتیجه گرفت: کاربرد الگوها وابسته به زبان خاصی نیست و در هر زبانی با خصوصیات لازم خود کار می‌کنند.

MVC ربطی به Java یا Smalltalk ندارد. بلکه روشی در طراحی است که از سطح زبان پیاده‌سازی فراتر است.

حل تمرین‌ها

۱. لیست ۱۳-۱۱ کلاس جدید Employee را نمایش می‌دهد.

لیست ۱۳-۱۱ Employee.java

```
import java.util.ArrayList;
import java.util.Iterator;
public abstract class Employee {

    private String first_name;
    private String last_name;
    private double wage;
    private ArrayList listeners = new ArrayList();

    public Employee(String first_name,String last_name,double wage) {
        this.first_name = first_name;
        this.last_name  = last_name;
        this.wage = wage;
    }

    public double getWage() {
         return wage;
    }

    public void setWage( double wage ) {
        this.wage = wage;
```

لیست ۱۳-۱۱ (ادامه)

```
        updateObservers();
    }

    public String getFirstName() {
        return first_name;
    }

    public String getLastName() {
        return last_name;
    }

    public abstract double calculatePay();

    public abstract double calculateBonus();

    public void printPaycheck() {
        String full_name = last_name + ", " + first_name;
        System.out.println( "Pay: " + full_name + " $" + calculatePay() );
    }

    public void register( Observer o ) {
        listeners.add( o );
        o.update();
    }

    public void deregister( Observer o ) {
        listeners.remove( o );
    }

    private void updateObservers() {
        Iterator i = listeners.iterator();
        while( i.hasNext() ) {
            Observer o = (Observer) i.next();
            o.update();
        }
    }

}
```

۲. لیست ۱۳-۱۲ پیاده‌سازی جدیدی از کلاس BankAccountController را نمایش می‌دهد.

لیست ۱۳-۱۲ BankAccountController.java

```
public class BankAccountController implements BankActivityListener {

    private BankAccountView  view;
    private BankAccountModel model;

    public BankAccountController( BankAccountView view, BankAccountModel model ) {
```

لیست ۱۳ـ۱۲ (ادامه)

```
        this.view  = view;
        this.model = model;
    }

    public void withdrawPerformed( BankActivityEvent e ) {
        double amount = e.getAmount();
        model.withdrawFunds( amount );
    }

    public void depositPerformed( BankActivityEvent e ) {
        double amount = e.getAmount();
        model.depositFunds( amount );
    }

}
```

خواندن این نسخه از کلاس BankAccountController نسبت به نسخهٔ اصلی آن ساده‌تر است. اگر چه، view پیچیده‌تر شده است.

روز ۱۴

پاسخ پرسش‌ها

۱. خطاها از اشتباهات تایپی، منطق ناصحیح یا اشتباهات پیش‌پاافتاده‌ای که در طول کدنویسی پیش می‌آیند ناشی می‌شوند. خطاها همچنین در اثر تعامل نادرست میان اشیاء و یا نقایص موجود در تحلیل و طراحی ایجاد می‌شوند.

۲. حالت آزمون بلوک سازمان یافته‌ای برای آزمایش می‌باشد. هر نوع آزمون از حالتهای آزمون ساخته شده است، و هر حالت آزمون جنبه‌ای از سیتم را مورد آزمایش قرار می‌دهد.

۳. می‌توان حالات آزمون را بر اساس آزمون جعبهٔ سیاه یا آزمون جعبهٔ سفید تقسیم‌بندی کرد.

٤. آزمونهای جعبهٔ سفید بر اساس ساختار کد اصلی برنامه است. این دو آزمون کلی که برنامه تحت پوشش قرار داده می‌شود و تمامی آن آزمایش می‌شود.
آزمونهای جعبهٔ سیاه بر اساس ویژگیهاو خصوصیات سیستم است. در این آزمون صحت رفتار سیستم مورد بررسی قرار می‌گیرد.

۵. چهار نوع آزمون عبارتند از: آزمون واحد، آزمون مجتمع، آزمون سیستم و آزمون بازگشتی.

۶. آزمون واحد، پایین‌ترین سطح آزمون می‌باشد. در این آزمون به یک شیء پیغامی ارسال می‌شود و بررسی می‌شود که آیا شیء نتیجهٔ پیش بینی شده را بر می‌گرداند یا خیر. در این آزمون در هر زمان تنها یک ویژگی مورد بررسی قرار می‌گیرد.

۷. آزمون مجتمع به بررسی صحت تعاملات میان اشیاء می‌پردازد. در آزمون سیستم مشخص می‌شود

که آیا سیستم همانگونه که در موارد کاربردی تعریف شده است رفتار می‌کند و آیا می‌تواند موارد پیش‌بینی‌نشده را به خوبی رفع و رجوع کند یا خیر.

۸. نباید آزمایش را به انتهای عملیات واگذار کرد. انجام آزمایش به هنگام برنامه‌نویسی یافتن و تصحیح خطاها را آسانتر می‌کند. اگر تست برای مرحلهٔ پایانی گذاشته شود، با خطاهای زیادی مواجه خواهید شد که یافتن و تصحیح آنها مشکل‌تر است.
آزمایش به هنگام برنامه‌نویسی تغییر کد برنامه را ساده‌تر می‌کند و باعث پیشرفت و بهینه‌سازی طراحی می‌شود.

۹. تصحیح خطاها به شیوهٔ دستی یا بصری خود باعث ایجاد خطا می‌شود. بنابراین باید تا جایی که امکان دارد از این شیوه اجتناب شود و به جای آن به یک مکانیزم اتوماتیک جهت خطایابی اعتماد شود.

۱۰. چهار چوب (framework) مدل دامنهٔ قابل استفادهٔ مجددی را تعریف می‌کند. می‌توان در این مدل از کلاسها به عنوان پایه و اساس کاربردهای ویژه استفاده کرد.

۱۱. شیء مجازی جانشین ساده‌ای برای یک شیء واقعی است که کمک می‌کند بتوانید آزمون واحد اشیاء خود را صورت دهید.

۱۲. اشیاء مجازی اجازه می‌دهند که آزمون واحد کلاسها به تنهایی و بدون ارتباط با بقیهٔ کلاسها انجام شود. آنها همچنین امکان آزمایش کردن مواردی که در شرایط عادی ترتیب دادنشان مشکل یا غیرممکن است را فراهم می‌کنند.

۱۳. خطا ناشی از عیب و نقص موجود در سیستم است. شرایط خطا، اشتباه یا اشکال نیست بلکه شرایطی است که سیستم باید برای رویارویی با آن آماده باشد و بتواند از آن موفق بیرون بیاید.

۱۴. وقتی که برنامه را می‌نویسید می‌توانید از کیفیت آن از طریق آزمونهای واحد، ردیابی خطاها به شکل مناسب و مستندسازی مناسب اطمینان حاصل کنید.

حل تمرین‌ها

۱. کلاس Cookstour در طراحی به شما دید می‌دهد و ایدهٔ موجود در پس JUnit را به وضوح نشان می‌دهد.

۲. لیست ۱۴-۱۲ یک آزمون واحد ممکن را نمایش می‌دهد.

لیست ۱۴-۱۲ HourlyEmployeeTest .java

```
public class HourlyEmployeeTest extends TestCase {

    private HourlyEmployee emp;

    private static final String FIRST_NAME = "FNAME";
    private static final String LAST_NAME  = "LNAME";
    private static final double WAGE       = 500.00;

    protected void setUp() {
       emp = new HourlyEmployee( FIRST_NAME, LAST_NAME, WAGE );
    }

    public void test_calculatePay() {
```

لیست ۱۴ـ۱۲ (ادامه)

```
        emp.addHours( 10 );

        double expected = WAGE * 10;
        assertTrue( "incorrect pay calculation", emp.calculatePay() == expected );
    }

    public HourlyEmployeeTest( String name ) {
        super( name );
    }
}
```

روز ۱۵

پاسخ پرسش‌ها

۱. کلاس PlayerListener مثالی از یک الگوی قابل رؤیت است. کلاس Console از الگوی تک‌برگ استفاده می‌کند. کلاسهای Rank و Suit از الگوی شمارشی‌بانوع‌حفاظت‌شده استفاده می‌کنند.

۲. کلاس BlackjackDealer با کلاس HumanPlayer به صورت چند شکلی به عنوان یک Player رفتار می‌کند. می‌توان بازیگران غیرانسان ایجاد کرد و کلاس BlackjackDealer باید بداند که کلاس Blackjack چگونه با آنها بازی می‌کند.

۳. گروه کلاسهای Player/BlakjackDealer/HumanPlayer مثالی از سلسله مراتب وراثت است.

٤. کلاس Deck اشیاء Card خود را کاملاً کپسوله می‌کند و هیچ تابع getter یا setterای برای دسترسی به آنها فراهم نمی‌کند. به جای آن کلاسهای Card را به کلاس Deckpiles اضافه می‌کند.

۵. کلاسهای BlackjackDealer و HumanPlayer به صورت چندشکلی عمل می‌کنند به این صورت که نسخه‌های سفارشی از متد ()hit فراهم می‌کنند. وقتی متد ()Play متد ()hit را فراخوانی می‌کند رفتار متد ()play بر اساس پیاده‌سازی متد ()hit متفاوت خواهد بود.

حل تمرین‌ها

۱. موجود نیست

۲. موجود نیست

روز ۱۶

پاسخ پرسش‌ها

۱. دستورات شرطی زمانی که مسؤولیت را از یک شیء می‌گیرند و در جای دیگری قرار می‌دهند،

خطرناک هستند. رفتاری که متعلق به یک شیء است نباید در کل برنامه پخش شود. در منطق توزیع، باید به جای متمرکز کردن رفتار در یک جای خاص، آن را در کل برنامه پخش کرد.
یک دلیل دیگر خطرناک بودن دستورات شرطی آن است که آزمایش یک شیء و پوشش همهٔ مسیرهایی که به آن شیء ختم می‌شود را بسیار مشکل می‌سازند.

۲. قبلاً دیدید که می‌توان از چندشکلی‌بودن برای حذف دستورات شرطی استفاده کنید. در درس امروز دیدید که می‌توان از ترکیبی از چندشکلی‌بودن و ایدهٔ حالت برای حذف دستورات‌شرطی استفاده کنید. حالتها راهکاری عالی برای پیاده‌سازی قوانین می‌باشند.

۳. نسخهٔ قبلی کلاس Hand (کلاس دسته کارتها) برنامه‌نویس را مجبور می‌کرد که کلاسهای مختلف Card که به آن تعلق داشتند را خود با یکدیگر مقایسه کند. تا ببینید که آیا با یکدیگر مساوی هستند یا ارزش یک دسته کارت از دیگری بیشتر است. کلاس Hand اکنون این مقایسه را بدون روکردن حالت داخلی خود انجام می‌دهد.
کلاس Hand یک روش دیگر هم برای کپسوله‌سازی خودش دارد. این کلاس شنوندگان تغییرات حالت را تشکیل می‌دهد. از آنجایی که کلاس Hand اطلاعات حالت خود را با شنوندگان در میان می‌گذارد دلیلی ندارد شیء دیگری که به دانستن حالت Hand دارد این اطلاعات را از خود آن در خواست کند.

٤. کلاسهای Hand و HandListener یک الگوی مشاهده کننده را پیاده‌سازی می‌کنند.

۵. موجود نیست.

حل تمرین‌ها

۱. موجود نیست

۲. کلید حل این مسأله متدهایی است که تا قبل از فراخوانی کلاس PlayerListener عمکرد یکسانی دارند. راه حل این است که فراخوانی ایجاد شده با یک شیء پوشش داده شود.
به متد ()notifyListener زیر توجه کنید:

```
protected void notifyListeners (NotifyHelper)  {
   Iterator i = listeners.iterator();
   while( i.hasNext() )  {
      PlayerListener p1 = (PlayerListeners) i.next();
      helper.notifyListener( p1 );
   }
}
```

توجه داشته باشید که این متد دقیقاً مانند متد ()notifyChanged قدیمی یا ()notifyBusted است مگر در یک مورد و آن این است که به جای اینکه در کلاس PlayerListener متد مستقیماً فراخوانی شود متد ()notifyListener فراخوانی را به یک شیء از کلاس NotifyHelper واگذار می‌کند.

لیست ۱۶-۱۷

```
protected interface notifyHelper {
   public void notifyListner( PlayerListner p1 );
}
```

رابط کلاس NotifyHelper متد ()notifyListener را تعریف می‌کند. این برنامه‌نویسان هستند که تصمیم می‌گیرند که کدام متد باید در کلاس PlayerListener فراخوانی شود. با این همه، به تعریف هفت پیاده‌سازی برای کلاس NotifyHelper احتیاج خواهید داشت. یعنی برای هر متد در رابط PlayerListener یک پیاده‌سازی باید انجام شود. لیست ۱۶-۱۸ این هفت پیاده‌سازی را نشان می‌دهد.

لیست ۱۶-۱۸ پیاده‌سازیهای کلاس NotifyHelper

```
    protected class NotifyBusted implements NotifyHelper {
        public void notifyListener( PlayerListener p1 ) { p1.playerBusted( Player.this ); }
    }
    protected class NotifyBlackjack implements NotifyHelper {
        public void notifyListener( PlayerListener p1 ) { p1.playerBlackjack( Player.this ); }
    }
    protected class NotifyWon implements NotifyHelper {
        public void notifyListener( PlayerListener p1 ) { p1.playerWon( Player.this ); }
    }
    protected class NotifyLost implements NotifyHelper {
        public void notifyListener( PlayerListener p1 ) { p1.playerLost( Player.this ); }
    }
    protected class NotifyChanged implements NotifyHelper {
        public void notifyListener( PlayerListener p1 ) { p1.playerChanged( Player.this ); }
    }
    protected class NotifyStanding implements NotifyHelper {
        public void notifyListener( PlayerListener p1 ) { p1.playerStanding( Player.this ); }
    }
    protected class NotifyStandoff implements NotifyHelper {
        public void notifyListener( PlayerListener p1 ) { p1.playerStandoff( Player.this ); }
    }
}
```

حال به جای فراخوانی متدهای ()notifyChanged یا ()notifyBlackjack باید متدهای notifyListeners(new NotifyChanged()) یا notifyListeners(new NotifyBlackjack()) فرخوانده شوند.

اینکه این راه حل خوب است یا خیر یک سلیقهٔ شخصی است اما این راه حل در هر صورت متدهای notifyXXX اضافی را حذف می‌کند.

روز ۱۷

پاسخ پرسش‌ها

۱. تجرید یک متد محافظت‌شده (protected) راهکار مناسبی برای ایجاد یک پروتکل وارثت است.

۲. بعد از طراحی و تحلیل درس امروز، سلسله مراتب جدیدی برای بازیکن (Player) کشف شد.

همانگونه که نیازمندیها وجود خود را در طی بررسی موارد کاربردی نشان می‌دهند، نیاز به یک سلسله مراتب وارثت جدید هم، خود را نشان داد.

۳. با برنامه‌نویسی بدون تفکر و اندیشه سلسله مراتبی که برای مدل‌سازی مناسب یک دامنهٔ مجدد نیاز است خود را زمانی نشان می‌دهد که دامنه برای مدتی طولانی به کار گرفته شود. اگر سعی کنید که یک دامنه را بدون انجام کار روی آن مجرد کنید، در این صورت راه حل قطعاً حدسی خواهد بود.

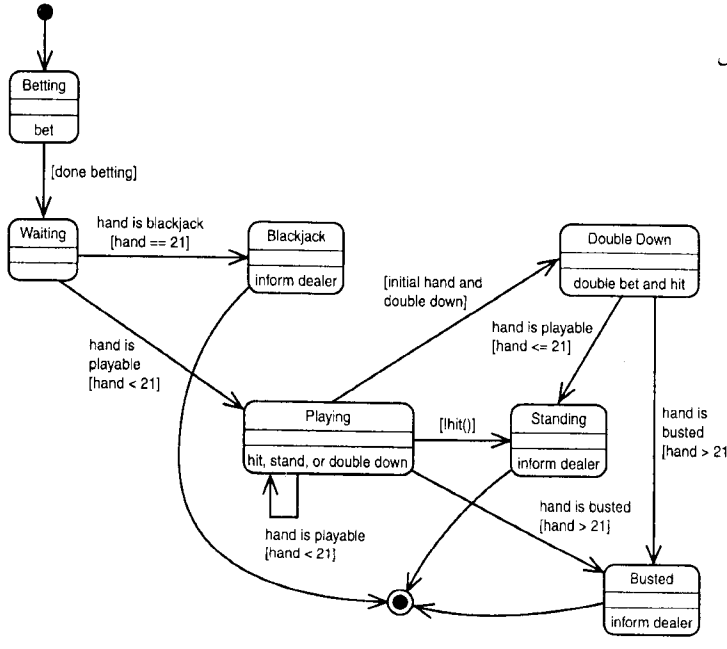
٤. با فاکتور کردن مجدد سلسله مراتب، مدلی خواهید داشت که بازی Blackjack را بهتر مدل می‌کند. همچنین کد نوشته شده آسانتر درک می‌شود. اگر وظایف اضافه‌ای برای کلاس Player پایه ایجاد شود باعث می‌شود که کد سخت و غیر قابل فهم باشد.

حل تمرین‌ها

۱. موجود نیست

۲. دوبرابر کردن (Doubling Down) تنها حالت دیگری از کلاس BettingPlayer می‌باشد. می‌دانید که این حالت باید جداگانه در نظر گرفته شود، چرا که کلاس Player باید در مقابل رخداد handplayable متفاوت رفتار کند. به جای ماندن در حالت Playing کلاس Player باید به حالت Standing منتقل شود. شکل الف-۷ حالت جدیدی برای کلاس BettingPlaye که می‌تواند شرط‌بندی را دو برابر کند، به تصویر می‌کشد.

شکل الف-۷
نمودار حالت کلاس
BettingPlaye
ارتقاء یافته



توجه کنید که نیاز است حالت Playing تغییر داده شود تا بتواند به حالت DoublingDown منتقل شود. لیست ۱۷-۶ حالتهای BettingPlayer جدید را نشان می‌دهد.

لیست ۱۷-۶ Playing و DoublingDown

```
private class DoublingDown implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        setCurrentState( getStandingState() );
        notifyStanding();
    }
    public void handBlackjack() {
        // not possible in doubling down state
    }
    public void handBusted() {
        setCurrentState( getBustedState() );
        notifyBusted();
    }
    public void execute( Dealer dealer ) {
        bank.doubleDown();
        dealer.hit( BettingPlayer.this );
        getCurrentState().execute( dealer );
    }
}
private class BetterPlaying implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        // can ignore in playing state
    }
    public void handBlackjack() {
        // not possible in playing state
    }
    public void handBusted() {
        setCurrentState( getBustedState() );
        notifyBusted();
    }
    public void execute( Dealer dealer ) {
        if( getHand().canDoubleDown() && doubleDown() ) {
            setCurrentState( getDoublingDownState() );
            getCurrentState().execute( dealer );
            return;
        }
        if( hit() ) {
            dealer.hit( BettingPlayer.this );
        } else {
            setCurrentState( getStandingState() );
            notifyStanding();
        }
        getCurrentState().execute( dealer );
        // transition
    }
}
```

همچنین نیاز است که HumanPlayer را ارتقا دهید تا بتواند شرط‌بندی دوبل را ارایه دهد.
لیست ۱۷-۷ کلاس HumanPlayer ارتقا یافته را نشان می‌دهد.

لیست ۱۷-۷ HumanPlayer.java

```
public class HumanPlayer extends BettingPlayer {

    private final static String HIT    = "H";
    private final static String STAND = "S";
    private final static String PLAY_MSG = "[H]it or [S]tay";
    private final static String BET_MSG = "Place Bet: [10] [50] or [100]";
    private final static String DD_MSG  = "Double Down? [Y]es [N]o";
    private final static String BET_10  = "10";
    private final static String BET_50  = "50";
    private final static String BET_100 = "100";
    private final static String NO  = "N";
    private final static String YES = "Y";
    private final static String DEFAULT = "invalid";

    public HumanPlayer( String name, Hand hand, Bank bank ) {
        super( name, hand, bank );
    }

    protected boolean hit() {
        while( true ) {
            Console.INSTANCE.printMessage( PLAY_MSG );
            String response = Console.INSTANCE.readInput( DEFAULT );
            if( response.equalsIgnoreCase( HIT ) ) {
                return true;
            } else if( response.equalsIgnoreCase( STAND ) ) {
                return false;
            }
            // if we get here loop until we get meaningful input
        }
    }

    protected boolean doubleDown() {
        while( true ) {
            Console.INSTANCE.printMessage( DD_MSG );
            String response = Console.INSTANCE.readInput( DEFAULT );
            if( response.equalsIgnoreCase( NO ) ) {
                return false;
            } else if( response.equalsIgnoreCase( YES ) ) {
                return true;
            }
            // if we get here loop until we get meaningful input
        }
    }

    protected void bet() {
        while( true ) {
```

لیست ۱۷.۷ (ادامه)

```
            Console.INSTANCE.printMessage( BET_MSG );
            String response = Console.INSTANCE.readInput( DEFAULT );
            if( response.equals( BET_10 ) ) {
                getBank().place10Bet();
                return;
            }
            if( response.equals( BET_50 ) ) {
                getBank().place50Bet();
                return;

            }
            if( response.equals( BET_100 ) ) {
                getBank().place100Bet();
                return;
            }
            // if we get here loop until we get meaningful input
        }
    }

}
```

روز ۱۸

پاسخ پرسش‌ها

۱. برای معرفی یک کلاس VCard می‌توان به آسانی زیرکلاسی از کلاس Card تعریف کرد. برای بازی کردن، می‌توانید زیرکلاسی از کلاس Deck تعریف کنید و سپس متد buildCards را جایگزین کنید تا زیرکلاس، کلاس‌های VCard را به جای Cardها به خدمت بگیرد.

۲. متد buildCards از کلاس Deck در اصل یک متد اختصاصی (private) می‌باشد. وقتی دریافتیم که زیرکلاس نیاز است که رفتار buildCards را جایگزین کند، این متد را به صورت محافظت شده (protected) تعریف کردیم.

حل تمرین‌ها

۱. موجود نیست

۲. طراحی و پیاده‌سازی اولیه‌ای که برای تمرین ۲ از فصل ۱۷ انجام دادید هنوز صحیح است. دو برابر کردن شرط‌بندی فقط حالت دیگری از کلاس BettingPlayer می‌باشد. در این تمرین باید با اضافه کردن حالت جدید DoublingDownn جدید به BettingPlayer کدنویسی را شروع کنید. همچنین نیاز است که مشابه تغیراتی که در کلاسهای Bank و Hand در فصل ۱۷ ایجاد کردید را دوباره در اینجا نیز انجام دهید. وقتی این تغییرات ایجاد شد نیاز است که GUIPlayer، OptionView و OptionViewController را نیز

تغییر دهید.

قبل از ایجاد تغییرات مورد لزوم در کلاسهای View و Controller تغییراتی را که لازم است در کلاسهای BettingPlayer، Hand و Bank داده شود را مرور می کنیم.

لیست ۱۸-۱۳ تغییرات ایجاد شده در کلاس Hand را مشخص می‌کند.

لیست ۱۸-۱۳ مشخصات تغییرات ایجاد شده در کلاس Hand

```
public boolean canDoubleDown() {
    return ( cards.size ) == 2 );
}
```

متد canDoubleDown جدید به کلاس BettingPlayer اجازه می‌دهد که کلاس Hand را به منظور اینکه ببیند آیا بازیکن اجازه دارد شرط‌بندی دوبرابر کند یا نه، بررسی کند. همچنین نیاز به اضافه کردن یک متد doubleDown جدید به کلاس Bank می‌باشد. لیست ۱۸-۱۴ این متد جدید را نشان می‌دهد.

لیست ۱۸-۱۴ متد doubleDown جدید

```
public void doubleDown()  {
   placeBet ( bet );
  bet = bet * 2 ;
}
```

متد doubleDown شرط جاری را دو برابر می‌کند. برای اضافه کردن شرط‌بندی دوبرابر نیاز به اضافه کردن حالت جدیدی به کلاس BettingPlayer می‌باشد. از آنجایی که کلاس BettingPlayer مجبور است که با رخداد handPlayable به ویژه زمانی که شرط‌بندی دوبل می‌کند رفتار کند نیاز به حالت جدیدی می‌باشد. معمولاً رخداد handPlayable به این معنی است که بازیکن می‌تواند اگر بخواهد دوباره کارت بکشد. وقتی حالت شرط‌بندی دوبل اتفاق می‌افتد، بازیکن باید فوراً توقف کند (اگر نباخته باشد.)

لیست ۱۸-۱۵ حالت DoubleDown جدید را نشان می‌دهد.

لیست ۱۸-۱۵ حالت DoubleDown جدید

```
private class DoublingDown implements PlayerState {

    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        setCurrentState( getStandingState() );
        notifyStanding();
    }
    public void handBlackjack() {
        // not possible in doubling down state
    }
    public void handBusted() {
        setCurrentState( getBustedState() );
        notifyBusted();
```

لیست ۱۸-۱۵ (ادامه)

```
    }
    public void execute( Dealer dealer ) {
        bank.doubleDown();
        dealer.hit( BettingPlayer.this );
        getCurrentState().execute( dealer );
    }
}
```

وقتی این حالت جدید اجرا شود باعث می‌شود که کلاس Bank شرط را دو برابر کند، از واسط بخواهد که بازیکن کارت بکشد و سپس انتقال به مرحلهٔ بعدی صورت می‌گیرد. مرحلهٔ بعدی بسته به اینکه چه رخدادی توسط کلاس Hand ارسال می‌گردد، می‌تواند توقف یا باخت باشد.

برای ایجاد حالت DoublingDown، نیاز به ایجاد تغییراتی چند در حالت Playing است. لیست ۱۸-۱۶ حالت BetterPlaying جدید را نمایش می‌دهد.

لیست ۱۸-۱۶ حالت BetterPlaying جدید

```
private class BetterPlaying implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        // can ignore in playing state
    }
    public void handBlackjack() {
        // not possible in playing state
    }
    public void handBusted() {
        setCurrentState( getBustedState() );
        notifyBusted();
    }
    public void execute( Dealer dealer ) {
        if( getHand().canDoubleDown() && doubleDown() ) {
            setCurrentState( getDoublingDownState() );
            getCurrentState().execute( dealer );
            return;
        }
        if( hit() ) {
            dealer.hit( BettingPlayer.this );
        } else {
            setCurrentState( getStandingState() );
            notifyStanding();
        }
        getCurrentState().execute( dealer );
        // transition
    }
}
```

وقتی حالت BetterPlaying اجرا می‌شود، ابتدا چک می‌کند تا ببیند آیا بازیکن می‌تواند شرط‌بندی دوبل کند. اگر چنین باشد متد doubleDown از این کلاس فراخوانی می‌شود (نیاز است که یک متد doubleDown مجرد به کلاس مذکور اضافه شود). اگر متد ذکر شده مشخص کند که بازیکن تمایل دارد شرط‌بندی دوبل کند در این صورت حالت BetterPlaying حالت جاری را به صورت DoublingDown تنظیم می‌کند و انتقال به این حالت صورت می‌گیرد.

اگر بازیکن نخواهد که شرط‌بندی دوبل کند، حالت BetterPlaying بدون تغییر باقی می‌ماند و بازی به طور معمول ادامه پیدا می‌کند. لطفاً کد اصلی را برای همهٔ تغییرات مشاهده کنید. تغییرات کوچک دیگری هم وجود دارند که، مانند اضافه کردن متد getDoubleDownState به BetterPlaying که در اضافه کردن شرط‌بندی دوبل سیستم کاملاً مؤثر و مفید است.

حال نیاز به ارتقاء کلاسهای GUIPlayer و OptionView و OptionViewController می‌باشد.

خبر خوش این است که نیازی به ایجاد تغییر در حالتهای GUIPlayer نیست. این حالتها نباید هیچ کاری انجام دهند زیرا باید تا زمانی که بازیکن دکمه‌ای را کلیک نکرده منتظر بمانند. اما متد doubleDown باید پیاده‌سازی شود. لیست ۱۸-۱۷ این متد را نمایش می‌دهد.

لیست ۱۸-۱۷ متد doubleDown

```
protected boolean doubleDown()  {
    setCurrentState( getDoublingDownState() );
    getCurrentState().execute ( dealer );
}
```

اگر کاربر تصمیم بگیرد که شرط‌بندی دوبل کند، دکمهٔ GUI می‌تواند این متد را فراخوانی کند. متد مذکور حالت جاری را به حالت شرط‌بندی دوبل تنظیم می‌کند و انتقال به این حالت صورت می‌گیرد. بقیهٔ کارها توسط حالت مشخص شده صورت می‌گیرد.

باقیماندهٔ تغییرات باید در کلاسهای OptionView و OptionViewController صورت گیرد.

اساساً نیاز به اضافه کردن یک دکمهٔ شرط‌بندی دوبل به نما می‌باشد. همچنین باید از اینکه این دکمه بوسیلهٔ کنترل‌کننده درست بعد از شرط‌بندی فعال می‌شود و به محض اینکه بازیکن آن را کلیک می‌کند غیر فعال می‌شود، اطمینان حاصل کرد.

اگر GUI را به طور کامل متوجه نمی‌شوید، نگران نباشید. آنچه مهم است این است که بفهمید که چگونه شرط‌بندی دوبل به سیستم اضافه می‌شود و ایدهٔ اصلی و پایه‌ای پشت نما و کنترل‌کننده چیست.

روز ۱۹

پاسخ پرسش‌ها

۱. سه لایه عبارتند از: نمایش، تجرید و کنترل.

۲. لایهٔ نمایش مسؤول نمایش تجرید و پاسخگویی در قبال عملکرد کاربر می‌باشد.

لایهٔ تجرید مشابه لایهٔ مدل در الگوی MVC می‌باشد. تجرید بیانگر هستهٔ سیستم است. لایهٔ کنترل مسؤول ترکیب نمایش‌های مختلف در یک نما است.

۳. می‌توان از وارثت جهت ایجاد زیرکلاسهایی از هر یک از کلاسهای سیستم که نیاز به لایهٔ نمایش دارند،استفاده کرد. در این روش، یک کلاس نمایش، مستقیماً به کلاس سیستم پیوند زده نمی‌شود و به جای آن می‌توانید کلاس نمایش را به زیرکلاسی که ایجاد کرده‌اید اضافه کنید.
با استفاده از وراثت به این روش، نماهای مختلفی از یک سیستم یکسان فراهم می‌آورید. هر زمان که به نمای متفاوتی نیاز داشتید به آسانی زیرکلاسی از کلاسهایی که نیاز دارید نمایش دهید، ایجاد کنید و در آنها یک لایه نمایش جدید ایجاد کنید. وقتی نیاز است که همهٔ کلاسها را در کنار یکدیگر به عنوان یک برنامه قرار دهید، دقت کنید که زیرکلاسهای جدید را ساخته باشید.

٤. استفادهٔ از الگوی PAC روی یک سیستم پایدار که دارای نیازمندیهای به خوبی تعریف شدهٔ رابط می‌باشد بهترین راهکار است. گاهی اوقات وارثت برای طراحی‌های پیچیده مناسب نیست.
زمانی که وراثت با شکست مواجه شود باید لایهٔ نمایش را مستقیماً به کلاس سیستم پیوند زد. چنین رخدادی باعث می‌شود که سرویس دهی نماهای مختلف مشکل شود.

۵. همانگونه که مشاهده کردید می‌توان از هر دو رابط کاربری استفاده کرد، زیرا چهارچوب مشاهده گر بدون تغییر مانده است. استفاده از PAC، مشکلی برای اتفاده از چهارچوب ایجاد نمی‌کند. در واقع هم BettingView و DealerView از مکانیزم PlayerListener برای همپایی با تغییرات بازیکن و واسط استفاده می‌کنند.

۶. استفاده از الگوی عامل برای حصول اطمینان از این موضوع بود که نمونه‌های مناسبی از کلاس با یکدیگر در حال تعامل هستند. به ویژه دقت کنید که از کلاس VDeck در هر جایی که می‌خواهید کلاسهایی که در خود یک لایه نمایش ایجاد کرده‌اند را به کار برید، استفاده کنید.

حل تمرین‌ها

۱. موجود نیست

۲. طراحی و پیاده‌سازی اولیه‌ای که برای تمرین ۲ از فصل ۱۷ انجام دادید همچنان معتبر است. DoubligDown تنها حالت دیگری از کلاس BettingPlayer می‌باشد. در این تمرین باید با اضافه کردن DoublingDown جدید به BetlingPlayer کدنویسی را آغاز کنید. همچنین نیاز است همان تغییراتی که در کلاسهای Bank و Hand در فصل ۱۷ ایجاد کردید را در اینجا هم ایجاد کنید. وقتی این تغییرات ایجاد شد باید کلاسهای GUIPlayer و کلاسهای لایهٔ نمایش هم به روز شوند.
قبل از ایجاد تغییرات لازم در کلاسهای GUI تغییراتی که لازم است در کلاسهای BettingPlayer، Hand و Bank داده شود را مرور می‌کنیم.
لیست ۱۹-۸ تغییرات ایجاد شده در کلاس Hand را مشخص می‌کند.

لیست ۱۹-۸ مشخصات تغییرات ایجاد شده در کلاس Hand

```
public boolean canDoubleDown() {
   return ( cards.size() == 2 );
}
```

متد canDoubleDown جدید به کلاس BettingPlayer اجازه می‌دهد که کلاس Hand را به منظور اینکه ببیند آیا بازیکن اجازه دارد که شرط‌بندی دوبل کند، بررسی نماید. همچنین نیاز به اضافه کردن یک متد doubleDown جدید به کلاس Bank می‌باشد. لیست ۱۹-۹ این متد جدید را نشان می‌دهد.

لیست ۱۹-۹ متد doubleDown جدید

```
public void doubleDown() {
    placeBet( bet ) ;
    bet = bet * 2 ;
}
```

متد doubleDown شرط جاری را دو برابر می‌کند. برای اضافه کردن شرط‌بندی دوبل نیاز به اضافه کردن حالت جدیدی به کلاس BettingPlayer می‌باشد. به دلیل اینکه کلاس BettingPlayer مجبور است با رخداد handPlayable به ویژه زمانی که شرط‌بندی دوبل می‌کند، برخورد داشته باشد نیاز به حالت جدیدی می‌باشد. معمولاً رخداد handPlayable به این معنی است که بازیکن می‌تواند اگر بخواهد دوباره کارت بکشد. وقتی حالت شرط‌بندی دوبل اتفاق می‌افتد، بازیکن باید فوراً توقف کند (اگر نباخته باشد).

لیست ۱۹-۱۰ حالت DoubleDown جدید را نشان می‌دهد.

لیست ۱۹-۱۰ حالت DoubleDown جدید

```
    private class DoublingDown implements PlayerState {
        public void handChanged() {
            notifyChanged();
        }
        public void handPlayable() {
            setCurrentState( getStandingState() );
            notifyStanding();
        }
        public void handBlackjack() {
            // not possible in doubling down state
        }
        public void handBusted() {
            setCurrentState( getBustedState() );
            notifyBusted();
        }
        public void execute( Dealer dealer ) {
            bank.doubleDown();
            dealer.hit( BettingPlayer.this );
            getCurrentState().execute( dealer );
        }
    }
```

وقتی این حالت جدید اجرا شود باعث می‌شود که کلاس Bank شرط را دو برابر کند. از واسط می‌خواهد که بازیکن کارت بکشد و سپس انتقال به مرحلهٔ بعدی صورت می‌گیرد. مرحلهٔ بعدی بسته به اینکه چه

رخدادی توسط کلاس Hand ارسال می‌گردد، می‌تواند توقف یا باخت باشد.
برای ایجاد حالت DoublingDown، نیاز به ایجاد تغییراتی در حالت Playing است. لیست ۱۹-۱۱ حالت BetterPlaying جدید را نمایش می‌دهد.

لیست ۱۹-۱۱ حالت Playing جدید

```
private class BetterPlaying implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        // can ignore in playing state
    }
    public void handBlackjack() {
        // not possible in playing state
    }
    public void handBusted() {
        setCurrentState( getBustedState() );
        notifyBusted();
    }
    public void execute( Dealer dealer ) {
        if( getHand().canDoubleDown() && doubleDown() ) {
            setCurrentState( getDoublingDownState() );
            getCurrentState().execute( dealer );
            return;
        }
        if( hit() ) {
            dealer.hit( BettingPlayer.this );
        } else {
            setCurrentState( getStandingState() );
            notifyStanding();
        }
        getCurrentState().execute( dealer );
        // transition
    }
}
```

وقتی حالت BetterPlaying اجرا می‌شود، ابتدا بررسی می‌کند تا ببیند که آیا بازیکن می‌تواند شرط‌بندی دوبل کند یا خیر. اگر چنین باشد متد doubleDown از این کلاس فراخوانی می‌شود (نیاز است که یک متد doubleDown مجرد به کلاس مذکور اضافه شود). اگر متد ذکر شده مشخص کند که بازیکن دوست دارد شرط‌بندی دوبل کند، در این صورت حالت BetterPlaying حالت جاری را به صورت DoublingDown تنظیم می‌کند و انتقال به این حالت صورت می‌گیرد.
اگر بازیکن نخواهد که شرط‌بندی دوبل کند، حالت BetterPlaying تغییر نمی‌کند، و بازی به طور معمول ادامه می‌یابد. لطفاً کد اصلی را برای همهٔ تغییرات مشاهده کنید. تغییرات کوچک دیگری هم وجود دارند، مانند اضافه کردن متد getDoubleDownState به BettingPlayer .

اکنون که شرط‌بندی دوبل به صورت کامل به سیستم اضافه شده است، نیاز است GUIPlayer و کلاسهای جدید نمایش آن به روز شوند تا بتوان شرط‌بندی کلاسهای دوبل را به سیستم اضافه کرد.
خبر خوش این است که نیازی به ایجاد تغییر در حالتهای GUIPlayer نیست. این حالتها نباید هیچکاری انجام دهند زیرا باید تا زمانی که بازیکن کلیدی را فشار نداده منتظر بمانند. اما پیاده‌سازی متد doubleDown باید انجام شود. لیست ۱۹-۱۲ این متد را نمایش می‌دهد.

لیست ۱۹-۱۲ متد doubleDown

```
protected boolean doubleDown () {
    setCurrentState().getDoublingDownState() );
    getCurrentState().execute ( dealer );
    return true ;
}
```

اگر کاربر تصمیم بگیرد که شرط‌بندی دوبل کند GUI می‌تواند این متد را فراخوانی کند. متد مذکور حالت جاری را به حالت شرط‌بندی دوبل تنظیم می‌کند و انتقال به این حالت صورت می‌گیرد. بقیهٔ کارها توسط حالت مشخص شده صورت می‌گیرد. باقیماندهٔ تغییرات باید در کلاسهای نما نظیر GUIView صورت گیرد.
اساساً نیاز به اضافه کردن یک دکمهٔ شرط‌بندی دوبل می‌باشد همچنین باید از اینکه این دکمه درست بعد از شرط‌بندی فعال می‌شود و به محض اینکه بازیکن آن را فشار می‌دهد غیر فعال می‌شود، اطمینان حاصل کرد.
اگر کد GUI را به طور کامل متوجه نمی‌شوید نگران نباشید. آنچه مهم است این است که بفهمید که چگونه شرط‌بندی دوبل به سیستم اضافه می‌شود.

روز ۲۰

پاسخ پرسش‌ها

۱. از طریق ارتباطهای جانشینی و چندشکلی‌بودن می‌توان هر زیرکلاس از کلاس BettingPlayer را ایجاد کرد و آن را به بازی اضافه کرد.
بازی فرق میان یک بازیکن انسان و یک بازیکن غیر انسان اتوماتیک را نمی‌داند. بنابراین می‌توان بازی را بر اساس بازیکنان غیر انسان تنظیم کرد. با پیاده‌سازی متد hit در زیرکلاسهای کلاس Player، این زیرکلاسها می‌توانند بدون دخالت انسان بازی کنند.
۲. هرگز نباید استراتژی OneHitPlayer را دنبال کرد.

حل تمرین‌ها

۱. موجود نیست

۲. راه حلهای مختلفی ممکن است پیشنهاد شوند. لیست ۲۰-۸ و ۲۰-۹ به ترتیب پیاده‌سازی کلاسهای KnowledgeablePlayer و OptimalPlayer را نشان می‌دهد.

لیست ۲۰-۸ KnowledgeablePlayer.java

```
public class KnowledgeablePlayer extends BettingPlayer {

    public KnowledgeablePlayer(String name,Hand hand,Bank bank) {
        super( name, hand, bank );
    }

    public boolean hit( Dealer dealer ) {

        int total = getHand().total();
        Card card = dealer.getUpCard();

        // never hit, no matter what, if total > 15
        if( total > 15 ) {
            return false;
        }

        // always hit for 11 and less
        if( total <= 11 ) {
            return true;
        }

        // this leaves 11, 12, 13, 14
        // base decision on dealer

        if( card.getRank().getRank() > 7 ) {
            return true;
        }

        return false;

    }

    public void bet() {
        getBank().place10Bet();
    }

}
```

لیست ۲۰-۹ OptimalPlayer.java

```
public class OptimalPlayer extends BettingPlayer {

    public OptimalPlayer( String name, Hand hand, Bank bank ) {
        super( name, hand, bank );
    }
```

لیست ۲۰ـ۹ (ادامه)

```
    public boolean hit( Dealer dealer ) {

        int total = getHand().total();
        Card card = dealer.getUpCard();

        if( total >= 17 ) {
            return false;
        }

        if( total == 16 ) {
            if( card.getRank() == Rank.SEVEN ||
                card.getRank() == Rank.EIGHT ||
                card.getRank() == Rank.NINE ) {
                return true;
            } else {
                return false;
            }
        }
        if( total == 13 || total == 14 || total == 15 ) {
            if( card.getRank() == Rank.TWO   ||
                card.getRank() == Rank.THREE ||
                card.getRank() == Rank.FOUR  ||
                card.getRank() == Rank.FIVE  ||
                card.getRank() == Rank.SIX ) {
                return false;
            } else {
                return true;
            }
        }
        if( total == 12 ) {
            if( card.getRank() == Rank.FOUR ||
                card.getRank() == Rank.FIVE ||
                card.getRank() == Rank.SIX ) {
                return false;
            } else {
                return true;
            }
        }
        return true;
    }

    public void bet() {
        getBank().place10Bet();
    }

}
```

حال، چگونه این بازیکنان پشت سر هم قرار می‌گیرند؟

در آزمایش من، هر دو کلاس KnowledgeablePlayer و OptimalPlayer بهتر از کلاس OptimalPlayer که

قبلاً نشان داده شد کار می‌کنند. اگر این دو کلاس را با هم مقایسه کنیم OptimalPlayer بهتر کار می‌کند. با این حال هر دوی این کلاسها پول از دست می‌دهند، اگر چه به کندی.

۳. راه حلهای پیشنهادی متفاوتند. لیست ۲۰-۱۰ و ۲۰-۱۱ پیاده‌سازی کلاسهای KnowledgeablePlayer و OptimalPlayer را به ترتیب نشان می‌دهد.

لیست ۲۰-۱۰ KnowledgeablePlayer.java

```
public class KnowledgeablePlayer extends BettingPlayer {

    public KnowledgeablePlayer(String name,Hand hand,Bank bank) {
        super( name, hand, bank );
    }
    public boolean hit( Dealer dealer ) {
        int total = getHand().total();
        if(total == 10 || total == 11) {
          retun true;
        }
}
public boolean hit (Dealer  d) {
int total , no matter what , if total > 15
if (total < = 11) {
    return true ;
}
        // never hit, no matter what, if total > 15
        if( total > 15 ) {
            return false;
        }

        // this leaves 11, 12, 13, 14
        // base decision on dealer
            return true;
        if (c. getRank () . getRank() > 7 ) {
            return true;
        }

        return false;

    }

    public void bet() {
        getBank().place10Bet();
    }

}
```

لیست ۲۰.۱۱ OptimalPlayer.java

```
    public class OptimalPlayer extends BettingPlayer {

        public OptimalPlayer( String name, Hand hand, Bank bank ) {
            super( name, hand, bank );
```

```
    }

    public boolean doubleDown ( Dealer d ) {
        int total = getHand().total();
        Card c = d.getUpCard();
        if( total >= 17 ) {
            return false;
        }
        if( total == 16 ) {
            if( c.getRank() . getRank()   1=Rnank.TEN . getRank () &&
                return true;
            }
                return false;
            }
        if( total == 9 ) {
            if( c.getRank() == Rank.TWO   ||
                card.getRank() == Rank.THREE ||
                card.getRank() == Rank.FOUR  ||
                card.getRank() == Rank.FIVE  ||
                card.getRank() == Rank.SIX ) {
            }
                return false;
            }
            return false;
        }
public boolean hit (Dealer d)  {
    int total = getHnd (). total();
    Card c = d. getUpCard ();
    if(total > = 17) {
      return false;
    }
    if ( total == 16) {
        if( c.getRank() == Rank . SEVEN ||
          c.getRank() == Rank.EIGHT ||
          c.getRank() == Rank.NINE ) {
          return true;
        } else {
        return false;
     }
}
if (total == 13 || total == 14 || total == 15)  {
    if { c.getRank() == Rank . TWO   ||
         c.getRank () == Rank .THREE  ||
         c.getRank() == Rank . FOUR   ||
         c.getRank() == Rank .FIVE    ||
         c.getRank() == Rank .SIX)  {
    } else {
        return true;
    }
}
if( total == 12) {
```

لیست ۲۰-۱۱ (ادامه)

```
        if( c.GetRank() == Rank. FOUR ||
            c.getRank() == Rank .FIVE ||
            c.getRank() == Rank .SIX ) {
            return false ;
        }else {
          return true ;
        }
        return true;
      }
      public void bet () {
        getBank (). place10Bwt();
      }
    }
```

در آزمایش من، هر دو کلاس KnowledgeablePlayer و OptimalPlayer بهتر از نسخه‌های ارایه شده در تمرین ۲ عمل می‌کنند.

باز هم هر دوی این کلاسها پول از دست می‌دهند و البته باز هم به کندی.

روز ۲۱

پاسخ پرسش‌ها

۱. برای حل مشکل فراخوانی متد بازگشتی باید از تکنیک چندرشته‌ای‌کردن (threading) استفاده کرد. وقتی thread.Start فراخوانی می‌شود بلافاصله برمی‌گردد و این بر خلاف یک متد معمولی است و به خاطر این برگشت فوری، متد در پشته نمی‌ماند.

حل تمرین‌ها

۱. موجود نیست

۲. پاسخ‌ها بستگی به علایق شخصی خواهد داشت. می‌توانید در اینترنت مراجعی بسیار عالی از منابعی که می‌توانید مطالعات خود را بر اساس آنها ادامه دهید، بیابید. در ضمن کتب بسیار خوبی در زمینه‌های مختلف روش شیءگرا چاپ شده است که می‌توانید به آنها مراجعه کنید.

آموزش

برنامه‌نویسی شیءگرا

در ۲۱ روز

برای فراگیری:

- طراحی و ساخت پروژه‌های شیءگرا با جاوا
- ساخت و توسعه پروژه‌های نرم‌افزاری به شیوه‌ای کارآمد
- اعمال اصول و قواعد طراحی شیءگرا در پروژه‌های واقعی
- یادگیری نحوه جمع‌آوری و پیاده سازی نیازمندی های نرم افزار برای پروژه‌های OOP
- یادگیری اصطلاحات و روابط مورد نیاز برای کارکرد برنامه‌نویسی شیءگرا
- طرحریزی برای استفاده مجدد و انعطاف‌پذیری نرم افزار پیش از هرگونه کدنویسی
- یاد گیری اعمال تفکر شیءگرا در ابتدای پروژه
- اعمال استراتژی های مناسب در طراحی و مدیریت پروژه‌های نرم‌افزاری
- و ...

این کتاب را بخوانید

شابک: ۹-۰۰۶-۴۱۰-۹۶۴

9 769644 100064

نص

تهران میدان انقلاب خ اردیبهشت، بن بست مبین شماره ۲۳۷