Wzorce projektowe. Analiza kodu sposobem na ich poznanie

Autor: Allen Holub
ISBN: 83-7361-948-8
Tytuł oryginału: Holub on Patterns: Learning Design Patterns by Looking at Code
Format: B5, stron: 464

Opanuj zasady stosowania wzorców projektowych na praktycznych przykładach

- Dowiedz się, czym są wzorce projektowe
- Zaimplementuj wzorce we własnych programach
- Poznaj rodzaje wzorców projektowych

Wzorce projektowe to zapisane w sposób formalny sposoby rozwiązywania najczęstszych problemów, z jakimi borykają się twórcy oprogramowania stosujący języki obiektowe. Najczęściej stosowane wzorce zostały skatalogowane i przedstawione w postaci diagramów UML, jednak do poprawnego ich wykorzystywania niezbędna jest wiedza praktyczna. Przystępując do implementacji wzorca projektowego, należy poznać zakres jego zastosowania. Taką wiedzę najlepiej zdobywa się, analizując przykłady kodów źródłowych.

Dzięki książce „Wzorce projektowe. Analiza kodu sposobem na ich poznanie" poznasz wzorce w taki właśnie sposób — badając programy, w których je zastosowano. Każdy z omawianych w książce wzorców zaprezentowany jest w oparciu o dwie implementacje szczegółowo wyjaśniające zasadę jego działania. Dzięki takim opisom wzorców opanujesz tę technologię znacznie szybciej niż w przypadku nauki teoretycznych podstaw oraz prób ich samodzielnego wdrażania we własnych aplikacjach. Unikniesz typowych błędów i dowiesz się, jak prawidłowo wykorzystywać każdy z wzorców.

- Zastosowanie wzorców projektowych
- Klasyfikacja wzorców
- Podstawowe pojęcia z dziedziny obiektowości
- Interfejsy i wzorce konstrukcyjne
- Implementacja wzorców obserwatora i fasady
- Wykorzystanie wzorców projektowych w aplikacjach bazodanowych

Książka zawiera również zestawienie najczęściej wykorzystywanych wzorców projektowych wraz z opisem ich zastosowań.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 1.

Wstęp: programowanie obiektowe i wzorce projektowe

W zwykłych okolicznościach tego typu książka powinna się rozpoczynać cytatem z Christophera Alexandra, architekta (budynków, nie oprogramowania), który jako pierwszy wprowadził termin wzorca projektowego. Odkryłem, że chociaż Alexander jest wspaniałym człowiekiem i pisze naprawdę świetne książki, jego dzieła mogą być zrozumiałe nie dla wszystkich, zatem pozwolę sobie w tym miejscu pominąć „obowiązkowy" cytat. Warto jednak pamiętać, że tezy prezentowane przez Alexandra stanowią podstawowe źródło współczesnych koncepcji wzorców projektowych.

Podobnie, prawdziwym przełomem w kwestii zastosowań wzorców projektowych w świecie oprogramowania była książka *Design Patterns: Elements of Reusable Object-Oriented Software* autorstwa Gammy, Helma, Johnsona i Vlissidesa (Addison-Wesley, 1995). (Czterej autorzy tej książki są przez większość projektantów dowcipnie nazywani *Bandą Czworga*.) Moja książka nie mogłaby powstać, gdyby nie powstała *książka Bandy Czworga*, zatem ja sam (podobnie jak chyba wszyscy programiści stosujący techniki obiektowe) jestem winien autorom tej książki ogromne podziękowania. Tak czy inaczej, wspomniana przeze mnie książka jest akademicką prezentacją wzorców, która dla większości początkujących programistów z natury rzeczy będzie po prostu niezrozumiała. Postanowiłem podjąć ryzyko zatracenia tej akademickiej precyzji i stworzyć coś bardziej przystępnego.

Wzorce kontra cechy języków programowania

Nasze rozważania powinniśmy rozpocząć od zgłębienia samego pojęcia *wzorca* przez analizę prostych własności języków programowania. Wiele wzorców projektowych jest wykorzystywanych tak powszechnie, jest tak głęboko zakorzenionych w umysłach tylu programistów, że nie są już traktowane jak wzorce, tylko jak cechy poszczególnych języków programowania. Wzorce te nie są uważane za coś specjalnego — stanowią po prostu typowe „sposoby rozwiązywania określonych problemów". Niektórzy odróżniają wzorce od cech językowych wyłącznie na podstawie ich zastosowania (przykładowo, wzorzec, w przeciwieństwie do cech językowych, jest reprezentowany w sposób formalny). Dla mnie taki podział nie jest jednak wiarygodny. Cecha (własność) językowa to po prostu wzorzec, którego stosowanie nabrało powszechnego wymiaru.

Doskonałym przykładem ewolucji wzorców do postaci cech językowych jest model wyprowadzania struktur danych. We wczesnych latach osiemdziesiątych ubiegłego wieku, kiedy królował język programowania C, wyprowadzanie struktur danych było typowym wzorcem projektowym. W języku C istniało wówczas wiele przykładów relacji „rozszerzania" jednych struktur w inne, bardziej szczegółowe. Przykładowo, standardowa implementacja funkcji `malloc()` wykorzystuje nagłówek (klasę bazową), który jest rozszerzany do postaci innej struktury (klasy potomnej), a ta z kolei dziedziczy metodę `free()` z klasy bazowej.

Także funkcje abstrakcyjne należały do wzorca wyprowadzania. W kodzie języka programowania C często można się było natknąć na przekazywanie tablic wskaźników do funkcji, które były inicjalizowane w różny sposób dla różnych „klas". W języku C++ w taki sam sposób zaimplementowano zarówno metody abstrakcyjne, jak i mechanizm dziedziczenia interfejsów (które w świecie języka C nie zostały nawet nazwane).

Wyprowadzanie struktur jako takie nie było wbudowane w język programowania C i większość programistów tego języka nie stosowała obiektów, zatem wyprowadzania struktur z pewnością nie można uznać za cechę języka C — był to więc wzorzec. Było to coś, co można było bez trudu znaleźć w wielu programach, które z rozmaitych względów musiały rozwiązywać podobne problemy, nie było to jednak działanie naturalne z perspektywy przeciętnego programisty języka C.

Mechanizmy wyprowadzania (dziedziczenia) i interfejsów są obecnie wbudowanymi elementami języków programowania, można więc przyjąć, że stały się cechami tych języków.

Czym właściwie jest wzorzec projektowy?

Po pierwsze, wzorce projektowe są, co bardzo ważne, odkrywane, nie są więc wynalazkami. Kiedy Christopher Alexander analizował konstrukcje wielu udanych budynków, za każdym razem skupiał się tylko na jednym ich aspekcie (np. starał się ocenić, co decyduje o „przytulności" pomieszczeń), co w dłuższej perspektywie *prowadziło do ciekawych odkryć właśnie w kwestii odpowiednich wzorców*. Udane, „przytulne" pomieszczenia rozwiązują określone klasy problemów (np. związane z oświetleniem) w bardzo podobny sposób. Podobnie, kiedy analizujesz kod wielu programów napisanych przez różnych programistów, i kiedy skupiasz się na konkretnym problemie implementacyjnym rozwiązywanym przez te programy z konieczności (np. problem izolacji podsystemów), również możesz dostrzec pewne wzorce. Odkrywasz wówczas, że wielu programistów niezależnie od siebie stosuje podobne techniki rozwiązywania podobnych problemów. Kiedy już zdasz sobie sprawę z istnienia tych technik, będziesz widział odpowiednie wzorce gdziekolwiek spojrzysz. Za wzorzec można jednak uznać tylko takie rozwiązanie, które zostało zastosowane w wielu programach opracowywanych zupełnie niezależnie od siebie. Twierdzenie „Wynaleźliśmy wzorzec projektowy, który…" jest więc niezawodnie oznaką autorów, którzy nie rozumieją istoty wzorców. Istnieje oczywiście możliwość opracowania genialnego projektu, ale nie będzie to wzorzec projektowy dopóty, dopóki nie zostanie zastosowany przez wielu programistów pracujących całkowicie niezależnie nie tylko od siebie, ale też od autorów projektu. (Nie można oczywiście wykluczyć sytuacji, w której wynaleziony „wzorzec" staje się wzorcem rzeczywistym z racji jego powszechnego stosowania).

Wzorzec projektowy jest więc ogólną techniką wykorzystywaną do rozwiązywania pewnej klasy powiązanych ze sobą problemów. Nigdy nie jest to konkretne rozwiązanie jakiegoś problemu. Prawdopodobnie każdy architekt, któremu udało się stworzyć „przytulne" pomieszczenie zaprojektował jego oświetlenie w nieco inny sposób — to samo dotyczy programistów, z których każdy trochę inaczej implementuje swoje rozwiązania. Wzorzec jest ogólną strukturą rozwiązania (jeśli chcesz, możesz to nazywać *metarozwiązaniem*), ale z pewnością nie jest rozwiązaniem jako takim.

Dobrą analogią jest świat muzyki. Pojęcie „muzyki klasycznej" można traktować jak pewien wzorzec kompozytorski. Identyfikacja muzyki pasującej do wzorca „muzyki klasycznej" jest możliwa, ponieważ analizowane utwory po prostu brzmią jak muzyka klasyczna. Nie oznacza to jednak, że poszczególne dzieła tej muzyki są identyczne.

Skoro natura wzorców jest tak ogólna, kopiowanie wzorców projektowych z kodu jednego programu i wklejanie ich do kodu innych programów jest niemożliwe (choć w niektórych przypadkach *możesz* ponownie wykorzystać określone rozwiązanie pod warunkiem, że bieżący kontekst jest podobny do oryginalnego kontekstu użycia tego rozwiązania). Ta fundamentalna zasada jest źródłem wielu nieporozumień wśród osób, które funkcjonują w świecie wzorców projektowych od niedawna. Komentarze, z którymi miałem okazję się zapoznać, przeglądając strony internetowe wskazują, że wielu programistów uważa, że jeśli jakaś książka nie zawiera tych samych przykładów co *książka Bandy Czworga*, dowodzi to braku kompetencji autora tej książki. Moim zdaniem, taka postawa dowodzi

raczej braku zrozumienia koncepcji wzorców projektowych po stronie autorów tego rodzaju opinii: mylą fragment kodu demonstrujący jakiś wzorzec z samym wzorcem. Mając to na uwadze, spróbuję zaprezentować różne przykłady dla każdego z omawianych wzorców, abyś mógł się przekonać, jak odmienne mogą być konkretne implementacje oparte na tym samym wzorcu — nie będę też wykorzystywał przykładów Bandy Czworga, chyba że będą dotyczyły rzeczywistych problemów związanych z programowaniem (ten warunek spełnia tylko niewielka ich część).

Analizę wzorców projektowych dodatkowo komplikuje fakt, że rzeczywiste obiekty i klasy należące do poszczególnych wzorców niemal zawsze należą też do innych wzorców. Kiedy przyjrzysz się interesującemu Cię rozwiązaniu pod określonym kątem, dostrzeżesz jeden wzorzec; jednak kiedy spojrzysz na to samo rozwiązanie pod innym kątem, zobaczysz zupełnie inny wzorzec. Badania wzorców bywają jeszcze trudniejsze w sytuacjach, gdy wiele implementacji wzorców opiera się na identycznych strukturach statycznych. Kiedy analizujesz przedstawione w *książce Bandy Czworga* diagramy struktury statycznej (zapisane w języku UML), wszystkie wyglądają bardzo podobnie — widać tam interfejs, klasę klienta i klasę implementacji. Różnic pomiędzy wzorcami należy szukać w dynamicznym zachowaniu systemu i w celach realizowanych przez programistę, a nigdy w klasach czy w sposobie ich łączenia.

Spróbuję zilustrować wymienione problemy, posługując się przykładem ze świata tradycyjnej architektury budynków — skupię się na dwóch dziedzinach: wentylacji i oświetleniu.

Jeśli chodzi o wentylację, nie chcę, by pokój robił wrażenie „dusznego". Jeśli przeanalizujemy rozwiązanie tego problemu w wielu naprawdę komfortowych pomieszczeniach, dostrzeżemy w nich jeden wzorzec, który będę nazywał *wentylacją przecinającą*. Pomieszczenia należące do tego wzorca mają źródło i ujście powietrza ustawione naprzeciwko siebie na dwóch przeciwległych ścianach, w dodatku na wysokości okien. Powietrze dostaje się do pomieszczenia poprzez źródło, po czym przechodzi przez całe to pomieszczenie i opuszcza je przez ujście. Kiedy już dysponowałem zidentyfikowanym (i nazwanym) wzorcem, stworzyłem krótki opis — nazywany przez Bandę Czworga *intencją* — który podsumowuje zarówno ogólny problem, jak i rozwiązanie wynikające z tego wzorca. W przypadku wentylacji przecinającej moim celem było „wyeliminowanie poczucia duszności i zapewnienie wyższego komfortu przez umożliwienie bezpośredniego przepływu powietrza w poziomie, na wysokości okien". Mechanizmem architektonicznym, który ten cel realizuje jest logiczna reifikacja (wyjaśnię to słowo za chwilę) wzorca. (Banda Czworga używa w tym kontekście słowa *intencja*, co jest dosyć dziwne. Sam nie będę się posługiwał tym słowem zbyt często, ponieważ w moim odczuciu znacznie lepszym słowem jest *cel*).

Reifikacja to brzydkie, ale dość przydatne słowo w tym kontekście. Słowo to nie jest zbyt często stosowane w literaturze. Reifikacja znaczy dosłownie „materializację, nadanie czemuś określonej treści i formy". Reifikacja koncepcji jest więc jej konkretną realizacją, a dla pojedynczej koncepcji mogą istnieć miliony możliwych reifikacji. Używam tego słowa (zamiast któregoś z bardziej popularnych wyrażeń), aby podkreślić czym nie jest wzorzec. Przykładowo, wzorzec nie jest „egzemplarzem" czy „instancją". Każdy egzemplarz klasy jest identyczny (przynajmniej w sensie struktury) z wszystkimi pozostałymi egzemplarzami tej samej klasy. Z pewnością nie jest to cecha wzorca

projektowego. Podobnie, reifikacja nie jest „implementacją" wzorca — reifikacja wzorca ma postać projektu, nigdy kodu, zaś dla danego projektu istnieje wiele możliwych (prawidłowych) implementacji.

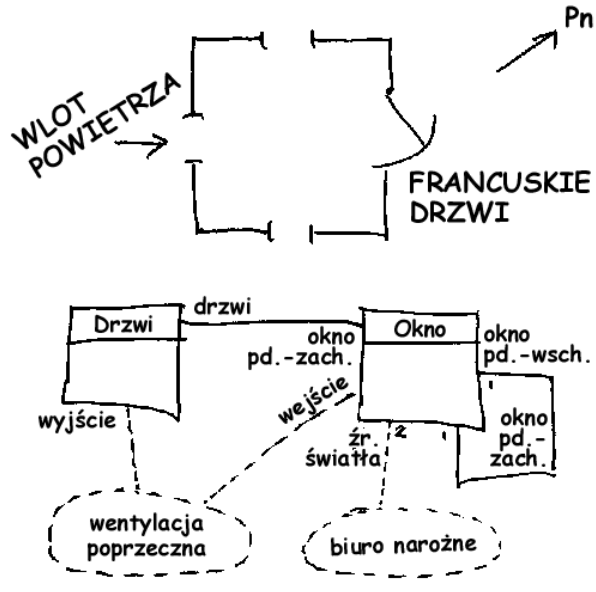
Jakie więc są reifikacje wzorca wentylacji przecinającej? W pomieszczeniu mogą istnieć okna usytuowane naprzeciwko siebie, okno na wprost drzwi, dwoje drzwi naprzeciw siebie, okno na wprost wentylatora „ujemnego" (wyciągającego powietrze), dwa wentylatory (wejściowy i wyjściowy) zainstalowane na przeciwległych ścianach lub wielki miech (obsługiwany przez skaczącego na nim orangutana) zainstalowany przy jednej ze ścian i skierowany w stronę przeciwległej ściany z wybitym otworem. Tak naprawdę w ogóle nie są nam potrzebne ściany — pomieszczenie bez dwóch przeciwległych ścian doskonale pasuje do tego wzorca. Istnieje nieskończenie wiele reifikacji tego wzorca.

Ponieważ reguły konstruowania reifikacji wzorca i tak są bardzo elastyczne, nie możesz wybierać tylko tych atrybutów, które uważasz za przydatne. Przykładowo, samo posiadanie wlotów i ujść powietrza nie jest warunkiem wystarczającym, jeśli nie są spełnione wymagania odnośnie do ich wysokości i usytuowania naprzeciwko siebie. Umieszczenie wlotu i wylotu powietrza np. na suficie nie będzie stanowiło prawidłowej reifikacji tego wzorca (co może potwierdzić każdy, kto pracuje w budynku z wielkimi przestrzeniami biurowymi z podwieszanymi wentylatorami).

Podsumowując, celem wentylacji poprzecznej jest „wyeliminowanie uczucia duszności i zapewnienie większego komfortu przez umożliwienie poziomego przepływu powietrza w poprzek pokoju, na wysokości okien". *Uczestnicy* tego wzorca (mogą to być okna, drzwi czy nawet orangutany) mają przypisane takie *role* jak źródło i ujście powietrza.

Przejdźmy teraz do problemu oświetlenia. Po przeanalizowaniu wielu pomieszczeń stwierdziłem, że najbardziej urokliwe są pokoje z oknami na dwóch przylegających ścianach. Dlatego też tak dużym powodzeniem cieszą się narożne gabinety — wielokierunkowe naturalne źródła światła czynią pomieszczenie znacznie przyjemniejszym. Zdecydowałem, że nazwę ten wzorzec projektowy *biurem narożnym*; cel wzorca zdefiniowałem w następujący sposób: „zapewnienie większego komfortu przez zastosowanie dwóch źródeł naturalnego światła na dwóch przylegających ścianach". Także w tym przypadku reifikacji wzorca jest nieskończenie wiele: okna na dwóch ścianach, okno na jednej ścianie i francuskie drzwi na drugiej, francuskie drzwi na dwóch ścianach itp. Mógłbyś teraz powiedzieć, że także rozwiązanie z oknem na jednej ścianie i lustrem na ścianie przylegającej jest reifikacją tego wzorca, ponieważ lustro odbijające naturalne światło można interpretować jak jego źródło. Cóż, gdybym był Billem Gatesem, mógłbym zaliczyć do tego wzorca także pomieszczenia z jednym oknem i zawieszonym na ścianie obok telewizorem plazmowym pokazującym obraz za tą ścianą, ale z pewnością nie byłaby to prawidłowa reifikacja tego wzorca, ponieważ telewizor plazmowy nie jest i nigdy nie będzie „naturalnym źródłem światła". Istnieje oczywiście mnóstwo sposobów implementacji wzorców samych okien i francuskich drzwi.

Przeanalizujmy teraz konkretny projekt — plan budynku. Na rysunku 1.1 przedstawiono reifikację wzorców projektowych wentylacji poprzecznej i biura narożnego (w ramach pojedynczego projektu). Widać tam zarówno diagram architektoniczny, jak i równoważny diagram UML. Wzorce są identyfikowane za pomocą symbolu *współpracy* języka UML w wersji 1.5. Nazwa wzorca jest zapisywana wewnątrz elipsy rysowanej przerywaną



Rysunek 1.1. *Połączona reifikacja wentylacji poprzecznej i biura narożnego*

linią i rozszerzającej klasę mającą udział w danym wzorcu. Linie łączące wzorce z klasami są oznaczane nazwami ról odgrywanych przez te klasy w odpowiednich wzorcach.

Okno od strony południowo-zachodniej pełni funkcję wlotu powietrza dla wentylacji poprzecznej, natomiast drzwi naprzeciwko tego okna są ujściem powietrza. Dwa pozostałe okna nie odgrywają żadnej roli we wzorcu wentylacji poprzecznej, ponieważ są rozmieszczone na ścianach przylegających do ściany południowo-zachodniej. Skupmy się teraz na czymś innym — okna południowo-zachodnie i południowo-wschodnie należą do wzorca biura narożnego, ponieważ pełnią funkcję dwóch źródeł naturalnego światła. Ani drzwi, ani okno północno-zachodnie nie należą do tego wzorca projektowego, ponieważ nie stanowią wystarczających źródeł światła. Okno od strony południowo-zachodniej jest o tyle interesujące, że należy do dwóch wzorców projektowych jednocześnie — pełni funkcję „źródła powietrza" we wzorcu wentylacji poprzecznej oraz „źródła światła" we wzorcu biura narożnego. Obiekty i klasy występujące w rozmaitych wzorcach często wzajemnie się mieszają w podobny sposób.

Kluczowe znaczenie ma zrozumienie, że identyfikacja wzorców nie jest możliwa wyłącznie na podstawie analizowanej struktury. Przykładowo, wentylacja może być blokowana np. przez meble — wówczas żadne z okien nie będzie stanowiło źródła powietrza. Podobnie, jedno z okien może się znajdować metr od ściany sąsiedniego budynku lub łączyć gabinet z korytarzem — wówczas nie jest wystarczającym źródłem naturalnego światła (choć niewykluczone, że sprawdza się w roli źródła lub ujścia powietrza). Kiedy będziesz analizował rzeczywiste wzorce, przekonasz się, że do identyfikacji wzorca projektowego w programie komputerowym niezbędna jest wiedza związana z kontekstem (włącznie ze znajomością zamierzeń architekta). Nie można identyfikować wzorców projektowych wyłącznie na podstawie diagramów UML. Musisz znać docelowe (a więc zgodne z zamierzeniami architekta) zastosowanie poszczególnych obiektów lub klas. W kolejnych rozdziałach będziesz miał okazję zapoznać się z wieloma przykładami tego niezwykłego zjawiska.

Wracając do problemu kopiowania i wklejania — mam nadzieję, że rozumiesz już reguły rządzące reifikacją wzorców i jesteś sobie w stanie wyobrazić ich stosowanie w rozmaitych projektach, z których każdy można by zaimplementować na mnóstwo sposobów. Twierdzenie, że można skopiować i wkleić wzorzec za pomocą odpowiedniego narzędzia projektowego jest pozbawione sensu. Niezależnie od tego wielu producentów obiektowych narzędzi CASE zachwala swoje programy jako zawierające bogate „biblioteki wzorców", z których możesz wybierać gotowe wzorce i wstawiać je do swoich projektów. W praktyce biblioteki te zawierają jedynie gotowe struktury języka UML dla pojedynczych reifikacji wzorca zaprezentowanego w książce Bandy Czworga. Co prawda wklejanie niektórych spośród tych struktur czasami może być pewnym ułatwieniem, jednak nie należy tej prostej operacji mylić z dużo bardziej wymagającym procesem stosowania wzorca w projekcie. Dobry projekt niemal zawsze musi wykorzystywać własną reifikację, która jest właściwa tylko w określonym kontekście. Bezmyślne kopiowanie i wklejanie ma tyle wspólnego z projektowaniem, co zabawa w „malowanie liczbami" z rzeczywistym malowaniem.

Czemu te wzorce mają w ogóle służyć?

Skoro wzorce nie mają żadnego konkretnego kształtu, czemu tak naprawdę służą?

Kiedy po raz pierwszy przeczytałem książkę Bandy Czworga, byłem rozczarowany. Początkowo miałem wrażenie, że nie jest to nic innego jak pedagogiczna prezentacja materiału, który jest już doskonale znany wszystkim kompetentnym projektantom oprogramowania (bardzo często próbującym bezskutecznie poszukiwać eleganckich rozwiązań dla problemów, których dotyczą wzorce). Przyznaję, że gdybym przeczytał tę książkę kilka lat wcześniej, mój umysł być może nie byłby tak obciążony pewnymi wątpliwościami i niechęcią do nowinek — zamieszanie wokół wzorców projektowych początkowo traktowałem jak szum wokół niczego.

Myślałem w ten sposób do momentu, w którym musiałem omówić pewien projekt z innym projektantem. Mój rozmówca wskazał fragment projektu i powiedział: „Te interfejsy tworzą most łączący te dwa podsystemy; sam most jest implementowany za pomocą tego zbioru adapterów (zarządców) obiektów". Byłem zażenowany tym, co się stało. Tym jednym zdaniem mój kolega prawdopodobnie oszczędził dobre pół godziny starannych wyjaśnień. Pomyślałem wtedy, że być może rzeczywiście w tych wzorcach projektowych kryje się jakaś korzyść.

Niedługo potem udałem się na prezentację pierwszej wersji języka programowania Java, podczas której wszystkie składniki podsystemu AWT były opisywane właśnie na bazie odpowiednich wzorców. Prezentacja była krótka i jednocześnie zupełnie klarowna — w praktyce była dużo krótsza i prostsza od odpowiedniej prelekcji wygłaszanej przez osobę, która nie posługuje się wzorcami.

Zanim przystąpiłem do pracy nad następnym projektem raz jeszcze przeczytałem książkę Bandy Czworga, po czym specjalnie próbowałem rozważać zadania mojego projektu pod kątem możliwości stosowania ewentualnych wzorców. Zacząłem sobie zadawać

pytanie: „Co tak naprawdę próbuję osiągnąć i czy istnieją jakieś wzorce, które mogłyby mi to zadanie ułatwić?" (do znajdowania właściwych wzorców wykorzystywałem te fragmenty dostępnych opisów, które były poświęcone ich przeznaczeniu). Kiedy odpowiedź brzmiała „tak", bez zastanowienia używałem wybranego wzorca. Szybko odkryłem, że takie podejście z jednej strony znacznie skraca czas projektowania, z drugiej strony przyczynia się do poprawy jakości projektu. Im lepiej znałem potrzebne wzorce, tym szybciej mogłem pracować nad kolejnymi projektami. Co więcej, moje początkowe wersje projektów wymagały znacznie mniejszej liczby poprawek niż projekty przygotowywane w sposób tradycyjny.

Złapałem bakcyla.

Wzorce stanowią organizacyjny szkielet, który zasadniczo poprawia możliwości komunikacyjne, będące przecież w dłuższej perspektywie prawdziwym celem projektu. Dyskusje, które poprzednio wymagały godzin, teraz — właśnie dzięki wzorcom — zajmują tylko kilka minut, co powoduje, że wszystkie osoby zaangażowane w prace nad projektem mogą realizować swoje zadania szybciej. Swego czasu uważnie czytałem wszystkie materiały, do których mogłem dotrzeć, i odkryłem, że *książka Bandy Czworga* dotyczyła jedynie warstwy wierzchniej tego głębokiego tematu. W internecie i w literaturze branżowej ukazywały się setki udokumentowanych wzorców projektowych, z których wiele miało zastosowanie w wykonywanej przeze mnie pracy. Z czasem doszedłem do przekonania, że właśnie solidna wiedza o wzorcach przydatnych w mojej pracy może być zasadniczym elementem decydującym o szybkości i jakości wykonywania zadań. (Przez „solidną" wiedzę o wzorcach rozumiem ich znajomość na pamięć, a więc taką, która nie będzie wymagała wertowania książek).

Rola wzorców w procesie projektowania

Kiedy wzorce zaczęły być wykorzystywane w procesach projektowania i jaka jest ich rola w tych procesach? Odpowiedź na to pytanie różni się w zależności od użytej metodyki (mam nadzieję, że *używasz* jakiejś metodyki), jednak zainteresowanie wzorcami projektowymi wynika przede wszystkim z codziennych problemów rozwiązywanych na poziomie implementacji, zatem źródeł popularności wzorców w procesach projektowych należy upatrywać w zbliżeniu warstw projektowania i implementacji. Ważniejsze jest inne pytanie: Kiedy kończy się analiza (która jako taka wiąże się z dziedziną problemu) i rozpoczyna projektowanie (które z natury rzeczy dotyczy implementacji)?

Najlepszą znaną mi analogią jest projektowanie i konstruowanie budynków. Plany budynków nie obejmują wszystkich szczegółów konstrukcyjnych. Najczęściej pokazują rozmieszczenie murów, ale nie mówią o sposobie ich wznoszenia. Określają miejsca instalacji wodno-kanalizacyjnej, ale nie definiują rozmieszczenia wszystkich rur. Dopiero w czasie konstruowania budynku *projektowane* są konkretne rozwiązania w zakresie wznoszenia murów i prowadzenia sieci wodno-kanalizacyjnej, jednak proponowane tam rozwiązania rzadko są w pełni realizowane, ponieważ proces implementacji

stawia przed budowniczymi konkretne problemy do rozwiązania (często nieprzewidywalne w fazie przygotowywania projektu). Przykładowo, cieśla może zastosować określony wzorzec „rozmieszczenia śrub i gwoździ", który zagwarantuje odpowiednio mocną konstrukcję ściany. Projekt przeważnie mówi tylko o miejscu, w którym powinna stanąć ściana, nie wspomina jednak o sposobie jej konstruowania.

Z analogiczną sytuacją mamy do czynienia w świecie oprogramowania: w przypadku większości przedsięwzięć programistycznych proces projektowania powinien się zakończyć w punkcie, w którym *dobry* programista może bez trudu ten projekt zaimplementować. Nie jestem w stanie wyobrazić sobie opisu techniki tworzenia okien w podsystemie graficznego interfejsu użytkownika Swing zawartego w projekcie. Jest to jedno z tych zadań, które programista po prostu powinien potrafić zrealizować; a jeśli kod jest pisany z zachowaniem profesjonalnych standardów (a więc z uważnie dobranymi nazwami, właściwym formatowaniem, komentarzami w miejscach, gdzie jest to konieczne itp.), decyzje podejmowane w fazie implementacji i tak powinny zostać odpowiednio udokumentowane.

W efekcie wzorce projektowe rzadko są szczegółowo opisywane w dokumentach tworzonych na etapie projektowania — zamiast tego reprezentują decyzje podejmowane przez osoby odpowiedzialne za implementację. Wzorce stosowane przez te osoby niemal nigdy nie są dogłębnie dokumentowane, zwykle same nazwy elementów tych wzorców (lub inne komentarze) powinny w stopniu wystarczającym identyfikować podejmowane działania. (Przykładowo, klasa `WidgetFactory` jest reifikacją wzorca klasy-wytwórni).

Istnieją oczywiście wyjątki od reguły oddzielania wzorców od właściwych projektów. W świecie oprogramowania odpowiednik okien występujących we wzorcu narożnego biura może się pojawiać w dokumentach projektowych (które wskazują programistom miejsca rozmieszczenia okien). Podobnie, bardzo skomplikowane systemy, których projekty wymagają stosowania znacznie bardziej szczegółowych opisów (tak jak plany architektoniczne dla drapacza chmur są dużo bardziej szczegółowe od planów dla domku jednorodzinnego), często już w wyniku fazy projektowania mają dogłębnie udokumentowane wzorce projektowe.

Napięcia pomiędzy wzorcami a prostotą

Istotnym problemem są dodatkowe komplikacje wprowadzane do systemów właśnie w wyniku stosowania wzorców projektowych. Jak ślepa zgodność jest domeną ludzi nierozgarniętych, tak niepotrzebna złożoność jest domeną kiepskich programistów. „Mali politycy, filozofowie i duchowni" ponad wszystko cenią sobie zgodność, natomiast wielu „małych" programistów i architektów sądzi, że wzorce projektowe są ze wszech miar dobre, i że powinni ich używać wszędzie tam, gdzie nadarza się taka okazja. Takie bezmyślne podejście niemal zawsze prowadzi do powstawania kruchych, trudnych w konserwacji programów komputerowych. Każdy wzorzec ma przecież swoje wady, które przemawiają za tym, by go nie stosować.

Proste systemy są łatwiejsze w budowie, łatwiejsze w konserwacji, mniejsze i szybsze od systemów skomplikowanych. Prosty system „maksymalizuje ilość wykonanej pracy" przez

zwiększanie (w porównaniu ze skomplikowanymi systemami) „ilości pracy, która nie jest wykonana niejako przy okazji". Program musi przecież robić dokładnie to, czego oczekuje od niego użytkownik. Dodawanie nieproszonych funkcji dramatycznie wydłuża czas tworzenia systemu i jednocześnie obniża jego stabilność.

Prostota zwykle nie jest celem łatwym do osiągnięcia. Programiści uwielbiają złożoność we wszelkich jej postaciach, zatem bardzo często wykazują silną skłonność do nadmiernego komplikowania swojej pracy. Z punktu widzenia wielu programistów znacznie łatwiej jest szybko zbudować dosyć skomplikowany system niż zmusić się do późniejszego upraszczania swojego dzieła. Programiści podejrzewający, że otrzymane wymagania będą z czasem rozwijane lub zmieniane mają tendencje do implementowania wymagań, które *mogą* się pojawić w przyszłości. Komplikowanie kodu tylko dlatego, że ktoś nabrał mglistych *podejrzeń* o przyszłych zmianach jest jednak zupełnie niewłaściwe. (Za każdym razem, gdy próbuję przewidzieć przyszłość, popełniam zasadnicze błędy). Programiści powinni pisać kod w taki sposób, aby dodawanie nowych lub modyfikowanie istniejących funkcji było możliwie proste, co nie oznacza, że muszą od razu implementować całą funkcjonalność, jaka przychodzi im do głowy.

Problemem może być także nadmierne uproszczenie rozwiązania, które z natury rzeczy powinno być skomplikowane. Programista, który naprawdę chce zrobić „dokładnie" to, co jest potrzebne (i w dążeniu do prostoty rezygnujący z wymaganej funkcjonalności), tworzy rozwiązanie równie złe, jak programista implementujący niepotrzebne funkcje. Przykładem nadmiernego uproszczenia jest popularna funkcja „cofnij". Alan Cooper — twórca Visual Basica i ceniony specjalista od interfejsów użytkownika — argumentuje, że nigdy nie należy pytać użytkowników, czy *naprawdę* chcą coś robić. Pewnie, że chcą — przecież gdyby nie chcieli, nie żądaliby określonych funkcji! Ile razy *zrezygnowałeś* z usuwania pliku tylko dlatego, że na ekranie pojawiło się to głupkowate okno dialogowe? Najlepszym rozwiązaniem problemu przypadkowego usunięcia pliku (lub innych, równoważnych problemów) jest wykonanie żądania użytkownika i jednocześnie zapewnienie możliwości cofnięcia tej operacji, jeśli użytkownik popełni błąd. Funkcja cofania zmian jest obsługiwana np. przez większość edytorów tekstu. (Wyobrażasz sobie, by edytor wyświetlał okno dialogowe z pytaniem: „Czy naprawdę chcesz usunąć ten znak?"). Funkcja cofania zmian jest jednak trudna do zaimplementowania, więc dosyć często można się spotkać z tłumaczeniem lenistwa koniecznością utrzymania prostoty oprogramowania. „Kompletny system cofania zmian wprowadza tyle dodatkowej złożoności, że znacznie lepszym rozwiązaniem wydaje się zastosowanie tradycyjnych okien dialogowych z potwierdzeniami".

Te trzy wymagania (prostota, kompletność i łatwość modyfikacji) niekiedy wzajemnie się wykluczają. Wzorce opisane w tej książce są ogromnym ułatwieniem między innymi wtedy, gdy konieczna jest zmiana lub dodanie jakiegoś elementu, jednak udogodnienia w tym względzie są okupione dodatkowym komplikowaniem kodu źródłowego. Niestety, nie istnieje jedna prosta reguła opisująca kiedy stosowanie wzorców jest dobre, a kiedy nie jest zalecane — ocena przydatności wzorców należy do stosujących je programistów. Duże znaczenie ma tutaj doświadczenie, którego wielu projektantów i programistów po prostu nie ma (a także — jak słusznie stwierdził Ken Arnold, współautor oryginalnej książki o programowaniu w Javie — zmysłu estetycznego, który także nie jest częstą cechą wśród programistów). Można więc pisać fatalne programy, które na każdym kroku wykorzystują wzorce projektowe. Samo stosowanie wzorców nie jest żadną gwarancją sukcesu w świecie oprogramowania.

Z drugiej strony, budowa w kodzie źródłowym bloków wzorców (np. przez częste wykorzystywanie interfejsów) zawsze jest rozwiązaniem korzystnym, nawet wtedy, gdy stosowanie w pełni rozwiniętych wzorców nie jest uzasadnione. Interfejsy nadmiernie nie komplikują kodu źródłowego, a ewentualny rozwój tego kodu w przyszłości będzie znacznie prostszy, jeśli system od początku będzie budowany na podstawie jasnej struktury interfejsów. Koszt takiego działania jest stosunkowo niski, natomiast potencjalne korzyści są bardzo duże.

Klasyfikacja wzorców

W niektórych sytuacjach przydaje się klasyfikacja wzorców, która ułatwia wybór właściwych rozwiązań. W tabeli 1.1 (pochodzącej z książki Bandy Czworga) przedstawiono jeden ze sposobów podziału wzorców projektowych. Możesz jednak samodzielnie tworzyć podobne tabele, w których będziesz dzielił wzorce na kategorie według dobranych przez siebie kryteriów.

Tabela 1.1. *Klasyfikacja wzorców projektowych według Bandy Czworga*

		Cel		
		Konstrukcyjne	**Strukturalne**	**Czynnościowe**
Zakres	**Klasa**	Metoda wytwórcza	Adapter klas	Interpretator Metoda szablonowa
	Obiekt	Wytwórnia abstrakcji Budowniczy Prototyp Singleton	Adapter obiektu Most Kompozyt Dekorator Fasada Waga piórkowa Pośrednik	Łańcuch odpowiedzialności Polecenie Iterator Mediator Memento Obserwator Stan Strategia Wizytator

Banda Czworga podzieliła wzorce projektowe na dwa zakresy: wzorce klas wymagają reifikacji implementacji mechanizmu dziedziczenia (słowo kluczowe `extends`), natomiast wzorce obiektów powinny być implementowane wyłącznie z wykorzystaniem mechanizmu dziedziczenia interfejsów (słowo kluczowe `implements`). To nie przypadek, że istnieje znacznie więcej wzorców klas niż wzorców obiektów. (Więcej informacji na ten temat znajdziesz w następnym rozdziale).

W ramach tych dwóch zakresów wzorce są dalej dzielone na trzy kategorie. Wzorce konstrukcyjne dotyczą wyłącznie tworzenia obiektów. Przykładowo, wzorzec wytwórni abstrakcji zapewnia mechanizmy tworzenia obiektów bez znajomości klas, do których te nowe obiekty należą. (Na tym etapie trochę upraszczam rzeczywiste znaczenie tego wzorca, ale zagadnienia te szczegółowo wyjaśnię w dalszej części tej książki). Wszystkie

wzorce strukturalne należą do modelu statycznego — obejmują organizację strukturalną programu. Przykładowo, most opisuje sposób oddzielania od siebie dwóch podsystemów w taki sposób, aby każdy z nich mógł być modyfikowany bez konieczności zmian drugiego. Wszystkie wzorce czynnościowe należą do tzw. modelu dynamicznego, a więc obejmują techniki wzajemnego oddziaływania obiektów w czasie wykonywania programu. Przykładowo, wzorzec łańcucha odpowiedzialności opisuje taki system przesyłania komunikatów pomiędzy obiektami, który umożliwia wypełnianie tych komunikatów danymi i ich przetwarzanie przez obiekty potrafiące tak przygotowane komunikaty właściwie obsługiwać. Okazuje się, że jeszcze w czasie kompilacji nie musisz wiedzieć, które to obiekty; odpowiednie decyzje będą podejmowane dopiero w fazie wykonywania programu.

Wszystkie te wzorce omówię szczegółowo (choć w innej kolejności) w dalszej części tej książki, pamiętaj jednak, że istnieje wiele innych kategorii wzorców projektowych poza tymi, które zostały zidentyfikowane w książce Bandy Czworga. Wzorzec programowania czasu rzeczywistego, wzorzec przetwarzania wielowątkowego czy wzorzec Enterprise JavaBean (EJB) Javy to tylko kilka z wielu przykładów.

Kolejnym ważnym zagadnieniem są występujące pomiędzy wzorcami wzajemne zależności. Przykładowo, podczas lektury dalszej części tej książki przekonasz się, że wzorzec polecenia w takiej czy innej formie występuje we wszystkich pozostałych wzorcach czynnościowych. W książce Bandy Czworga przedstawiono diagram pokazujący te relacje zależnościowe, ale, szczerze mówiąc, ich diagram jest na tyle zawiły, że nie ma zbyt dużej wartości praktycznej. Najważniejsze jest zapamiętanie, że rozmaite wzorce rzeczywiście są ze sobą powiązane, czasem w znacznym stopniu, czasem w sposób niejasny.

Jeśli masz problem z odróżnieniem jednego wzorca od innego, z pewnością nie jesteś sam. Tego typu nieporozumienia najczęściej są wynikiem naturalnych zależności pomiędzy wzorcami. W takich przypadkach radzę się skupiać na tych częściach opisu wzorców, które dotyczą ich celu (przeznaczenia) — pamiętaj, że każda reifikacja zgodna z zamierzeniami projektanta jest dopuszczalna. Sama analiza struktury (naturalna w przypadku programistów) często rodzi niepotrzebne nieporozumienia. Z pewnością odkryjesz, że np. wszystkie wzorce strukturalne mają niemal identyczne struktury statyczne, choć szczegółowe efekty stosowania tych struktur są bardzo zróżnicowane. Struktury te dotyczą w równym stopniu komunikacji i oprogramowania, zatem nie należy się skupiać wyłącznie na analizie oprogramowania.

O programowaniu, ogólnie

Kolejnym ważnym zagadnieniem, które muszę przynajmniej ogólnie omówić jeszcze przed przystąpieniem do analizy samych wzorców jest projektowanie obiektowe (zorientowane obiektowo).

Po pierwsze, projektowanie obiektowe (ang. *Object-Oriented Design — OOD*) oraz programowanie obiektowe (ang. *Object-Oriented Programming — OOP*) to dwie zupełnie różne dziedziny. Proces projektowania rozpoczyna się od gromadzenia wymagań i wiąże się z systematycznym realizowaniem takich zadań jak analiza przypadków użycia — efektem tego procesu jest projekt, na podstawie którego programista może opracowywać

kod źródłowy programu. Proces programowania rozpoczyna się od analizy projektu lub jego części oraz stosowania takich rozwiązań jak wyprowadzanie struktur, hermetyzacja oraz wzorce projektowe, by ostatecznie stworzyć program komputerowy (będący realizacją otrzymanego na początku projektu). Wielu ludzi myli programowanie z projektowaniem. To, że od sześciu lat programujesz w Javie, rozumiesz mechanizmy wydzielania podklas i potrafisz pisać 1000 linii testowanego na bieżąco kodu dziennie nie oznacza jeszcze, że znasz się na projektowaniu obiektowym. Istnieje nawet teoria, zgodnie z którą wielu nawet najlepszych programistów nie rozumie podstawowych reguł projektowania obiektowego.

Dobrą analogią jest branża budowlana. Budynki są *projektowane* przez architektów, ale *budowane* przez wykonawców. Systemy obiektowe są projektowane przez projektantów obiektowych i implementowane przez programistów obiektowych. Te dwie role mogą co prawda być łączone przez jedną osobę, ale takie rozwiązanie jest dosyć rzadkie. Architekci muszą wiedzieć, jak konstruuje się budynki — w przeciwnym razie nie byliby w stanie opracowywać właściwych projektów. Z drugiej strony, wykonawcy w ogóle nie muszą rozumieć metod pracy architektów. (Nie twierdzę przy tym, że nie ma architektów, którzy bardzo chętnie projektowaliby budynki nienadające się do budowy lub do zagospodarowania, lub że nie istnieją wykonawcy, którzy potrafią bez trudu identyfikować kiepskie projekty). Najlepsi programiści są też dobrymi architektami, a najlepsi architekci są jednocześnie dobrymi programistami. To wymieszanie umiejętności jest szczególnie istotne w popularnych obecnie tzw. metodykach lekkich, zwinnych (ang. *agile*), które przewidują równoległe projektowanie i kodowanie. Żadna z tych metodyk nie uwzględnia nadrzędnej roli architekta, który pociąga za sznurki kierujące działaniami programistów.

Mówi się, że wielu programistów jest doświadczonymi rzemieślnikami, którzy produkują co prawda piękny kod, ale w ogóle nie rozumieją procesu projektowania — są budowniczymi, nie projektantami. Proszę, nie traktuj tego zdania jako przejawu mojej pogardy dla niesamowitych umiejętności budowniczych — chodzi tylko o to, że projekty przygotowywane *ad hoc* przez programistów są często dalekie od ideału.

Ostatni raport zespołu Standish Group dotyczący tysięcy projektów programistycznych powstałych w ciągu wielu lat stwierdza, że około 72 procent tego typu przedsięwzięć zakończyło się niepowodzeniem. Za najważniejszą przyczynę tak wysokiego współczynnika nieudanych projektów uznano właśnie brak odpowiednich projektów i wszystkie jego następstwa (a więc np. niewłaściwa metodyka gromadzenia informacji o wymaganiach). Oznacza to, że nawet doświadczeni architekci mogą popełniać błędy, jeśli zaniechają stosowania reguł rządzących procesami architektonicznymi.

Ta książka jest poświęcona programowaniu obiektowemu i architekturom obiektowym, nie samym procesom planowania architektur. Wzorce projektowe dotyczą zwykle szczegółów implementacyjnych, które mają zastosowanie w pracy programistów obiektowych w czasie przekładania otrzymywanych projektów na kod źródłowy programów. Stworzenie dobrego projektu nie jest jednak możliwe bez odpowiednich procesów. (Z pewnością takimi procesami są rozwiązania proponowane w ramach metodyk zwinnych). Co więcej, nie jest możliwe tworzenie dobrego kodu bez właściwych projektów (które dodatkowo mogą ewoluować w czasie). Proste stosowanie wzorców projektowych w kodzie (*ad hoc*, w nieprzemyślany sposób) z pewnością nie przyczyni

się do znacznego poprawienia jakości programu, a w skrajnych przypadkach może tę jakość dodatkowo obniżyć. Niepotrzebne komplikacje — a wiele wzorców projektowych jest skomplikowanych — niczego nie poprawią.

Proszę więc, abyś nie mylił zagadnień omawianych w tej książce z procesem projektowania obiektowego jako całością. Wzorce są tylko niewielką częścią tej układanki — w niektórych przypadkach ich udział jest wręcz minimalny. Nie jest to książka o projektowaniu obiektowym, tylko o przekształcaniu projektów obiektowych w konkretne implementacje. Aby jednak skutecznie stosować wzorce projektowe, musisz wiedzieć, jak projektować. Musisz znać proces projektowania. Na wspomnianej w przedmowie stronie internetowej wymieniłem wiele książek poświęconych projektowaniu i zalecam ich uważne przeczytanie.

Programowanie języka FORTRAN w Javie

Skoro niniejsza książka jest tak ściśle związana z zagadnieniami programowania obiektowego, warto chyba poświęcić chwilę na omówienie różnic pomiędzy techniką obiektową a proceduralnym programowaniem systemów (przynamniej na poziomie strukturalnym). Proceduralne podejście do programowania można scharakteryzować jako „ukierunkowane na dane", gdyż struktura programu proceduralnego koncentruje się wokół przepływu danych pomiędzy funkcjami (procedurami), które te dane przetwarzają lub sprawdzają. Centralnym elementem projektów tego typu programów zwykle jest baza danych; w praktyce bardzo wiele programów proceduralnych sprowadza się do prezentowania tabel bazy danych za pośrednictwem odpowiedniego interfejsu użytkownika. Systemy proceduralne są zwykle mocno zhierarchizowane, koncentrują się wokół pojęcia „globalnej kontroli". Element globalny (mający zwykle postać funkcji lub procedury na szczycie tej hierarchii) przetwarza dane zebrane z innych źródeł — albo funkcji na niższych poziomach hierarchii, albo utworzonych wcześniej danych globalnych. Główną wadą systemów proceduralnych są utrudnienia w diagnostyce i konserwacji. Współdzielone dane tworzą relacje „sprzęgające" (niepożądane zależności) pomiędzy poszczególnymi funkcjami systemu. Kiedy zostanie zmieniona jedna funkcja, wprowadzona modyfikacja będzie miała wpływ na działanie pozostałych. W skrajnych przypadkach efektem z pozoru banalnej zmiany w jednej z funkcji może być konieczność poświęcenia całych miesięcy na oczyszczenie i naprawienie kodu całego systemu.

Z drugiej strony, systemy obiektowe są w istocie siatką współpracujących agentów, które wzajemnie się komunikują za pośrednictwem jakiegoś systemu przesyłania komunikatów. Obiekty są równoważne — nie istnieje obiekt zwierzchni, który mógłby wydawać dyrektywy pozostałym obiektom. Co prawda właściwości dobrze zaprojektowanych obiektów będę szczegółowo omawiał w dalszej części tego rozdziału, warto jednak już teraz wprowadzić kilka najważniejszych zagadnień. Patrząc na obiekt z zewnątrz, nie powinniśmy mieć pojęcia o sposobie jego implementacji. Powinna więc istnieć możliwość wymiany całej implementacji bez konieczności modyfikowania któregokolwiek z obiektów klienckich (obiektów wykorzystujących obiekt, który właśnie został zmieniony). Mimo że obiekty czasami przekazują pomiędzy sobą inne obiekty, przepływ danych jest zupełnie inny niż w systemach proceduralnych. Obiekt pilnie strzeże swoich danych i wykonuje na nich operacje w odpowiedzi na otrzymywane komunikaty. Obiekty

nie przekazują danych do innych obiektów, chyba że jest to absolutnie konieczne (nawet wówczas przekazywane dane są hermetycznie zamykane w ramach tworzonych w tym celu obiektów). Te dwie koncepcje (*ukrywanie implementacji* i *abstrakcja danych*) w świecie systemów obiektowych mają kluczowe znaczenie.

Dobrym sposobem odróżniania systemów obiektowych od systemów proceduralnych jest analiza efektów wprowadzanych zmian. W systemach proceduralnych wszelkie zmiany wpływają na pozostałe składniki programu, a duże zmiany w działaniu programu wymagają zwykle bardzo głębokich i rozległych zmian w całym kodzie. W systemach obiektowych niezbędne modyfikacje można wprowadzać tylko w wybranych punktach. Pojedyncza zmiana w kodzie systemu obiektowego może powodować znaczne zmiany w zachowaniu całego programu. Przykładowo, jeśli decydujesz się na zmianę formatu danych wykorzystywanego do przechowywania informacji, w przypadku systemu proceduralnego musiałbyś zmienić kod w wielu miejscach, ponieważ dane są przetwarzane przez wiele procedur; w systemie obiektowym zmiany będą dotyczyły wyłącznie obiektu odpowiedzialnego za składowanie danych.

Oczywiście takie reguły programowania obiektowego jak zapewnianie abstrakcji danych (ukrywanie sposobu działania zestawu funkcji przez chowanie struktur danych przed użytkownikami tych funkcji) istniały od dawna i stanowiły podstawę technik tworzenia wysokiej jakości oprogramowania w rozmaitych środowiskach i językach — także proceduralnych. Przykładowo, zarówno system obsługi operacji wejścia-wyjścia na plikach w języku C, jak i biblioteka *Curses* Kena Arnolda są rozwiązaniami obiektowymi. System proceduralny może miejscami wyglądać jak system obiektowy. System „czysto" obiektowy charakteryzuje się przede wszystkim konsekwentnym i skrupulatnym stosowaniem pewnych reguł (np. wspominanej już abstrakcji danych).

Systemy obiektowe odróżnia od systemów proceduralnych jeszcze jeden kluczowy element. Przykładowo, systemy obiektowe w większości przypadków odwzorowują (modelują) procesy świata rzeczywistego. Ten tok myślenia może Cię doprowadzić do ogólnej koncepcji procesu projektowania obiektowego; ponieważ jednak ta książka w głównej mierze dotyczy struktury obiektowej, nie będę poświęcał tym zagadnieniom zbyt wiele czasu.

Wiele osób, które wychowały się w świecie systemów proceduralnych sądzi, że obiektowe podejście do problemów informatycznych jest czymś niewłaściwym. Zawsze jestem zdumiony, kiedy czytam o kontrowersjach wywołanych przez moje artykuły o technikach programowania obiektowego. Kiedy opublikowałem (w internetowym magazynie *JavaWorld*) wstępny zarys tej książki, doznałem prawdziwego szoku, czytając obelżywe opinie o rzekomym oderwaniu prezentowanych koncepcji od rzeczywistości — koncepcji, które przewijają się w literaturze informatycznej od trzydziestu lat. Byłem nazywany „dyletantem", „szukającym po omacku", „niegodziwcem", „głupkiem" i opisywany kilkoma innymi epitetami, których nie powinienem tutaj przytaczać. Moje artykuły były przez niektórych określane jako „durne" i „pozbawione najmniejszego sensu". Jeden z czytelników posunął się nawet do fizycznych gróźb, rozpoczynając swój napastliwy list (szybko usunięty przez administratora witryny) w następujący sposób: „TEEMU [*sic*] AUTOROWI KTOŚ WRESZCIE POWINIEN ROZWALIĆ ŁEB PRĘTEM!"

Nie należy utożsamiać tego, co jest nam „znane", z tym, co jest „prawidłowe". Wielu programistów zakłada, że biblioteki, które regularnie wykorzystują w swojej pracy są „właściwe"; a jeśli któraś z nich wykonuje pewne operacje w określony sposób, programiści ci uważają, że właśnie ta technika realizacji tego typu operacji jest standardem. Takie postrzeganie swojego środowiska można obserwować szczególnie często u osób, które uczyły się programowania z podręczników opisujących rozwiązywanie konkretnych problemów i realizację ściśle wybranych zadań. Jeśli jedyną architekturą, z którą kiedykolwiek mieli do czynienia, było EJB lub Struts, będą klasyfikowali jako niewłaściwe wszelkie rozwiązania nieprzypominające ani EJB, ani Struts. To, że w przeszłości realizowaliśmy zadania w określony sposób, wcale nie oznacza, że był to sposób najlepszy; gdyby rzeczywiście tak było, nadal programowalibyśmy w językach asemblerowych.

Wiele lat temu odbyłem bardzo interesującą rozmowę z osobą, która pracowała w firmie Microsoft nad środowiskiem programowania w języku C++ i nad biblioteką Foundation Class (MFC). Kiedy stwierdziłem, że MFC nie jest biblioteką obiektową, odpowiedział, że doskonale zdaje sobie z tego sprawę, ale większość ludzi programujących dla systemów firmy Microsoft po prostu nie rozumie reguł rządzących światem obiektów. Powiedział też, że uczenie ludzi programowania obiektowego nie jest zadaniem Microsoftu. Efekt był taki, że Microsoft celowo stworzył system proceduralny w języku C++, ponieważ właśnie taki system był „łatwiejszy do zrozumienia". Teoria o trudnym do zrozumienia programowaniu obiektowym jest w firmie Microsoft zakorzeniona tak głęboko, że nawet struktura interfejsów API technologii .NET jest w istocie proceduralna, a np. język programowania C# zawiera mechanizmy wręcz zachęcające do myślenia proceduralnego. Nic dziwnego, że wiele aplikacji firmy Microsoft jest tworzonych wbrew podstawowym regułom rządzącym systemami obiektowymi. Wielu spośród programistów tej firmy podchodzi bardzo nerwowo do wszelkich technik obiektowych, które nie są zgodne z rozwiązaniami zastosowanymi w ramach technologii .NET. Mylą to, co jest im „znane", z tym, co „prawidłowe".

Nie próbuj do systemów obiektowych przykładać miary typowej dla systemów proceduralnych, nie krytykuj też opisywanych przeze mnie technik obiektowych tylko dlatego, że nie są zgodne z koncepcją programowania proceduralnego. Wiele popularnych pojęć związanych z programowaniem obiektowym mogło w ogóle nie występować w istniejącym kodzie, który miałeś okazję analizować. Twierdzenie, że któraś z technik kodowania jest niewykonalna w systemach obiektowych wcale nie musi oznaczać, że dana technika w ogóle nie ma zastosowań. Będę o tym przypominał za każdym razem, gdy będę analizował obiektowe podejście do jakiegoś problemu.

I wreszcie pamiętaj, że rozwiązanie „czysto" obiektowe nie zawsze jest konieczne ani najwłaściwsze. Tak jak w przypadku większości decyzji podejmowanych w fazie projektowania rozwiązań, koncepcja projektowania obiektowego ma pewne wady i może być źródłem dodatkowego ryzyka. Przykładowo, prosta witryna internetowa zbudowana na podstawie serwletów, która stanowi tzw. cienki fronton bazy danych, prawdopodobnie nie musi być tworzona zgodnie z surowymi regułami programowania obiektowego. W tym przypadku ryzyko sprowadza się do nadmiernego komplikowania kodu (nawet uniemożliwiającego właściwe zarządzanie) w związku z naturalną ewolucją programu. Podobnie, wielu programistów nie rozumie koncepcji programowania obiektowego, jeśli więc Twój system nie ma określonych długoterminowych wymagań w zakresie konserwacji, i jeśli można przyjąć, że wymagania biznesowe nie ulegną zmianie, przypisanie

do realizacji tego zadania programisty, który będzie potrafił szybko zaimplementować rozwiązanie proceduralne wcale nie musi być złym posunięciem. Istnieje oczywiście ryzyko, że czas życia tego programu będzie dłuższy od oczekiwanego, lub że nastąpią istotne zmiany reguł biznesowych, które sprawią, że wyrzucenie oryginalnego kodu do kosza będzie mniej kosztowne od jego modyfikowania. Stosowanie rozwiązań proceduralnych nie jest z gruntu złe, powinieneś jednak podejmować odpowiednie decyzje z pełną świadomością przyjmowanego ryzyka.

Programowanie z otwartymi oczami

Porozmawiajmy teraz o ogólnej filozofii projektowania.

Projekt jest szeregiem odpowiednio uzasadnionych wyborów i ustępstw, a także analizą ryzyka. Jeśli nie rozumiesz obu stron rozwiązywanego problemu, nie możesz dokonywać rozsądnych wyborów i efektywnie zarządzać ryzykiem; w praktyce, jeśli nie masz świadomości wszystkich następstw podejmowanych decyzji, tak naprawdę niczego nie projektujesz, a jedynie błądzisz po omacku. To nie przypadek, że każdy rozdział książki Bandy Czworga zawiera podrozdział „Skutki", w którym opisano (wraz z uzasadnieniem) sytuacje, w których stosowanie danego wzorca projektowego jest niewłaściwe.

Co więcej, w świecie projektów pojęcia „dobra" i „zła" nie są bezwarunkowe. „Dobra" decyzja w jednym kontekście może być „złą" decyzją w innym kontekście. Każda decyzja ma swoje dobre i złe strony — jest podejmowana w kontekście wszelkich, zdefiniowanych z konieczności kryteriów. Ocena decyzji rzadko może być wyrażana w skali binarnej. Bardzo często mamy do czynienia z odcieniami słuszności (konsekwencji dokonanego wyboru), które w wielu przypadkach oznaczają, że żadna z rozważanych możliwości nie może być jednoznacznie uznana za „najlepszą". Co więcej, decyzje, które wyglądają na słuszne, za kilka miesięcy mogą być postrzegane zupełnie inaczej.

Twierdzenie, że któraś z funkcji języka lub jedno z popularnych rozwiązań programistycznych stwarza pewne problemy, wcale nie jest równoznaczne z tezą, że nigdy, w żadnych okolicznościach nie należy z tej funkcji lub z tego rozwiązania korzystać. Podobnie, sama popularność pewnej funkcji lub rozwiązania nie oznacza, że zawsze należy z nich korzystać. Bardzo wiele programów jest pisanych przez niedoinformowanych i niekompetentnych programistów, ponieważ samo zatrudnienie w takich firmach jak Sun, Microsoft czy IBM nie powoduje nagłej poprawy umiejętności programowania i projektowania. W pakietach Javy można znaleźć mnóstwo doskonałego kodu, ale także niemało kodu, w przypadku którego trudno znaleźć programistę gotowego się przyznać do autorstwa (co wcale mnie nie dziwi).

Projektowanie jest dodatkowo utrudniane przez fakt, że pewne cechy projektów są promowane wyłącznie z przyczyn marketingowych lub politycznych. Zdarza się, że to programista podejmuje złą decyzję, ale zatrudniająca go firma koniecznie chce sprawdzić *możliwości* wykorzystywanej technologii i celowo dezawuuje właściwe rozwiązania. To chyba najbardziej jaskrawy przykład nieodpowiedniego podejścia do problemu projektowania systemów informatycznych. W tym kontekście wdrażanie wszelkich technik programowania tylko dlatego, że „tak to się teraz robi" jest oczywiście przejawem wyjątkowej lekkomyślności. Doskonałym dowodem na poważne ryzyko wynikające

z takiego podejścia jest ogromna liczba nieudanych projektów opartych na technologii EJB. Jeśli technologia ta jest stosowana właściwie, może być bardzo przydatna; jeśli jednak jest wykorzystywana w sposób bezmyślny, może w krótkim czasie doprowadzić do bankructwa jej bezkrytycznych propagatorów.

Tak naprawdę próbuję Cię tylko zniechęcić do ślepego, bezmyślnego stosowania popularnych rozwiązań. Rozumienie potencjalnych zagrożeń wynikających ze stosowania danej funkcji lub danego rozwiązania stawia Cię w znacznie lepszej pozycji podczas decydowania o wykorzystaniu funkcji lub rozwiązania w swoim programie. Podejmując decyzję w tym zakresie, powinieneś korzystać z całej dostępnej wiedzy i jednocześnie wykazywać pragmatyzm — tylko w ten sposób możesz przynajmniej ograniczyć ryzyko dokonywania złych wyborów. Dlatego też podjąłem trud napisania tej książki; chciałem, abyś potrafił się poruszać w świecie programowania z otwartymi oczami.

Czym jest obiekt?

Co tak naprawdę oznacza popularne dzisiaj zorientowanie na obiekty (zorientowanie obiektowe)?

Omawiane w tej książce wzorce projektowe są składnikami systemów obiektowych (zorientowanych obiektowo). Jeśli jakiś system jako cały nie jest obiektowy, stosowanie wzorca obiektowego tylko w jednym z jego składników nie przyniesie oczekiwanych korzyści. Odkryłem, że wielu programistom (także tym, którzy latami pracowali w takich językach jak C++ czy Java) brakuje wiedzy o tym, co dokładnie czyni system informatyczny obiektowym, zatem muszę się już teraz upewnić, że mamy w tym względzie całkowitą jasność.

Nonsens!

Bjarne Stroustrup, twórca języka programowania C++, scharakteryzował kiedyś programowanie obiektowe jako „programowanie zorientowane na bełkot techniczny" i z pewnością jednym z najczęściej nadużywanych (lub przynajmniej używanych w sposób zupełnie bezmyślny) słów-bełkotów jest sam *obiekt*. Ponieważ właśnie koncepcja obiektu ma charakter centralny, precyzyjne wyjaśnienie, czym faktycznie jest obiekt, ma zasadnicze znaczenie dla rozumienia systemów obiektowych i ich potrzeb.

Po pierwsze, system obiektowy powinieneś traktować jak stado inteligentnych zwierząt (zbiór obiektów) zamkniętych w Twoim komputerze, które rozmawiają, przesyłając pomiędzy sobą *komunikaty*. Wyobraź sobie te „obiekty". Klasy są tutaj nieistotne — mają jedynie charakter struktur pomocniczych stworzonych z myślą o łatwiejszej pracy kompilatora. Zwierzęta tworzące taki system mogą być klasyfikowane (grupowane), jeśli mają podobne właściwości (jeśli np. potrafią obsługiwać te same komunikaty, co inne obiekty należące do tej samej klasy). Programiści mówiący o *klasach* tak naprawdę mają na myśli klasy obiektów. Oznacza to, że obiekty mające te same właściwości tworzą jedną klasę obiektów. Są to rozważania na poziomie językowym (nie jest to więc

techniczny bełkot) i właśnie ten sposób jest najwłaściwszą formą mówienia o systemach obiektowych. Zawsze mówimy o projektowaniu obiektowym, nigdy o projektowaniu klasowym.

Najważniejszym aspektem całej koncepcji projektowania obiektowego jest *abstrakcja danych*. Właśnie w abstrakcji danych tkwi tajemnica projektowania tego typu programów. Wszystkie informacje są ukrywane. Poszczególne obiekty „nie mają pojęcia" o zawartości pozostałych obiektów — nie „wiedzą" o nim więcej niż Ty o woreczku żółciowym swojego współmałżonka. (Zarówno w przypadku obiektów, jak i wspomnianego woreczka żółciowego zapewne żadna ze stron nawet nie chce tej brakującej wiedzy zdobyć).

Być może miałeś kiedyś okazję czytać książkę, której autor dowodził, że obiekt jest strukturą danych połączoną ze zbiorem funkcji, nazywanych *metodami*, przetwarzających tę strukturę danych. Nonsens! Bzdura!

Obiekt jest zbiorem zdolności

Po pierwsze i najważniejsze, obiekt jest definiowany przez to, co może *robić*, nie przez to, jak to robi. W sensie praktycznym oznacza to, że obiekt jest definiowany przez komunikaty, które może otrzymywać i wysyłać. „Metody" obsługujące te komunikaty tworzą jedynie interfejs łączący ten obiekt ze światem zewnętrznym. Nacisk należy kłaść na to, co obiekt może robić (na jego zdolności), nie na to, jak oferowane czynności zostały zaimplementowane. Przetwarzane „dane" nie mają większego znaczenia. Większość projektantów programów obiektowych spędza mnóstwo czasu na planowaniu określonych mechanizmów w oderwaniu od danych (będących tylko jednym ze składników obiektu). Zdecydowana większość obiektów będzie oczywiście wymagała jakichś danych potrzebnych do zaimplementowania wykonywanych czynności, ale sama struktura tych danych jest — a przynajmniej powinna być — nieistotna z punktu widzenia projektanta.

Naczelna zasada systemów obiektowych mówi:

> *Nigdy nie proś obiektu o informacje, których potrzebujesz do wykonania jakiejś czynności; zamiast tego proś obiekt zawierający te informacje o wykonanie tych czynności za Ciebie.*

Ken Arnold wyrażał tę zasadę w sposób następujący: „Proś o pomoc, nie o informacje".

Powyższą zasadę wyjaśnię za chwilę, najpierw jednak skupię się na kilku ważnych regułach wynikających wprost z tej zasady — na podstawie tych reguł możesz oceniać, czy rzeczywiście masz do czynienia z projektem systemu obiektowego (na razie przedstawię je w możliwie zwięzły sposób; więcej szczegółów znajdziesz w dalszej części tego rozdziału):

- Obiekty są definiowane na zasadzie „kontraktów". Obiekty nigdy nie łamią swoich kontraktów.
- Wszystkie dane są prywatne. Kropka. (Ta reguła powinna być stosowana do wszystkich szczegółów implementacyjnych, nie tylko do danych).

- Musi istnieć możliwość wprowadzania dowolnych zmian w sposobie implementacji obiektu (niezależnie od tego, jak istotne są te zmiany) przez modyfikowanie pojedynczej klasy, która ten obiekt definiuje.
- Funkcje zwracające i ustawiające (`get` i `set`) są złe, jeśli stosuje się je bezmyślnie (jeśli stanowią tylko wyszukany sposób upubliczniania danych przechowywanych w obiekcie). Zagadnienie to dodatkowo rozwinę w podrozdziale „Metody zwracające i ustawiające są złe".

Jeśli analizowany przez Ciebie system nie spełnia tych reguł, możesz być pewien, że nie masz do czynienia z systemem obiektowym. To bardzo proste. Nie twierdzę przy tym, że systemy nieobiektowe są złe — istnieje mnóstwo doskonałych systemów proceduralnych. Tak czy inaczej, ukrywanie danych jest w projektowaniu obiektowym zasadą absolutnie podstawową; jeśli więc naruszysz tę regułę, popełnisz niewybaczalny błąd. To samo dotyczy systemów obiektowych — jeśli system informatyczny łamie zasadę ukrywania danych, zgodnie z definicją nie jest systemem obiektowym, a jedynie jakąś dziwaczną hybrydą, która może, ale nie musi działać prawidłowo. Kiedy taki system popadnie w tarapaty i pociągnie za sobą Twoją firmę, nie wiń za to techniki obiektowej. Warto jednak pamiętać, że systemy obiektowe mogą być pisane w językach proceduralnych (i odwrotnie). To kwestia reguł stosowanych przez programistę, nie wykorzystywanego języka programowania.

Nie daj się zwieść marketingowym hasłom o systemach „opartych na technice obiektowej" lub o "istnieniu wielu różnych sposobów definiowania obiektów". Tego typu zapewnienia możesz tłumaczyć w następujący sposób: „Nasz produkt tak naprawdę nie jest obiektowy — my o tym wiemy, ale takiej wiedzy najprawdopodobniej nie ma Twój szef (który podejmuje decyzje o zakupach oprogramowania), więc próbujemy zaciemnić obraz w nadziei, że nikt tej mistyfikacji nie zauważy". Microsoft posunął się jeszcze dalej — wprowadził własną definicję systemów obiektowych, która pasowała do produktów oferowanych przez tę firmę. Przykładowo, w przeszłości język Visual Basic nie był językiem obiektowym i nawet teraz, kiedy inżynierowie i specjaliści od marketingu robią wszystko, by VB był postrzegany jako język obiektowy, większość programów pisanych w tym języku i tak nie jest obiektowa, ponieważ biblioteki Microsoftu nie są obiektowe. (Ilu programistów Microsoftu jest potrzebnych do wkręcenia żarówki? Żaden — zdefiniujmy ciemność jako nowy standard przemysłu informatycznego).

Przejdźmy teraz do omawiania wymienionych przed chwilą reguł, które rządzą systemami obiektowymi.

W pierwszej kolejności musimy wyjaśnić pojęcie *kontraktu*. Kontrakt obiektu definiuje sposób prezentowania jego zachowań dla świata zewnętrznego. Użytkownicy tego obiektu zakładają, że zachowanie to nie będzie podlegało zmianom. Częścią tego kontraktu są nie tylko interfejsy implementowane przez obiekt (nie możesz więc lekką ręką zmieniać np. argumentów metod czy typów zwracanych przez nie wartości), ale także gwarancje w zakresie wydajności, ograniczenia rozmiarów itp. Implementacja obiektu nigdy nie należy do jego kontraktu, zatem zawsze powinieneś mieć możliwość jej modyfikowania według uznania.

Wszystkie pozostałe wymienione reguły tak naprawdę mają na celu jedynie wymuszenie zgodności obiektu z kontraktem. Odkrycie szczegółów implementacyjnych w praktyce

oznaczałoby włączenie tych szczegółów do kontraktu obiektu, a więc wykluczenie możliwości modyfikowania implementacji (np. w odpowiedzi na wykrycie błędu lub pojawienie się nowych wymagań biznesowych).

Podobnie, regułę zapewniania prywatności wszystkich danych obiektu należy interpretować w następujący sposób — jeśli któryś ze składników obiektu nie jest prywatny, automatycznie staje się częścią kontraktu, zatem nie może być modyfikowany. W pewnych (rzadkich) sytuacjach decyzja o publicznym deklarowaniu pola obiektu może być poprawna, ale jej następstwa zawsze są bardzo poważne.

Pojęcie kontraktu odgrywa też istotną rolę w trzeciej z wymienionych reguł. Idealnym rozwiązaniem jest ograniczenie zakresu zmian do pojedynczej klasy, jednak w większości przypadków uniknięcie wzajemnych zależności jest po prostu niemożliwe. Przykładowo, klasa `HashMap` wymaga od przechowywanych obiektów implementowania metody `hashCode()`. To i podobne wymagania muszą być częścią kontraktów przechowywanych obiektów.

Jak nie należy tego robić?

Zasadniczym argumentem przemawiającym za stosowaniem reguł wymienionych w poprzednim punkcie jest znaczne ułatwienie konserwacji kodu źródłowego, ponieważ wszelkie zmiany związane z usuwaniem usterek bądź dodawaniem nowych funkcji są dokonywane w jednym miejscu. Nawiasem mówiąc, nie należy mylić łatwości konserwacji kodu z jego prostotą. Systemy obiektowe są zwykle bardziej skomplikowane od systemów proceduralnych, ale jednocześnie łatwiejsze w konserwacji. Idea programowania obiektowego sprowadza się do odpowiedniego organizowania nieuniknionej złożoności (będącej naturalnym elementem rzeczywistych programów komputerowych), nie do jej eliminowania — projektanci systemów obiektowych doskonale zdają sobie sprawę z faktu, że cel ten jest niemożliwy do osiągnięcia.

Przykładowo, podczas tworzenia systemu, który musi pobierać nazwisko od użytkownika możesz ulec pokusie użycia kontrolki `TextField`, z której odczytasz obiekt klasy `String` — takie rozwiązanie nie będzie jednak funkcjonowało prawidłowo we wszystkich możliwych sytuacjach. Co będzie, jeśli system będzie wykorzystywany w Chinach? (Twórcy Unicode — zbioru znaków wykorzystywanego w Javie — są obecnie bliscy osiągnięcia reprezentacji wszystkich znaków ideograficznych stosowanych w języku chińskim). Co będzie, jeśli ktoś zechce wpisać swoje imię za pomocą pióra (lub użyć mechanizmu rozpoznawania mowy) zamiast tradycyjnej klawiatury? Co będzie, jeśli wykorzystywana przez Ciebie baza danych nie zezwala na umieszczanie w niej znaków Unicode? Co powinieneś zrobić, jeśli za rok będziesz musiał zmodyfikować program w taki sposób, aby pobierał jednocześnie nazwisko i identyfikator pracownika wszędzie tam, gdzie do tej pory było pobierane lub wyświetlane samo nazwisko? W systemie proceduralnym rozwiązanie tych kwestii będzie się wiązało z poważnymi problemami w zakresie konserwacji kodu źródłowego (co w przypadku tego typu systemów jest zupełnie normalne). Po prostu nie istnieje łatwy sposób rozwiązywania nawet z pozoru bardzo prostych problemów, a zupełnie banalne modyfikacje wymagają zwykle ogromnego wysiłku.

Rozwiązanie obiektowe jest próbą hermetycznego zamknięcia tych składników oprogramowania, które najprawdopodobniej będą podlegały zmianom — w ten sposób modyfikowana część programu nie będzie miała wpływu na jego pozostałe składniki. Przykładowo, typowym rozwiązaniem obiektowym dla opisanego przed chwilą problemu jest zastosowanie klasy `Name`, której obiekty będą „wiedziały" nie tylko jak wyświetlać swoje informacje, ale także jak się inicjalizować. Wyświetlanie nazwiska będzie wówczas wymagało użycia polecenia „Wyświetl swoje informacje w tym miejscu" i przekazania obiektu klasy `Graphics` (a być może także obiektu klasy `Container`, za pośrednictwem którego obiekt nazwiska będzie mógł umieścić przechowywane dane w panelu `JPanel`). Obiekt klasy `Name` może oczywiście wybrać rozwiązanie polegające na utworzeniu kontrolki `TextField`, ale decyzja w tym względzie należy wyłącznie do niego. Ciebie, programistę, po prostu nie interesuje *sposób*, w jaki obiekt nazwiska będzie się inicjalizował (przynajmniej do czasu, kiedy ta inicjalizacja przebiega prawidłowo). Implementacja tego obiektu może nie tworzyć żadnych elementów interfejsu użytkownika — może np. pobierać wartość początkową przez wykonanie odpowiedniego żądania na bazie danych lub za pośrednictwem sieci komputerowej.

Wracając do mojej krytyki języka Visual Basic sprzed kilku akapitów, przyjrzyjmy się typowemu, strukturalnemu sposobowi generowania elementów interfejsu użytkownika przez ten język (lub niezliczone systemy podobne do Visual Basica): tworzysz klasę `Frame`, której zadaniem jest gromadzenie komunikatów przychodzących z „kontrolek" (z obiektów kontrolek) w odpowiedzi na działania podejmowane przez użytkownika. Klasa `Frame` przekazuje te komunikaty dalej do systemu obiektów. Zbudowany w ten sposób kod zazwyczaj przyjmuje następującą postać:

1. „Wyciąganie" jakiejś wartości za pomocą metody `get`.
2. „Umieszczanie" jakiejś wartości w obiekcie biznesowym za pośrednictwem metody `set`.

Takie rozwiązanie jest często nazywane architekturą modelu-widoku-kontrolera (ang. *Model-View-Controler*, w skrócie *MVC*) — kontrolki odgrywają rolę „widoku", obiekt klasy `Frame` pełni funkcję „kontrolera", natomiast działający w tle system jest „modelem".

Architektura MVC sprawdza się znakomicie w przypadku implementacji małych elementów, np. przycisków, ale kompletnie zawodzi na poziomie całych aplikacji, ponieważ koncepcja MVC przewiduje, że kontroler „wie" znacznie więcej o sposobie implementacji obiektów na poziomie modelu niż np. w koncepcji programowania obiektowego. W systemie zbudowanym na podstawie tej architektury zbyt dużo danych jest przesyłanych pomiędzy warstwami, aby była możliwa jego łatwa konserwacja.

Zamiast wierzyć mi na słowo, sam przeanalizuj kilka typowych problemów konserwacyjnych (opisywanych już w tym rozdziale), które wystąpią w przypadku prób tworzenia dużych programów na podstawie architektury modelu-widoku-kontrolera. Jeśli np. weźmiemy wspomniany już problem dodania identyfikatora pracownika do każdego okna wyświetlającego jego nazwisko, w architekturze typowej dla języka Visual Basic musiałbyś ręcznie zmodyfikować wszystkie te okna (przez zmianę lub dodanie kontrolek obsługujących nowe pole identyfikatora). Musiałbyś też dodać nowe elementy do klasy `Employee`, aby umożliwić ustawianie tego identyfikatora, a także skrupulatnie przeanalizować kod *każdej* klasy wykorzystującej obiekty klasy `Employee`, aby upewnić się,

że wprowadzenie identyfikatora nie rodzi żadnych dodatkowych komplikacji. (Przykładowo, operacje porównywania dwóch obiektów klasy `Employee` muszą teraz uwzględniać pole `ID`, zatem konieczne będzie zmodyfikowanie kodu zawierającego takie operacje). Gdybyś zawarł identyfikator pracownika w ramach klasy `Name`, mógłbyś uniknąć tego typu problemów. Obiekty tej klasy byłyby po prostu wyświetlane w nowy sposób. Dwa obiekty klasy `Name` można by nawet porównywać na podstawie informacji na temat identyfikatora, ale np. kod w postaci `fred.compareTo(ginger)` lub `fred.equals (ginger)` nie wymagałby wprowadzania żadnych modyfikacji.

Nie możesz nawet zautomatyzować procesu aktualizacji (dostosowywania) kodu, ponieważ cała ta funkcjonalność WYSIWYG, o której tak często można przeczytać w materiałach marketingowych, ukrywa proces generowania kodu. Jeśli automatycznie modyfikujesz kod wygenerowany przez komputer, wprowadzone zmiany zostaną utracone w momencie, w którym ktoś inny użyje tego samego generatora. Nawet jeśli jesteś pewien, że narzędzie to nie zostanie ponownie zastosowane, modyfikowanie wygenerowanego kodu jest zawsze ryzykowne, ponieważ większość narzędzi z rodziny Visual Basica wykazuje bardzo niewiele tolerancji dla kodu wyglądającego inaczej niż ten, który zazwyczaj generują. A jeśli Twoje modyfikacje będą polegały na wprowadzeniu jakichś nieoczekiwanych elementów, stosowane narzędzie może się pogubić do tego stopnia, że odmówi wykonywania na tym kodzie wszelkich dalszych operacji wtedy, gdy będziesz tego *naprawdę* potrzebował. Co więcej, automatycznie generowany kod zwykle jest dość kiepski, ponieważ tworzące go narzędzia nie uwzględniają takich czynników jak efektywność, zwięzłość, czytelność itp.

Bodaj największą wadą architektury MVC jest koncepcja „kontrolki siatki powiązanej z danymi" (ang. *data-bound grid control*), czyli takiego formantu-tabeli, który reprezentuje jednocześnie kod języka SQL potrzebny do wypełnienia jego komórek zawartością bazy danych. Co się dzieje, kiedy słownik danych ulega zmianie? Otóż cały ten osadzony kod SQL-a nadaje się wówczas do kosza lub przynajmniej wymaga zasadniczych modyfikacji. Będziesz musiał dokładnie przeanalizować wszystkie te okna swojego systemu, które zawierają kontrolki powiązane z danymi i modyfikować je za pomocą swojego wizualnego narzędzia. Zastosowanie architektury „trójwarstwowej", w której warstwa interfejsu użytkownika komunikuje się z warstwą zawierającą kod języka SQL, a ta z kolei współpracuje z warstwą bazy danych, nie tylko nie rozwiąże problemu, ale wręcz pogorszy Twoją sytuację, ponieważ kod, który będziesz musiał zmodyfikować zostanie dodatkowo rozproszony pomiędzy warstwy. Tak czy inaczej, jeśli do budowy warstwy środkowej wykorzystasz automatyczny generator kodu (co w dzisiejszych czasach jest zjawiskiem powszechnym), to przydatność tej warstwy z punktu widzenia osób odpowiedzialnych za konserwację kodu będzie minimalna.

Całe to zamieszanie związane z koniecznością ręcznego modyfikowania wszystkich okien przynajmniej dla mnie stanowi wystarczający argument, by nie stosować tego typu rozwiązań. Jakiekolwiek oszczędności czasowe wynikające ze stosowania generatora kodu prędzej czy później zostaną zrównoważone przez znacznie wyższe koszty konserwacji systemu.

Systemy tego typu przyciągają programistów przede wszystkim tym, że są konstruowane za pomocą znanych im technik. Generatory kodu ułatwiają programowanie w nieznanym języku obiektowym, oferując przyjazne lub wręcz intuicyjne mechanizmy proceduralne.

Podejście, które można streścić zdaniem: „potrafię programować w języku FORTRAN, stosując dowolny inny język", w praktyce wyklucza możliwość osiągania rzeczywistych korzyści w zakresie konserwacji systemów obiektowych. Uważam, że używanie Javy kompletnie mija się z celem, jeśli nie implementuje się naprawdę obiektowego projektu. Java jest prosta, ale tylko w porównaniu z językiem C++. Jeśli chcesz pisać systemy proceduralne, lepiej od razu wybierz któryś z naprawdę prostych języków proceduralnych. (Nie zgadzam się z wieloma zwolennikami Javy, którzy twierdzą, że takie mechanizmy jak bezpieczeństwo typów, dynamiczne wczytywanie itp. usprawiedliwiają pisanie w tym języku kodu proceduralnego).

Z drugiej strony, jeśli *rzeczywiście* realizujesz projekt obiektowy, stosowanie języka zaprojektowanego specjalnie z myślą o tego typu systemach (a więc np. Javy) może znacznie ułatwić proces implementacji. Wielu programistów języka C próbuje programować w Javie zupełnie tak, jakby pracowali w C i implementuje w języku Java systemy proceduralne zamiast systemów obiektowych (dla których ten język stworzono). Do takich praktyk skłania programistów sam język programowania, którego składnia niestety imituje składnię języków C i C++, włącznie z takimi usterkami jak nieuporządkowany system poprzedzania operatorów bitowych. Java trochę łagodzi negatywne skutki takiego podejścia, ponieważ wykazuje więcej cech języka „czysto" obiektowego niż np. C++. Ominięcie mechanizmów obiektowych jest w tym języku dużo trudniejsze, jeśli w ogóle możliwe. Zdeterminowany programista potrafi jednak stworzyć fatalny kod w każdym języku.

Jak zatem należy to robić „prawidłowo"?

Ponieważ obiektowe podejście do projektowania i programowania systemów informatycznych jest niezbędne, ale też po prostu nieznane, przeanalizujmy jeszcze jeden przykład złego (i dobrego) sposobu budowania tego typu systemów z perspektywy projektowania obiektowego. Na potrzeby tego przykładu posłużę się terminalami ATM (podobnie jak autorzy wielu innych książek) nie dlatego, że ktokolwiek implementuje jeszcze systemy ATM, tylko dlatego, że ATM doskonale nadaje się do analizy zarówno technik obiektowych, jak i architektur klient-serwer. Przyjmijmy, że mamy centralny komputer banku (w roli obiektu serwera) oraz terminal ATM (w roli obiektu klienta).

Większość programistów proceduralnych baz danych będzie traktowała taki serwer jak repozytorium z danymi, natomiast tego typu klient będzie postrzegany wyłącznie jak źródło żądań tych danych. Programiści podzielający ten pogląd prawdopodobnie rozwiążą problem transakcji ATM w następujący sposób:

1. Użytkownik podchodzi do terminala, wkłada kartę i wpisuje swój kod PIN.
2. Terminal ATM formułuje zapytanie w postaci: „podaj mi kod PIN przypisany do tej karty", wysyła to zapytanie do centralnej bazy danych i sprawdza, czy zwrócona wartość jest zgodna z danymi podanymi przez użytkownika. Terminal ATM wysyła kod PIN do serwera w formie łańcucha (będącego częścią zapytania języka SQL), ale zwracana liczba jest składowana w postaci 16-bitowej liczby całkowitej, która ułatwia operację porównania.
3. Użytkownik żąda operacji wypłaty środków.

4. Terminal ATM formułuje kolejne zapytanie; tym razem ma ono postać: „podaj mi stan konta tego użytkownika". Po otrzymaniu odpowiedzi zwrócone saldo jest umieszczane w odpowiednio przeskalowanej 32-bitowej liczbie całkowitej.
5. Jeśli ilość środków na koncie jest wystarczająca, terminal wydaje gotówkę i wysyła do serwera następujący komunikat: „zaktualizuj stan konta tego użytkownika".

(Nawiasem mówiąc, rzeczywisty sposób funkcjonowania bankomatów jest inny.)

Jakie są wady takiego podejścia do problemu działania terminali ATM? Rozpocznijmy naszą analizę od zwracanego stanu konta. Co będzie, jeśli Bill Gates uda się do banku, zechce otworzyć konto czekowe i wpłaci tam wszystkie swoje pieniądze? Nie chcesz przecież odesłać go z kwitkiem, ale masz świadomość, że według ostatnich szacunków jego majątek to jakieś 100 gigadolarów. Niestety, wykorzystywana do składowania stanu konta 32-bitowa liczba całkowita może reprezentować najwyżej 20 megadolarów (4 gigadolary podzielone najpierw przez 2 w związku z koniecznością reprezentowania znaku oraz dalej podzielone przez 100 w związku z potrzebą reprezentowania centów). Podobnie, 16-bitowa liczba całkowita używana do reprezentowania kodów PIN może zawierać co najwyżej 4 cyfry dziesiętne. Co będzie, jeśli Bill zażyczy sobie kodu *GATES* (pięciocyfrowego)? Ostatnim problemem są formułowane przez terminal ATM zapytania do centralnej bazy danych. Jeśli wykorzystywany słownik danych ulegnie zmianie (jeśli np. zmodyfikujemy pole nazwiska), wszystkie te zapytania języka SQL po prostu przestaną działać. (Mimo że przedstawiony przykład jest z oczywistych względów nonsensowny, warto choć przez chwilę pomyśleć o komplikacjach związanych z rezygnacją z włoskich lirów na rzecz wspólnej waluty europejskiej).

Proceduralne rozwiązanie wszystkich tych problemów wiąże się z koniecznością takiego zmodyfikowania pamięci ROM wszystkich terminali ATM na świecie (przecież nigdy nie wiadomo, z którego Bill zechce skorzystać), aby obsługiwały 64-bitowe liczby zmiennoprzecinkowe podwójnej precyzji zamiast 32-bitowych liczb całkowitych (aby umożliwić przechowywanie wyjątkowego stanu konta Billa) oraz 32-bitowych liczb całkowitych do przechowywania 5-cyfrowych kodów PIN. Będzie to oczywiście poważny problem dla organizacji, która odpowiada za konserwację tych terminali.

Wróćmy na chwilę do problemów rzeczywistego świata, w którym koszt wdrażania oprogramowania jest jednym z najważniejszych składników budżetów działów IT. W architekturach klient-serwer odpowiednikiem „zmodyfikowania pamięci ROM wszystkich terminali ATM" jest wdrożenie nowej wersji aplikacji klienta, co w większości przypadków wymaga *ogromnych* nakładów. Podobne problemy konserwacyjne ujawniają się w większości programów proceduralnych, nawet tych, które nie korzystają z baz danych. Zmiana definicji kilku centralnych typów danych lub zmiennych globalnych (odpowiadających w tym przypadku centralnemu słownikowi danych) może wymagać ponownego napisania wszystkich funkcji i procedur programu. Właśnie tego rodzaju koszmarów konserwacji i rozwoju oprogramowania można uniknąć dzięki technikom obiektowym.

Aby przekonać się, jak podejście obiektowe może eliminować tego typu problemy, spróbujmy przeanalizować możliwe rozwiązanie wspomnianego już przykładu terminali

ATM z wykorzystaniem technik obiektowych, a więc traktując cały system jak zbiór współpracujących obiektów, które oferują określone zdolności (możliwość wykonywania pewnych czynności). Pierwszym krokiem każdego procesu projektowania obiektowego jest sformułowanie „definicji problemu", czyli prezentacji problemu, który próbujemy całościowo rozwiązać w ramach czegoś, co nazywa się często „dziedziną problemu". W tym przypadku dziedziną problemu jest bankowość. Definicja problemu opisuje sam *problem*, nie program komputerowy. Analizowany problem można opisać w następujący sposób:

> *Klient przychodzi do banku, bierze od kasjera w okienku blankiet potwierdzenia wypłaty i wypełnia go, wpisując numer rachunku i kwotę wypłaty. Klient wraca do kasjera, potwierdza swoją tożsamość i przekazuje wypełniony dokument. (Kasjer weryfikuje tożsamość klienta, sprawdzając zapisy rejestru bankowego). Następnie kasjer zwraca się o odpowiednie upoważnienie do urzędnika bankowego i wypłaca pieniądze klientowi.*

Dysponując nawet tak prostą definicją problemu, możesz bez trudu zidentyfikować kilka potencjalnych „kluczowych abstrakcji" (klas) wraz z odpowiednimi operacjami (patrz tabela 1.2). W tym celu posłużę się formatem karty CRC stworzonym przez Warda Cunninghama (który szczegółowo omówię w dalszej części tego rozdziału).

Tabela 1.2. *Udziałowcy przypadków użycia wymienieni w formacie karty CRC*

Klasa	Odpowiedzialność	Obiekty współpracujące
Rejestry bankowe	Tworzy potwierdzenia wypłaty. Sprawdza, czy klient jest tym, za kogo się podaje	Kasjer: żąda blankietu potwierdzenia wypłaty
Urzędnik bankowy	Upoważnia kasjera do wypłaty gotówki	Kasjer: żąda upoważnienia
Potwierdzenie wypłaty	Rejestruje ilość gotówki żądanej przez kasjera	Rejestry bankowe: tworzą potwierdzenia Urzędnik bankowy: autoryzuje wypłatę Kasjer: przedstawia potwierdzenie klientowi
Kasjer	Pobiera potwierdzenie wysokości depozytu z rejestru bankowego i przekazuje je urzędnikowi bankowemu do zatwierdzenia	Rejestry bankowe: tworzą potwierdzenia wysokości depozytów Urzędnik bankowy: autoryzuje transakcję

W przedstawionym modelu serwerem jest tak naprawdę obiekt urzędnika bankowego, którego zasadniczą rolą jest autoryzowanie (zatwierdzanie) transakcji proponowanych przez kasjera. Bank, który także powinien mieć postać obiektu działającego po stronie serwera, na żądanie kasjera tworzy blankiety potwierdzeń wypłaty. Strona klienta jest reprezentowana przez obiekt kasjera, którego główną rolą jest pobieranie z obiektu banku blankietów dokumentów i przekazywanie ich bezpośrednio klientowi. Co ciekawe, klient (Bill) jest dla tego systemu elementem zewnętrznym i dlatego nie został uwzględniony w przedstawionym modelu. (Banki z pewnością mają klientów, ale klient nie jest już atrybutem banku, tak jak nie są jego atrybutami usługi portierów. Atrybutami banku z pewnością mogłyby być rachunki klientów, ale nie sami klienci. Przykładowo,

zapewne nie definiujesz się jako część banku, którego jesteś klientem). Obiektowy system terminali ATM modeluje tylko przedstawioną wcześniej definicję problemu. Oto przewidywany przepływ komunikatów:

1. Bill podchodzi do terminala ATM, wkłada swoją kartę, podaje kod PIN i wydaje dyspozycję wypłaty gotówki.
2. Obiekt `Kasjer` wysyła do działającego po stronie serwera obiektu `RejestryBankowe` następujące zapytanie: „Czy ta karta i podany kod PIN są prawidłowe?"
3. Obiekt `RejestryBankowe` zwraca odpowiedź *tak* lub *nie*.
4. Obiekt `Kasjer` prosi obiekt `RejestryBankowe` o pusty obiekt `PotwierdzenieWypłaty`. Żądany obiekt będzie egzemplarzem pewnej klasy implementującej interfejs `PotwierdzenieWypłaty` i zostanie przekazany przez obiekt `RejestryBankowe` obiektowi `Kasjer` *przez wartość*, za pośrednictwem mechanizmu zdalnego wywoływania metod (ang. *Remote Method Invocation — RMI*). To bardzo ważne. Wiedza obiektu `Kasjer` o obiekcie `PotwierdzenieWypłaty` sprowadza się do znajomości implementowanego przez ten obiekt interfejsu — implementacja (plik *.class*) dociera do obiektu `Kasjer` wraz z samym obiektem, zatem `Kasjer` nie ma możliwości określenia, jak otrzymany obiekt będzie przetwarzał przekazywane mu komunikaty. Taka abstrakcja jest o tyle *korzystna*, że ewentualne zmiany funkcjonowania obiektu `PotwierdzenieWypłaty` nie będą wymuszały zmian w definicji obiektu `Kasjer`.
5. Obiekt `Kasjer` żąda od obiektu `PotwierdzenieWypłaty` wyświetlenia interfejsu użytkownika. (Obiekt `PotwierdzenieWypłaty` realizuje to zadanie, generując interfejs na ekranie terminala ATM na podstawie podsystemu graficznego AWT).
6. Bill wypełnia potwierdzenie wypłaty.
7. Obiekt `Kasjer` rejestruje zakończenie operacji uwierzytelniania klienta (być może metodą monitorowania przycisku *Akceptuj* na klawiaturze terminala) i przekazuje wypełniony obiekt `PotwierdzenieWypłaty` działającemu po stronie serwera obiektowi `UrzędnikBankowy` (także przez wartość, za pośrednictwem mechanizmu RMI) w formie argumentu przesyłanego komunikatu: „Czy mam upoważnienie do wypłaty tak dużej ilości gotówki?"
8. Obiekt `UrzędnikBankowy` zwraca odpowiedź *tak* lub *nie*.
9. Jeśli terminal ATM otrzymał odpowiedź *tak*, gotówka jest natychmiast wypłacana klientowi. (Dla uproszczenia nie będę już analizował tego procesu).

Nie jest to oczywiście jedyne (ani nawet najlepsze) rozwiązanie problemu, ale — uwierz mi — jest pod wieloma względami lepsze od wcześniejszego rozwiązania proceduralnego.

Ważną cechą przedstawionego protokołu, na którą warto zwrócić uwagę, jest to, że cała wiedza na temat sposobu składowania stanu rachunku i kodu PIN, a więc między innymi na temat procedur podejmowania decyzji o możliwości wypłacenia pieniędzy, jest ukryta wewnątrz różnych obiektów. Takie rozwiązanie jest możliwe, ponieważ serwer jest teraz obiektem, który implementuje funkcję „autoryzacji". Zamiast żądać danych niezbędnych do autoryzowania transakcji, obiekt `Kasjer` zleca wykonanie tego zadania (działającemu

po stronie serwera) obiektowi `UrzędnikBankowy`, który dysponuje wszystkimi potrzebnymi informacjami. Żadne dane (ani stan rachunku, ani kod PIN) nie są odsyłane do terminala ATM, zatem ewentualne zmiany kodu serwera nie będą się wiązały z koniecznością modyfikowania kodu tego terminala.

Warto też zwrócić uwagę na fakt, iż obiekt `Kasjer` nie „wie" nawet, jak w całym systemie reprezentowane są pieniądze. Oznacza to, że żądana kwota jest w całości hermetyzowana w ramach obiektu `PotwierdzenieWypłaty`. W efekcie, ewentualne zmiany mechanizmu reprezentowania kwot pieniężnych po stronie serwera są całkowicie transparentne z perspektywy działającego po stronie klienta obiektu `Kasjer`. Osoba odpowiedzialna za utrzymywanie tego systemu będzie mogła spokojnie drzemać w swoim biurze zamiast jeździć po całym świecie i modyfikować pamięci ROM wszystkich terminali ATM.

Gdyby europejskie terminale były implementowane w ten sposób, przejście na wspólną walutę — euro — byłoby banalnie proste — wymagałoby tylko zmiany definicji klasy `PotwierdzenieWypłaty` (lub klasy `Pieniądze`) po stronie serwera. Wszelkie dalsze żądania dotyczące obiektów `PotwierdzenieWypłaty` generowane przez terminale ATM powinny skutkować zwracaniem odpowiedzi dostosowanych do nowej sytuacji.

Automat komórkowy

Rozszerzmy teraz nasze wyobrażenie koncepcji obiektowej w taki sposób, aby obejmowało pojęcie interfejsu — posłużę się kolejnym przykładem, który wybrukuje drogę do zrozumienia programu *Gra w życie* (ang. *The Game of Life*), z którego będziemy korzystać w dalszej części tej książki.

Dobrym przykładem naturalnego systemu obiektowego jest klasa programów nazywana *automatem komórkowym*. Tego typu programy rozwiązują skomplikowane problemy w sposób bardzo obiektowy — ogromny problem jest rozwiązywany przez zbiór małych, identycznych obiektów (komórek), z których każdy implementuje prosty zbiór reguł i komunikuje się ze swoimi bezpośrednimi sąsiadami. Poszczególne komórki nie mają co prawda żadnej wiedzy o większym problemie, ale komunikują się z innymi komórkami w taki sposób, że z ich perspektywy problem ten rozwiązuje się sam.

Typowym przykładem automatu komórkowego jest modelowanie ruchu ulicznego (którego omówienie znacznie wykraczałoby poza zakres tematyczny tej książki). Problem przewidywania ruchu ulicznego jest wyjątkowo trudny — jest klasycznym problemem teorii chaosu. Tak czy inaczej, warto pamiętać, że można modelować ruch pojazdów w taki sposób, by obserwacja jego symulacji umożliwiała przewidywanie pewnych zdarzeń wyłącznie na podstawie zachowania badanego modelu. Przewidywanie i symulowanie ruchu ulicznego to oczywiście dwa różne problemy — automaty komórkowe sprawdzają się doskonale właśnie w symulowaniu z pozoru chaotycznych procesów.

Poświęcę kilka kolejnych stron na omawianie problemu ruchu ulicznego nie tylko dlatego, że problem ten tak dobrze demonstruje mechanizm automatów komórkowych, ale także dlatego, że przykład ten doskonale ilustruje wiele podstawowych zagadnień

związanych z projektowaniem obiektowym, co do których chciałbym mieć pewność, że są zrozumiałe, zanim jeszcze przystąpię do analizy takich systemów obiektowych jak *Gra w życie*.

Większość programów działa przez implementowanie pewnych algorytmów — pojedynczych (choć w wielu przypadkach skomplikowanych) wzorów, które pracują w dobrze udokumentowany, przewidywalny sposób pod warunkiem, że otrzymają znany zbiór danych wejściowych. Jakiekolwiek rozwiązanie próbujące modelować ruch uliczny na poziomie całego miasta, opierając się na pojedynczym (skomplikowanym) algorytmie jest po prostu zbyt trudne do zaimplementowania. Podobnie jak w przypadku większości problemów teorii chaosu, najzwyczajniej w świecie nie wiadomo, jak należy pisać algorytmy „rozwiązujące" problem ruchu ulicznego.

Automat komórkowy radzi sobie z tym problemem, omijając go. W tego typu rozwiązaniach nie stosuje się algorytmów jako takich, a jedynie pewne mechanizmy modelowania zachowań tych części systemu, które można stosunkowo łatwo śledzić. Przykładowo, zamiast modelować ruch uliczny w całym mieście, automat komórkowy rozbija pełną sieć ulic na mniejsze fragmenty i skupia się na ich modelowaniu. Każdy odcinek drogi może się komunikować z sąsiadującymi odcinkami, ale żaden z tych odcinków nie ma wiedzy na temat całej sieci ulic w danym mieście.

Modelowanie zachowań na niewielkim wycinku ulicy jest stosunkowo łatwe. Każdy taki fragment ma określoną pojemność (wyrażającą się zwykle liczbą pasów ruchu) i inne, równie proste atrybuty. Fragment drogi ma oczywiście swoją długość oraz prędkość maksymalną zależną od pojemności i stosowanych na tym odcinku ograniczeń. To wszystko. Samochody pojawiają się na jednym końcu odcinka i po jakimś czasie opuszczają go na drugim końcu. Do wykończenia systemu będziemy jeszcze potrzebowali dwóch dodatkowych obiektów (reprezentujących samochód i mapę), które także charakteryzują się łatwym do modelowania zachowaniem. (Oba dodatkowe obiekty omówię dokładniej za chwilę).

Rozmaite obiekty w ramach tego systemu muszą się wzajemnie komunikować za pośrednictwem precyzyjnie zdefiniowanych interfejsów. (Na rysunku 1.2 przedstawiono cały proces konwersacji, który za chwilę omówię).

Interfejs `Road` składa się z dwóch metod:

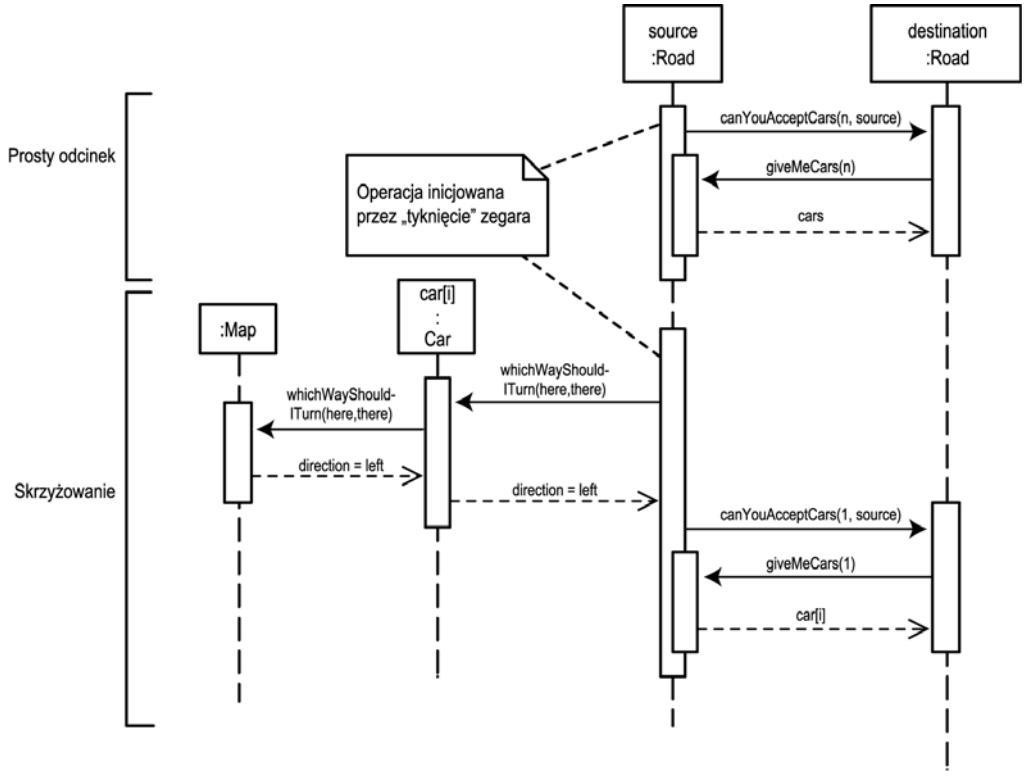
1. Czy możesz przyjąć *N* samochodów?

```
boolean canYouAcceptCars(int n, Road fromThisRoad)
```

2. Przekaż mi *N* samochodów.

```
car[] giveMeCars(int n)
```

Sąsiadujące odcinki drogi komunikują się w bardzo prosty sposób. Kiedy bieżący odcinek decyduje, że musi się pozbyć kilku samochodów, pyta odcinek sąsiadujący, czy ten może je przyjąć (pierwszy komunikat). Zapytany odcinek może te samochody przyjąć — prosi o ich przekazanie sąsiedni odcinek, który zainicjował konwersację (drugi komunikat).



Rysunek 1.2. *Diagram sekwencji UML dla modelu ruchu ulicznego*

Z podobną komunikacją dwustronną będziemy mieli do czynienia w rozdziale 3. Początkowe żądanie musi zawierać referencję do generującego je obiektu `Road`, którą docelowy obiekt `Road` będzie mógł wykorzystać podczas żądania kolejnych samochodów; w przeciwnym razie obiekt docelowy nie wiedziałby, który odcinek źródłowy nadesłał dane żądanie. Odcinek (fragment drogi) w środku takiego bloku komunikuje się z dwoma sąsiadami (z dwoma przylegającymi odcinkami drogi, obiektami `Road`), obiekt reprezentujący skrzyżowanie ma czterech bezpośrednich sąsiadów itd. (Odpowiednie połączenia są ustawiane w momencie tworzenia sieci ulic i mogą być implementowane w formie argumentów konstruktora).

Odcinek drogi (obiekt `Road`) musi wewnętrznie wykorzystywać kilka reguł, na bazie których będzie decydował, kiedy może przyjąć kolejne samochody. Przykładowo, średnia efektywna prędkość samochodu (obiektu `Car`), czyli różnica pomiędzy momentem wjazdu samochodu na reprezentowany odcinek a momentem opuszczenia tego odcinka, może być funkcją natężenia ruchu — liczby samochodów przebywających na danym odcinku. Różne typy dróg (autostrady, uliczki itp.) mogą implementować te reguły w zróżnicowany sposób. Reguły te są jednak znane tylko obiektom `Road`. Podobnie jak w przypadku wszystkich innych systemów obiektowych, tego typu reguły mogą być zasadniczo zmieniane bez wpływu na otaczający kod, ponieważ modyfikacje reguł nie będą wpływały na kształt interfejsu odcinka drogi (obiektu `Road`).

Następnym potrzebnym elementem jest obiekt reprezentujący samochód (`Car`). Każdy obiekt `Road` pełni funkcję strażnika swoich obiektów `Car`. Ponieważ ograniczenie prędkości i długość odcinka drogi są atrybutami obiektu `Road`, obiekt ten może łatwo określić, jak długo będzie gościł poszczególne samochody bez konieczności komunikowania się z odpowiednimi obiektami `Car`.

Pewnym utrudnieniem są skrzyżowania. Obiekt `Road` musi przecież „wiedzieć", do którego ze swoich sąsiadów skierować dany obiekt `Car`. Rozwiązaniem tego problemu jest drugi, równie prosty interfejs (implementowany przez obiekt `Car` i wykorzystywany przez obiekt `Road`).

1. Jesteś *tutaj*; w którą stronę chcesz teraz jechać?

```
Direction whichWayWouldYouLikeToTurn(Location here)
```

Także w tym przypadku obiektu `Road` nie „interesuje" sposób generowania odpowiedzi przez obiekt `Car` na tak postawione pytanie — ważne jest tylko uzyskanie odpowiedniej odpowiedzi. W fazie diagnozowania kodu można wymusić na metodzie obsługującej ten komunikat wyświetlanie zapytania na konsoli i wykonywanie wpisywanych poleceń. Rzeczywisty system będzie oczywiście wymagał zautomatyzowanego rozwiązania, ale zmiana ręcznego funkcjonowania klasy `Car` na działanie automatyczne będzie wymagała zmodyfikowania tylko tej jednej klasy — nie będzie to miało wpływu na resztę systemu.

Warto zwrócić uwagę na fakt, że obiekt `Car` nie musi dokładnie „wiedzieć", gdzie się znajduje (przypomina to trochę rzeczywistą sytuację, którą modelujemy). Inaczej jest w przypadku obiektu `Road`, który zna swoje położenie (obiekt `Location`), zatem to właśnie obiekt `Road` przekazuje obiekt `Location` obiektowi `Car`. Ponieważ obiekt `Location` stale się zmienia, obiekt `Car` nawet nie próbuje tego obiektu wewnętrznie utrwalać. Obiekt `Car` musi zawierać tylko jeden atrybut reprezentujący cel podróży.

Obiekt `Car` musi dysponować mechanizmem, który umożliwi mu odpowiadanie na pytanie o kierunek jazdy, powinniśmy więc stworzyć jeszcze jeden obiekt: `Map`. Obiekt ten będzie implementował jeszcze jeden interfejs przesyłania komunikatów.

2. Jestem *tutaj* i chcę jechać *tam*; w którym kierunku powinienem się udać?

```
Direction whichWayShouldITurn(Location here, Location there)
```

Obiekt `Car`, ponownie, nie ma wiedzy na temat techniki udzielania odpowiedzi na zadane pytanie przez obiekt mapy — wystarcza mu samo uzyskiwanie potrzebnych informacji. (Problem wyboru trasy jest najtrudniejszą częścią tego systemu, ale został już rozwiązany przez wszystkie dostępne na rynku urządzenia nawigacyjne oparte na systemie GPS. Oznacza to, że odpowiednie rozwiązanie można po prostu kupić). Warto zwrócić uwagę na sposób przekazywania położenia (na podstawie obiektu `Map`) samochodu do reprezentującego go obiektu `Car`. Proces ten, nazywany *delegacją*, jest typowym rozwiązaniem w systemach obiektowych. Obiekt rozwiązuje pewien problem, delegując go do zawieranego przez siebie obiektu i przekazując tam wszelkie niezbędne informacje z zewnątrz. Komunikaty przekazywane przez obiekt delegujący do obiektu delegowanego są zwykle opatrywane dodatkowymi argumentami.

Ostatnim elementem tej układanki jest określenie sposobu, w jaki samochody trafiają na drogę (obiekt `Road`). Z punktu widzenia systemu modelującego ruch uliczny, prawdziwym końcem drogi jest dom kierowcy (nazywany *wjazdem*). Podobnie, miejsce pracy kierowcy także może stanowić obiekt `Road` (nazwany *parkingiem*). Obiekty domu i miejsca pracy muszą nie tylko implementować interfejs `Road`, ale także „znać" odcinki drogi, z którymi te obiekty są połączone. Przydomowy garaż i parking w miejscu pracy powinny też „wiedzieć", jak wypuszczać samochody do systemu i akceptować ich przyjmowanie o określonych porach dnia — implementacja tego elementu będzie prosta, ponieważ będzie wymagała jedynie użycia odpowiedniego interfejsu obiektów `Road`.

Dodajmy teraz interfejs użytkownika. Klasyczne wymaganie odnośnie do systemów obiektowych mówi, że obiekt nie powinien ujawniać szczegółów swojej implementacji. Naszym celem jest maksymalizacja możliwości konserwacji tworzonego oprogramowania. Jeśli wszystkie informacje na temat implementacji są zazdrośnie strzeżonym sekretem obiektu, ewentualne zmiany tej implementacji nie będą miały wpływu na funkcjonowanie kodu wykorzystującego dany obiekt. Oznacza to, że modyfikacja implementacji pojedynczego obiektu nie „wywraca" reszty systemu. Ponieważ wszystkie zmiany koncentrują się zwykle wokół definicji pojedynczej klasy, konserwacja systemów obiektowych jest wyjątkowo łatwa, ale tylko wtedy, gdy zachowana jest zasada hermetyczności. (W niektórych sytuacjach mogą istnieć istotne przesłanki do rezygnacji z tej reguły, jednak decyzja w tym zakresie powinna być podejmowana z uwzględnieniem przyszłych utrudnień w konserwacji systemu).

Wymaganie hermetyczności oznacza, że dobrze zaprojektowany obiekt będzie przynajmniej w jakimś stopniu odpowiadał za tworzenie własnego wycinka interfejsu użytkownika. Oznacza to, że dobrze zaimplementowana klasa nie będzie udostępniała metod zwracających i ustawiających jej pola, ponieważ takie metody stanowiłyby furtkę do szczegółów implementacji, a więc w konsekwencji prowadziły do poważnych problemów w zakresie konserwacji systemu zbudowanego na ich bazie. Jeśli implementacja obiektu zostanie zmodyfikowana w taki sposób, że niezbędna będzie zmiana np. typu lub zakresu wartości zwracanych przez tego typu metodę, będziesz musiał przebudować nie tylko obiekt definiujący tę metodę, ale także cały kod, który z niej pośrednio lub bezpośrednio korzysta. Więcej uwagi temu zagadnieniu oraz technikom projektowania systemów bez metod zwracających i ustawiających poświęcę za chwilę.

W bieżącym systemie możemy zbudować interfejs użytkownika, dodając pojedynczą metodę do interfejsu `Road`.

3. Narysuj swoją reprezentację wzdłuż tej *linii*:

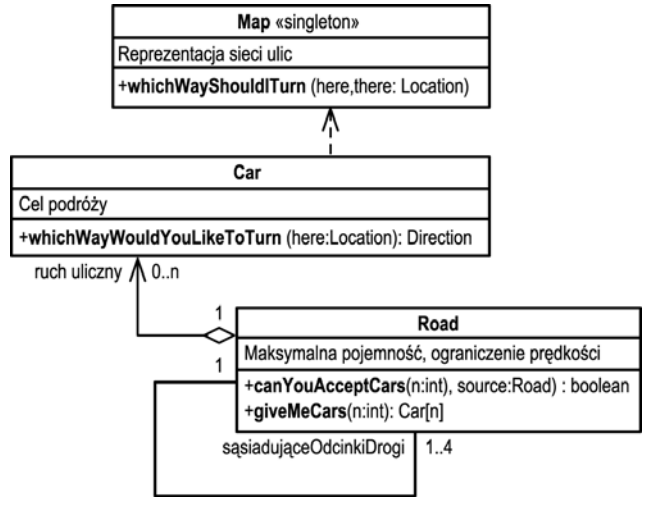
```
drawYourself(Graphics g, Point begin, Point end);
```

Interfejs użytkownika obiektów `Road` wyświetla średnią prędkość samochodów na reprezentowanym odcinku drogi (który może się różnić w zależności od natężenia ruchu) za pomocą zmieniających się kolorów rysowanych linii. W wyniku zastosowania tego mechanizmu powinniśmy otrzymać plan miasta, na którym kolory ulic będą wskazywały na przewidywaną prędkość pokonywania poszczególnych odcinków. Obiekt `Map` musi oczywiście „wiedzieć" o wszystkich obiektach `Road`, aby właściwie kierować procesem wizualizacji planu i w razie konieczności delegować żądania rysowania do poszczególnych obiektów `Road`. Ponieważ obiekty te same odpowiadają za swoją wizualizację, nie

ma potrzeby tworzenia mnóstwa metod zwracających i proszenia ich o informacje niezbędne dla zewnętrznych generatorów interfejsu użytkownika — przykładowo, taka metoda jak `getAverageSpeed()` będzie zupełnie zbędna.

Teraz, kiedy już wykonaliśmy podstawowe zadania, możemy wprawić nasz system w ruch. Musimy wiązać ze sobą wszystkie drogi, wjazdy i parkingi za każdym razem, gdy kompilujemy nasz system. Musimy też umieścić w tym systemie kilka pojazdów (także w czasie kompilacji) i rozpocząć obserwację przygotowanego modelu. Z każdym „tyknięciem" zegara wszystkie odcinki dróg decydują o liczbie samochodów, których muszą się pozbyć, i przekazują je do sąsiednich odcinków. Każdy odcinek (obiekt Road) automatycznie aktualizuje swój wycinek interfejsu użytkownika, aby odzwierciedlał obserwowane zmiany średniej prędkości. Voilà! Mamy model ruchu ulicznego.

Kiedy już zaprojektujesz system przesyłania komunikatów, staniesz przed koniecznością wyciągnięcia właściwych wniosków z diagramu modelu statycznego. Ewentualne powiązania występują wyłącznie pomiędzy klasami, których obiekty wzajemnie się komunikują — definiowane są tylko naprawdę potrzebne komunikaty. Na rysunku 1.3 przedstawiono odpowiedni diagram UML. Warto pamiętać, że rozpoczęcie projektowania od tworzenia i analizy takiego diagramu byłoby stratą czasu. Zanim zrozumiesz relacje pomiędzy klasami, musisz przecież wiedzieć, jaki powinien być przepływ komunikatów.



Rysunek 1.3. *Diagram modelu statycznego UML dla systemu modelującego ruch uliczny*

Jeśli chcesz zdobyć doświadczenie w zakresie eksperymentowania z tego typu symulatorami ruchu ulicznego, powinieneś się przyjrzeć popularnej grze SimCity firmy Maxis Software. Ponieważ nie widziałem kodu źródłowego tego programu, tak naprawdę nie wiem, czy SimCity rzeczywiście *zaimplementowano* w formie automatu komórkowego, byłbym jednak zdumiony, gdyby okazało się, że jest inaczej. Pewne jest, że na poziomie interfejsu użytkownika gra SimCity działa podobnie jak nasz system modelowania ruchu ulicznego. Maxis oferuje darmową wersję tego produktu (zatytułowaną SimCity Classic) na swojej witrynie internetowej (patrz *http://www.maxis.com*).

Metody zwracające i ustawiające są złe

Jak już wspomniałem, fundamentalną regułą rządzącą systemami obiektowymi jest ukrywanie przez obiekty ich szczegółów implementacyjnych. Stosowanie tej zasady umożliwia modyfikowanie implementacji obiektów bez konieczności wprowadzania zmian w kodzie, który te obiekty *wykorzystuje*. Oznacza to, że powinieneś unikać stosowania funkcji zwracających i ustawiających, które z jednej strony niczego nie ułatwiają, a jednocześnie zapewniają dostęp do szczegółów implementacyjnych (pól) w systemach obiektowych. Warto zwrócić uwagę na fakt, iż ani przykład systemu terminali ATM, ani przykład modelu ruchu ulicznego nie wykorzystywał metod zwracających i ustawiających pola obiektów.

Nie twierdzę przy tym, że Twoje funkcje nigdy nie powinny zwracać wartości, lub że funkcji `get` i `set` nigdy nie należy stosować. Obiekty czasami po prostu muszą przekazywać swoje dane, aby cały system mógł funkcjonować prawidłowo. Tak czy inaczej, funkcje `get` i `set` często są wykorzystywane w sposób niewłaściwy, a więc w roli środków zapewniających dostęp do pól, które w przeciwnym razie byłyby prywatne, zatem ich używanie może prowadzić do poważnych utrudnień. To, co rozumiem przez właściwe zastosowania tych metod omówię na końcu tego podrozdziału. Obecność metod zwracających i ustawiających (nazywanych często *akcesorami* i *mutatorami*, choć zdarza się, że słowo *akcesor* jest używane w odniesieniu do obu tych metod) zwykle oznacza brak jasności oraz niewłaściwe podejście do rozwiązywanego problemu. Programiści często umieszczają te metody w definicjach klas, ponieważ najzwyczajniej w świecie chcą uniknąć myślenia o sposobie komunikowania się obiektów klas w czasie wykonywania systemu. Użycie metod zwracających pozwala odwlec moment analizy technik komunikacji do czasu właściwego kodowania. Takie podejście jest przejawem czystego lenistwa; z pewnością nie jest to oznaka „programowania dla elastyczności".

Przeanalizujmy banalny przykład, który dobrze pokazuje, dlaczego należy unikać metod zwracających. W Twoim programie może występować tysiąc wywołań metody `getX()`, z których każde zakłada, że wartość zwracana przez tę metodę należy do określonego typu. Wartość zwracana przez metodę `getX()` może być składowana np. w zmiennej lokalnej, zatem typ tej zmiennej musi odpowiadać typowi zwracanej wartości. Jeśli będziesz musiał zmienić implementację obiektu w taki sposób, że zmieni się typ zmiennej *`X`*, popadniesz w poważne tarapaty. Jeśli dotychczasowym typem zmiennej *`X`* był `int`, a nowym typem jest `long`, po wprowadzeniu odpowiedniej zmiany otrzymasz tysiąc błędów kompilacji. Jeśli rozwiążesz ten problem w sposób niewłaściwy, a więc zastosujesz operację rzutowania typu zwracanej wartości na typ `int`, kod będzie co prawda kompilowany, ale z pewnością nie będzie działał prawidłowo (zwracana wartość będzie obcinana). Aby uniknąć negatywnych skutków tej zmiany, będziesz musiał zmodyfikować kod otaczający każde z tysiąca wywołań metody `getX()`. Zdecydowanie nie chciałbym się znaleźć w podobnej sytuacji.

Przyjrzyjmy się teraz klasie `Money`. Ponieważ klasa ta była początkowo pisana wyłącznie z myślą o obsłudze dolarów amerykańskich, zdefiniowano w niej metodę `getValue()`, która zwraca liczbę zmiennoprzecinkową typu `double`, oraz `setValue()`, która ustawia nową wartość. Pierwszy problem polega na tym, że takie rozwiązanie umożliwia zupełnie nonsensowne działania na reprezentowanych kwotach pieniężnych, co ilustruje poniższy fragment kodu:

```
Money a, b, c;
// ...
a.setValue( b.getValue() * c.getValue() );
```

Co właściwie oznacza pomnożenie dwóch dolarów przez pięć dolarów?

Drugi problem jest jeszcze poważniejszy: może zaistnieć konieczność wprowadzenia do aplikacji rozwiązań międzynarodowych, a więc umożliwiających między innymi obsługę wielu różnych walut. Taka zmiana będzie wymagała dodania pola nazwanego `currency`, które będzie miało wewnętrznie przypisywane takie wartości jak `US_DOLLAR`, `YEN`, `LEU`, `ZLOTY` czy `HRYVNA`. Stosunkowo drobna zmiana — wielkie problemy. Co w tej sytuacji będzie zwracała metoda `getValue()`? Metoda ta nie może po prostu zwracać wartości typu double, ponieważ sama wartość niewiele Ci mówi. Musisz przecież wiedzieć, z jaką walutą masz do czynienia. Nie jest też możliwe normalizowanie zwracanej wartości, np. do dolarów, ponieważ przeliczniki stosowane na rynku walutowym zmieniają się co minutę. Warto się też zastanowić, co należałoby zrobić z otrzymaną wartością. Nie można ich przecież tak po prostu wyświetlić, ponieważ wymagałoby to opatrzenia ich symbolem odpowiedniej waluty. Możesz oczywiście do istniejącej metody `getValue()` dołączyć nową metodę `getCurrency()`, ale od tej chwili cały kod wykorzystujący tę wartość będzie musiał dodatkowo pobierać informację o walucie i lokalnie normalizować uzyskiwane wartości do standardowej waluty. Powielenie tego rozwiązania w tysiącu miejsc istniejącego kodu źródłowego wymagałoby mnóstwa pracy. Musiałbyś też znaleźć wszystkie okna swojego systemu, w których wyświetlane są kwoty pieniężne — logika odpowiednich elementów graficznych musiałaby zostać przebudowana do postaci obsługującej różne waluty. Ta „prosta" zmiana błyskawicznie staje się źródłem ogromnych komplikacji.

Oto kolejny przykład: przeanalizuj wszystkie problemy związane z polami `System.in`, `System.out` oraz `System.err` po wprowadzeniu do Javy klas `Reader` i `Writer`. Wszystkie trzy pola były publiczne, co samo w sobie było przekleństwem tego rozwiązania. Samo zastosowanie odpowiednich otoczek (np. metody `System.getOut()`, która zwracała pole `System.out`) oczywiście nie rozwiązywało problemu — pola `System.out` i `System.err` musiały być (opartymi na formacie Unicode) obiektami klasy `Writer`, nie (opartymi na formatowaniu bajtowym) obiektami klasy `PrintStream`. To samo dotyczyło pola `System.in` i klasy `Reader`. Zmiana zadeklarowanych typów obiektów zawierających pola `System.out` nie jest rozwiązaniem wystarczającym. Obiekty klasy `Writer` są wykorzystywane w nieco inny sposób niż strumienie wyjściowe. Oba mechanizmy różnią się przecież semantyką i udostępniają różne metody. W efekcie musisz więc zmienić (lub przynajmniej sprawdzić) cały kod otaczający, wykorzystujący pole `System.out` do wyświetlania na konsoli danych wyjściowych. Jeśli Twój program np. stosował formatowanie Unicode dla danych tekstowych wyświetlanych za pośrednictwem pola `System.out`, będziesz musiał użyć aż dwóch wywołań metody `write()` do zapisania na konsoli pojedynczego znaku. Co więcej, będziesz zmuszony do wprowadzenia do swojego programu dodatkowego kodu wyciągającego bardziej i mniej znaczące bajty znaków i wyświetlającego je osobno. Cały ten kod będzie jednak trzeba usunąć w wersji opartej na klasie `Writer`.

Opisany problem jest efektem siły przyzwyczajenia. Kiedy programiści proceduralni zaczynają korzystać z Javy, próbują budować kod, który będzie choć trochę przypominał ich dotychczasowe dokonania. Języki proceduralne nie oferują możliwości

wykorzystywania klas, ale zawierają zwykle rozwiązania podobne do struktur języka C (czyli w praktyce klasy bez metod i z samymi polami publicznymi). Dla takich programistów naśladowanie struktur języka C jest zupełnie naturalne — próbują budować definicje klas pozbawionych metod i udostępniających wyłącznie publiczne pola. Kiedy taki programista proceduralny usłyszy gdzieś, że należy stosować pola prywatne, odpowiednio zmienia ich deklaracje i tworzy publiczne metody `get` i `set`. Takie rozwiązanie nie jest jednak niczym więcej niż niepotrzebnym komplikowaniem publicznego dostępu do danych. Programiści stosujący tego typu działania z pewnością nie tworzą systemów obiektowych.

Programiści proceduralni będą argumentowali, że publiczne akcesory otaczające pola prywatne są o tyle „lepsze" od dostępnych bezpośrednio pól publicznych, że dają znacznie większą kontrolę nad operacjami modyfikowania wartości pól. Programista obiektowy stwierdzi natomiast, że każdy dostęp — kontrolowany czy nie — jest źródłem potencjalnych problemów konserwacyjnych. Dostęp kontrolowany może być co prawda lepszy od dostępu swobodnego, ale nie zmienia to faktu, że takie rozwiązanie jest z wielu względów niewłaściwe. Argument o wyższości akcesorów nad bezpośrednim dostępem w ogóle nie uwzględnia najważniejszej słabości obu rozwiązań — zdecydowana większość klas i tak nie potrzebuje metod akcesorów (ani mutatorów). Oznacza to, że dobrze zaprojektowany system przesyłania komunikatów (o sposobach jego projektowania za chwilę) w większości przypadków pozwala całkowicie wyeliminować metody `get` i `set` oraz w konsekwencji tworzyć klasy, których konserwacja będzie dużo łatwiejsza.

Nie twierdzę, że zwracanie wartości jest złe, lub że powinieneś wyeliminować ze swojego programu wszystkie metody `get` — to po prostu niemożliwe. Minimalizowanie udziału tego typu funkcji w definicjach klas spowoduje jednak znaczne ułatwienia w konserwacji kodu.

Z czysto praktycznej perspektywy, częste stosowanie metod `get` i `set` sprawia, że kod jest nie tylko bardziej skomplikowany, ale także mniej elastyczny. Przeanalizuj typową „wszechmogącą" klasę proceduralną, która z innych obiektów zbiera informacje niezbędne do wykonania jakiegoś zadania. Implementacja tej klasy pełna jest wywołań zewnętrznych metod `get`. Co jednak powinieneś zrobić, kiedy okaże się, że odpowiednie zadania są już wykonywane przez jeden z obiektów udostępniających swoje dane tą drogą? Czy można przenieść kod wykonujący właściwe zadania z jednej, „wszechmogącej" klasy w miejsca, w których składowane są niezbędne dane? Wywołania akcesorów przestają być potrzebne, a cały kod jest dużo prostszy.

Stosowanie metod `get` i `set` powoduje też, że program staje się nieelastyczny (nie można w jego ramach łatwo implementować nowych wymagań biznesowych) i wyjątkowo trudny w konserwacji. Prawdopodobnie najważniejszą zasadą systemów obiektowych jest *abstrakcja danych*, a więc ścisłe ukrywanie implementacji mechanizmów obsługi komunikatów przed innymi obiektami. To tylko jeden z powodów, dla których wszystkie Twoje zmienne klasowe (pola klasy niebędące stałymi) powinny być prywatne (deklarowane ze słowem kluczowym `private`). Jeśli stworzysz publiczną zmienną klasową, nie będziesz mógł zmieniać tego pola wraz z ewolucją całej klasy, ponieważ w ten sposób uniemożliwiłbyś prawidłowe funkcjonowanie kodu zewnętrznego, który z tego pola korzysta. Z pewnością nie masz ochoty na przeszukiwanie tysiąca zastosowań jakiejś klasy tylko dlatego, że wprowadziłeś drobną zmianę w jej definicji.

Bezmyślne stosowanie metod zwracających i ustawiających jest niebezpieczne z tych samych względów, które decydują o zagrożeniach związanych z używaniem pól publicznych — także metody `get` i `set` zapewniają zewnętrzny dostęp do szczegółów implementacyjnych. Co będzie, jeśli zostaniesz zmuszony do zmiany typu udostępnianego w ten sposób pola? Przecież będziesz wówczas musiał zmienić także typ zwracany przez metodę akcesora. Jeśli wartość ta jest używana w wielu miejscach, będziesz musiał zmienić odpowiednie fragmenty w całym kodzie. Ja wolałbym jednak ograniczyć zakres koniecznych zmian do definicji pojedynczej klasy. Nie chcę, by prosta modyfikacja przenosiła się na cały program.

Na podstawie reguły ukrywania szczegółów implementacji można stworzyć wiarygodny test systemów obiektowych. Czy możesz dokonywać istotnych zmian w definicji pojedynczej klasy (włącznie z wyrzuceniem całych fragmentów kodu i wstawieniem w ich miejsce zupełnie nowej implementacji) bez konieczności modyfikowania kodu wykorzystującego obiekty tej klasy? Tak głęboka modularyzacja oprogramowania bardzo ułatwia konserwację oprogramowania i pełni zasadniczą funkcję w ocenie jego obiektowości. Nieprzestrzeganie zasady ukrywania szczegółów implementacji bardzo ogranicza możliwość stosowania pozostałych mechanizmów obiektowych.

Ponieważ metody akcesorów naruszają regułę hermetyzacji, można bez trudu wykazać, że systemy, w których często lub niewłaściwie stosuje się tego typu rozwiązania po prostu nie są systemami obiektowymi. Co więcej, jeśli skrupulatnie przeprowadzisz proces projektowania (w przeciwieństwie do kodowania *ad hoc*), szybko stwierdzisz, że Twój program nie musi zawierać niemal żadnych metod akcesorów. Proces ten jest więc bardzo ważny.

Zapewne zauważyłeś, że w przedstawionym przykładzie modelowania ruchu ulicznego w ogóle nie stosowano metod zwracających i ustawiających. Obiekty klasy `Car` nie udostępniają metody `getSpeed()`, także obiekty klasy `Road` nie definiują metody `getAverageSpeed()`. Nie potrzebujemy metod `getLocation()` czy `setLocation()` w obiektach klasy `Car`, ponieważ składujemy informacje o lokalizacji pojazdów w obiektach klasy `Road` reprezentujących odcinki dróg, na których te pojazdy aktualnie przebywają. Nie potrzebujemy też metody `setAverageSpeed()` w obiektach klasy `Road`, ponieważ obiekty te same obliczają średnią prędkość pojazdów. Brak metod zwracających i ustawiających nie oznacza, że jakieś potrzebne dane nie mogą być przekazywane pomiędzy poszczególnymi modułami tego systemu — przykładowo, obiekt `Road` przekazuje informacje o lokalizacji do obiektów klasy `Car`. Tak czy inaczej, podczas projektowania systemów obiektowych należy możliwie głęboko minimalizować przepływy danych. Doskonałym papierkiem lakmusowym wskazującym na „słuszność" zastosowanych mechanizmów jest następująca reguła: **Nie proś obiektu o informacje, których potrzebujesz do wykonania jakiejś czynności; zamiast tego proś obiekt zawierający te informacje o wykonanie tych czynności za Ciebie.**

Przykładowo, nie powinieneś stosować przedstawionego wcześniej wyrażenia:

```
Money a, b, c;
// ...
a.setValue( b.getValue() * c.getValue() );
```

Zamiast tego zażądaj od obiektu klasy `Money` wykonania odpowiednich działań za Ciebie:

```
Money a, b, c;
// ...
a.increaseBy( b );
```

Nie mówisz już: „daj mi ten atrybut, abym mógł go wyświetlić". Teraz żądasz czegoś zupełnie innego: „daj mi możliwą do wyświetlenia wizualizację tego atrybutu" lub „wyświetl swoją zawartość".

Kolejnym wyznacznikiem realizacji tej reguły jest szczegółowość operacji. Operacje obszerne sprowadzają się do jednorazowego żądania od obiektów wykonania dużych zadań. Szczegółowe operacje żądają od obiektów realizacji stosunkowo niewielkich zadań. Ogólnie, wolę obszerne metody, ponieważ ich stosowanie upraszcza kod i w wielu przypadkach eliminuje konieczność stosowania metod zwracających i ustawiających.

Metody akcesorów i mutatorów można eliminować dopiero na etapie budowy modelu systemu, ponieważ bez dobrze przemyślanego modelu dynamicznego wiarygodne przewidywanie przyszłych mechanizmów komunikacji obiektów klas jest po prostu niemożliwe. Oznacza to, że w przypadku braku tego modelu musisz zapewniać jak najwięcej technik udostępniania danych, ponieważ nie jesteś w stanie przewidzieć, które z tych technik faktycznie będą konieczne. Taka strategia projektowania przez zgadywanie jest w najlepszym przypadku nieefektywna, ponieważ w praktyce oznacza stratę czasu na pisanie niepotrzebnych metod (lub dodawanie zbędnych mechanizmów do implementowanych klas). Jeśli postępujesz w myśl zasady, że w pierwszej kolejności należy budować model statyczny, musisz być przygotowany na to, że stracisz mnóstwo czasu na tworzenie nieprzydatnych lub zbyt elastycznych metod. Co więcej, jeśli niewłaściwy model statyczny wymusi zbyt duże nakłady na tworzenie niepotrzebnych rozwiązań, cały projekt może się zakończyć niepowodzeniem; a jeśli mimo to zdecydujesz się na jego kontynuowanie, koszt konserwacji kodu będzie przewyższał koszt jego ponownego napisania. Wróćmy teraz do przykładu modelowania ruchu ulicznego, w którym użyłem modelu statycznego do analizy relacji odkrytych w fazie planowania systemu przesyłania komunikatów. Nie projektowałem modelu statycznego, by później próbować na jego podstawie budować model dynamiczny (z uwzględnieniem ograniczeń narzuconych przez przygotowany wcześniej model statyczny).

Uważne projektując i skupiając się na tym, co chcesz osiągnąć (nie na tym, jak to zrobić), możesz wyeliminować ze swojego programu znaczną część metod zwracających i ustawiających.

Sam wizualizuj swoje dane

Prawdopodobnie najbardziej zdumiewającym posunięciem w procesie projektowania systemu modelującego ruch uliczny było zdefiniowanie metody `drawYourself(…)` w ramach klasy `Road`. Umieściłem kod interfejsu użytkownika w logice biznesowej aplikacji! Warto się jednak zastanowić, jakie będą następstwa ewentualnej zmiany wymagań odnośnie do interfejsu użytkownika. Przykładowo, mogę zechcieć reprezentować odcinek drogi w formie podwójnej linii, której oba kierunki jazdy będą oznaczone różnymi kolorami. Mogę też zdecydować o rysowaniu na tych liniach kropek reprezentujących

samochody. Skoro obiekty klasy `Road` same odpowiadają za swoją reprezentację graficzną, wszystkie te zmiany powinny dotyczyć wyłącznie definicji tej klasy. Co więcej, różne typy obiektów klasy `Road` (np. parkingi) mogą realizować zadanie wizualizacji w różny sposób. Wadą takiego rozwiązania jest oczywiście zamieszanie w samej klasie `Road`, ale cały kod związany z obsługą interfejsu użytkownika można skupić w jednej klasie wewnętrznej, aby zapewnić przejrzystość systemu.

Warto też pamiętać, że żaden kod interfejsu użytkownika tak naprawdę nie został umieszczony w logice biznesowej systemu. Do opracowania warstwy interfejsu użytkownika użyłem podsystemu AWT lub Swing — oba stanowią dodatkowe warstwy abstrakcyjne. Właściwy kod interfejsu znajduje się więc w implementacji tych podsystemów. Na tym właśnie polega zaleta warstw abstrakcyjnych — umożliwia izolowanie „logiki biznesowej" od odpowiednich mechanizmów podsystemu. Mogę łatwo przenieść swój system do innego środowiska graficznego bez konieczności zmiany kodu, zatem jedynym problemem jest skomplikowany kod. Z tym utrudnieniem mogę sobie poradzić, umieszczając wszystkie operacje związane z wizualizacją obiektów w wewnętrznej klasie (lub stosując wzorzec fasady, który omówię za chwilę).

Warto pamiętać, że tylko najprostsze klasy mogą obsługiwać swoją wizualizację za pomocą pojedynczej metody `drawYourself()`. W większości przypadków programista będzie potrzebował większej kontroli. Zdarza się, że te same obiekty muszą prezentować swoją zawartość na wiele sposobów (język HTML, kontrolka `JLabel` podsystemu Swing itp.); równie często będziesz musiał wizualizować tylko kilka spośród wielu atrybutów obiektu.

Co więcej, izolacja implementacji obiektu od reszty programu wcale nie wymaga fizycznego rysowania reprezentowanych przez ten obiekt danych na ekranie. Tak naprawdę potrzebujesz jedynie jakiegoś uniwersalnego (przynajmniej na poziomie programu) mechanizmu graficznego reprezentowania danych. Obiekt może np. przekazywać wizualizację swoich danych w formie dokumentu XML do podsystemu wyświetlania. Klasa pomocnicza użyta wraz z wyrażeniami `java.text.NumberFormat` może przekształcać tę reprezentację zgodnie z określonymi ustawieniami regionalnymi. Wspominana klasa `Money` może zwracać w formacie Unicode odpowiednią wizualizację łańcucha (obiektu klasy `String`), która będzie łączyła kwotę pieniężną z symbolem waluty. Można nawet zwracać obraz w formacie *.gif* lub kontrolkę `JLabel`.

Zmierzam do tego, że jeśli reprezentacje tych atrybutów są obsługiwane w sposób właściwy, możliwe jest modyfikowanie wewnętrznych mechanizmów klasy bez konieczności dostosowywania kodu używającego tych reprezentacji. (Reprezentację jakiegoś obiektu bądź atrybutu, która jest prezentowana w sposób umożliwiający wyświetlanie, ale nie modyfikowanie, można traktować jak pewną odmianę wzorca memento — patrz dalsza część tego rozdziału. Istnieje też możliwość wykorzystywania innych wzorców projektowych, w szczególności wzorca budowniczego, do wymuszania na obiektach samodzielnego prezentowania swoich danych i jednocześnie izolowania kodu tworzącego interfejs użytkownika od właściwych obiektów. Więcej informacji na temat tego wzorca znajdziesz w rozdziale 4.).

JavaBeans i Struts

„Ale" — możesz protestować — „co z JavaBeans, Struts i innymi bibliotekami, które wykorzystują przecież zarówno akcesory, jak i mutatory?" No właśnie, co z nimi? Istnieje doskonały sposób budowy komponentów JavaBean bez metod zwracających i ustawiających — właśnie w tym celu stworzono klasy `BeanCustomer`, `BeanInfo` i `BeanDescriptor`. Projektanci specyfikacji komponentów JavaBean uwzględnili metody `get` i `set` tylko dlatego, że sądzili, iż ułatwi to przygotowywanie komponentów niedoświadczonym programistom (mieli nadzieję, że tacy programiści będą mogli tworzyć komponenty i jednocześnie uczyć się, jak należy to robić „prawidłowo"). Przewidywania twórców tej technologii niestety się nie sprawdziły.

Ludzie mają skłonność do nadmiernego przywiązywania się do biblioteki JavaBeans (i wszelkich innych bibliotek wykorzystujących jakieś elementy proceduralne). Programiści zdają się zapominać, że biblioteki te początkowo opracowywano z myślą o rozwiązywaniu konkretnych problemów napotykanych przez programistów. Niektórzy programiści są szczególnie związani z technikami programowania proceduralnego, inni nie mogą zapomnieć o technikach obiektowych. Część projektantów celowo „ogłupia" swoje interfejsy, ponieważ wie, że w przeciwnym razie wielu ludzi po prostu by tych interfejsów nie zrozumiało.

Przykładem takiego właśnie podejścia są metody `get` i `set` zastosowane w technologii JavaBeans. Metody te wprowadzono wyłącznie w celu zapewnienie dostępu do pewnych właściwości między innymi z poziomu narzędzi budowania interfejsów użytkownika. Projektanci tej biblioteki nie przypuszczali, że programiści będą te metody wywoływali samodzielnie. Jedynym celem ich zdefiniowania było umożliwienie interfejsom API introspekcji klasy `Class`, sprawdzanie istnienia poszczególnych „właściwości" (w ramach zautomatyzowanych narzędzi, w tym środowisk implementowania interfejsów użytkownika) na podstawie analizy samych nazw metod. Takie podejście nie sprawdziło się w praktyce. Wraz z metodami zwracającymi i ustawiającymi wprowadzono do definicji klas mnóstwo niepotrzebnego kodu, który uczynił całą technologię zdecydowanie zbyt skomplikowaną i zbyt proceduralną. Programiści, którzy nie rozumieli idei abstrakcji danych, wywoływali te metody, powodując, że kod tworzony na podstawie biblioteki JavaBeans stał się niezwykle trudny w konserwacji. Między innymi z tego względu w wersji 1.5 Javy wprowadzono mechanizm „metadanych", dzięki któremu można wyeliminować z kodu następujące wyrażenia:

```
private int  property;
public  int  getProperty (         ){ return property;  }
public  void setProperty (int value){ property = value; }
```

i zastąpić je jedną deklaracją w postaci:

```
private @property int property;
```

Narzędzia wspomagające budowę interfejsów użytkownika mogą teraz używać interfejsów API introspekcji do odnajdywania właściwości zamiast analizować nazwy metod i na ich podstawie wnioskować o istnieniu lub braku właściwości o określonej nazwie. Co więcej, nowy mechanizm eliminuje konieczność zaśmiecania kodu metodami akcesorów.

Wracając do biblioteki Struts, z pewnością nie jest to przykład architektury obiektowej i jego projektanci nigdy nie próbowali tego celu osiągnąć. Zastosowana w bibliotece Struts architektura MVC po prostu zmusza programistów do stosowania metod `get` i `set`. Możesz oczywiście argumentować, że to uniwersalny charakter biblioteki Struts *wyklucza* jego obiektowość; okazuje się jednak, że inne architektury interfejsu użytkownika są znacznie lepiej odizolowane od logiki biznesowej niż w przypadku MVC. (Być może najlepszym rozwiązaniem jest po prostu unikanie technologii interfejsu użytkownika zbudowanych na podstawie koncepcji MVC. Architektura MVC powstała blisko 30 lat temu i od tamtego czasu nasza wiedza na temat systemów informatycznych znacznie się poszerzyła.) Istnieje jeden istotny argument przemawiający za stosowaniem technologii Struts — biblioteka ta zawiera mnóstwo kodu, którego nie musisz pisać, i który jest „wystarczająco dobry" dla bardzo wielu zastosowań. Jeśli „wystarczająco dobry" oznacza, że jakość tego kodu rzeczywiście wystarczy, po co szukać lepszych rozwiązań?

Podsumowując, wielu programistów mówiło mi, że podstawowe koncepcje decydujące o obiektowości systemów (np. ukrywanie szczegółów implementacyjnych) są „niepotrzebne" tylko dlatego, że nie zostały urzeczywistnione w bibliotekach używanych na co dzień przez tych programistów (JavaBeans, Struts, .NET itp.). Moim zdaniem niepotrzebne są tego typu dyskusje.

Dostrajanie

Słyszałem jeszcze jeden argument przemawiający za stosowaniem metod akcesorów i mutatorów, zgodnie z którym takie środowiska programowania jak Eclipse i pokrewne do tego stopnia ułatwiają dostrajanie definicji metod do zwracania określonych typów, że w praktyce nie ma się czym przejmować. Ja jednak nadal się przejmuję.

Po pierwsze, mechanizm dostrajania kodu w środowisku Eclipse obejmuje tylko istniejący projekt. Jeśli więc Twoja klasa jest wykorzystywana w wielu projektach, będziesz musiał dostroić każdy z nich osobno. Firmy, które prawidłowo korzystają z techniki wielokrotnego używania gotowych klas, zatrudniają zwykle wiele grup programistów pracujących równolegle nad różnymi projektami — programiści z poszczególnych zespołów nie będą zadowoleni, kiedy powiesz im o daleko idących zmianach w kodzie współużytkowanej klasy, wynikających z jakiejś, skądinąd słusznej, obserwacji.

Po drugie, automatyczne dostrajanie sprawdza się tylko w przypadku stosunkowo niewielkich i prostych zmian, nie jest więc właściwym rozwiązaniem w przypadku poważnych modyfikacji. Konsekwencje takich zmian są zwykle zbyt rozległe, aby mogło sobie z nimi poradzić zautomatyzowane narzędzie. Możesz np. zmienić skrypty języka SQL i pośrednio wpłynąć na metody wywoływane w otoczeniu zmodyfikowanych skryptów.

I wreszcie po trzecie, warto jeszcze raz odnieść się do wspomnianych zmian w klasie `Money` i polu `System.out`. Prosta modyfikacja typów kilku zwracanych wartości nie wystarczy do właściwej obsługi omówionych przeze mnie zmian. Konieczna będzie przebudowa kodu otaczającego wywołania odpowiednich metod zwracających. Trudno oczywiście dowieść, że dostrajanie kodu jest czymś złym, możesz sam się przekonać, że zmian, o których mówię po prostu nie da się wprowadzić za pomocą zautomatyzowanego narzędzia.

Osoby wykorzystujące w dyskusji argument o możliwości automatycznego dostrajania kodu zwykle nie rozumieją czegoś znacznie ważniejszego: Nadużywanie akcesorów i mutatorów na poziomie kluczowej abstrakcji jest wyznacznikiem kiepsko zaprojektowanego systemu przesyłania komunikatów. Innymi słowy, zbudowany w ten sposób kod ma prawdopodobnie tak niedopracowaną strukturę, że jego konserwacja będzie się wiązała z mnóstwem niepotrzebnych utrudnień, niezależnie od potencjalnych możliwości w zakresie automatycznego dostrajania. W takim przypadku konieczne jest ponowne zaprojektowanie systemu, nie jego dostrajanie.

Wróćmy do wcześniejszego przykładu pola `System.out`, który jest dość charakterystyczny — wyobraź sobie, że zaprojektowałeś język Java od początku, i że obiekty klasy `String` są wyświetlane na konsoli w sposób następujący:

```
String s = "witaj świecie";

s.print( String.TO_CONSOLE );
```

natomiast wczytywanie łańcuchów z konsoli odbywa się tak:

```
s.load( String.FROM_CONSOLE );
```

W takim przypadku wszystkie problemy związane z reprezentacją bajtową i formatem Unicode zostałyby rozwiązane wewnątrz implementacji klasy `String`. Także wszelkie zmiany operacji wejścia-wyjścia (np. przekształcenia reprezentacji bajtowej na reprezentację znakową) przestałyby stanowić problem. Ponieważ tak naprawdę jedynym zadaniem interfejsów `Reader` i `Writer` jest wczytywanie i zapisywanie łańcuchów, przedstawione powyżej rozwiązanie umożliwiłoby całkowitą rezygnację z tych interfejsów. Operacje wejścia-wyjścia mogłyby być obsługiwane przez odpowiednie przeciążenia metod `print(…)` i `load(…)`.

Oczywiście możesz się ze mną spierać, czy wspomniane problemy powinny być rozwiązywane w ten sposób. Możesz też krytykować pomysł wprowadzenia składowej `TO_CONSOLE` do klasy `String` lub `File`. Tak czy inaczej, zaproponowany projekt całkowicie eliminuje konieczność stosowania pola `System.out` i wszystkich jego akcesorów. Można rozważać mnóstwo operacji na łańcuchach i sugerować, że żadna z nich nie powinna być częścią klasy `String`, ale zapewniam Cię, że rzeczywistym rozwiązaniem tego problemu są wzorce projektowe (wzorzec wizytatora, strategii itd.).

Życie bez metod get i set

Jak można projektować systemy obiektowe bez metod zwracających i ustawiających (odpowiednio `get` i `set`)? Jak należy projektować systemy przesyłania komunikatów, które będą minimalizowały udział metod akcesorów i mutatorów? Problem tkwi we właściwym projektowaniu, nie w kodowaniu. Nie ma prostej reguły, według której należałoby „umieścić ten kod w tym miejscu”, ponieważ problem dotyczy raczej właściwego podejścia do mechanizmów komunikowania się obiektów. Nie wystarczy po prostu wyrzucić z kodu metody `get` i `set` — należy przebudować kod od podstaw przy użyciu innej struktury.

Proces projektowania systemów obiektowych koncentruje się wokół tzw. *przypadków użycia*, czyli autonomicznych zadań realizowanych przez użytkownika końcowego i pozwalających wygenerować jakieś przydatne dane wyjściowe. Przykładowo, „logowanie do systemu" nie jest przypadkiem użycia, ponieważ nie prowadzi do wytworzenia jakichkolwiek przydatnych wyników w dziedzinie problemu. Przypadkiem użycia jest natomiast „generowanie czeku". W przykładzie poświęconym terminalom ATM skupiłem się na przypadku użycia polegającym na „wypłacaniu gotówki".

System obiektowy jest więc implementacją czynności niezbędnych do realizacji określonych „scenariuszy", które wynikają z uprzednio przeanalizowanych przypadków użycia. Obiekty czasu wykonywania aplikacji, które mają przypisane role w danym przypadku użycia wykonują swoje zadania, wymieniając się komunikatami z innymi obiektami. Warto jednak pamiętać, że nie wszystkie komunikaty są takie same. Budowa programu proceduralnego, który wykorzystuje obiekty i klasy nie daje większych korzyści.

Kiedy w roku 1989 Kent Beck i Ward Cunningham prowadzili zajęcia poświęcone projektowaniu systemów obiektowych, napotykali na poważny opór ze strony programistów, którzy nie byli skłonni rezygnować ze swoich przyzwyczajeń (w szczególności — stosowania metod `get` i `set`). Scharakteryzowali ten problem w następujący sposób:

> *Największym problemem podczas nauki programowania obiektowego jest skłonienie osoby nauczanej do odrzucenia przyzwyczajeń do globalnej wiedzy o kontroli, która jest możliwa w programach proceduralnych, i skupienia się na lokalnej wiedzy poszczególnych obiektów, która także pozwala osiągać wyznaczone cele. Niedoświadczeni projektanci wykazują skłonność do myślenia globalnego — stosują niepotrzebne zmienne globalne, wskaźniki i w sposób nieuzasadniony uzależniają funkcjonowanie swoich obiektów od implementacji innych obiektów.*

Mówiąc o „globalnej wiedzy o kontroli" Beck i Cunningham tak naprawdę opisywali wspominaną w tym rozdziale „wszechmogącą" klasę — klasę, której obiekty gromadzą informacje ze wszelkich możliwych źródeł i samodzielnie przetwarzają wszystkie zebrane w ten sposób dane (zamiast pozwolić na przetwarzanie danych obiektom, które je przechowują). Zdanie o „nieuzasadnionym uzależnianiu funkcjonowania obiektów od implementacji innych obiektów" dotyczy między innymi wywołań metod akcesorów i mutatorów.

Cunningham opracował metodykę nauczania, która w przystępny sposób demonstruje proces projektowania — tzw. kartę CRC. Idea karty CRC sprowadza się do budowy tabel 4×6 cali, które są dzielone na następujące trzy części:

Klasa	Nazwa klasy obiektów.
Odpowiedzialność	Co te obiekty mogą robić? Zakres odpowiedzialności obiektów jednej klasy powinien dotyczyć pojedynczego obszaru wiedzy specjalistycznej.
Klasy współpracujące	Pozostałe klasy obiektów, z którymi bieżąca klasa obiektów może się komunikować. Zbiór klas współpracujących powinien być możliwie niewielki.

Pierwszy cykl opracowywania kart CRC zawsze jest obarczony poważnymi błędami — pewne czynniki będą jeszcze ulegały zmianie.

W czasie swoich zajęć Beck i Cunningham prezentowali przypadek użycia i namawiali słuchaczy do odgadywania, które obiekty będą wymagane do realizacji opisanego scenariusza. Analiza rozpoczynała się zwykle od dwóch obiektów, do których z czasem dodawano kolejne w odpowiedzi na demonstrowany rozwój scenariusza. Uczestnicy ćwiczeń byli wyznaczani do roli poszczególnych obiektów i otrzymywali kopie przypisanych im kart CRC. Jeśli niezbędne było użycie wielu obiektów pojedynczej klasy, wyznaczano wiele osób (po jednej dla każdego z tych obiektów). Studenci dosłownie wystawiali sztukę opartą na przypadku użycia. Oto kilka reguł, których staram się przestrzegać podczas „gry" w przypadku użycia z kartami CRC:

- Wykonuję czynności odpowiadające roli danego obiektu w przypadku użycia, rozmawiając z innymi „obiektami".
- Mogę rozmawiać wyłącznie z obiektami klas współpracujących. Jeśli muszę się skontaktować z obiektem spoza tej grupy, rozmawiam z obiektem klasy współpracującej, który może się komunikować z tamtym obiektem. Jeśli taka pośrednia komunikacja nie jest możliwa, dodaję do swojej karty CRC kolejną klasę współpracującą.
- Nie mogę pytać o informacje, które są mi potrzebne do realizacji jakiegoś zadania. Zamiast tego powinienem prosić te obiekty klas współpracujących, które dysponują odpowiednimi danymi o wykonanie tego zadania dla mnie. W takim przypadku można oczywiście przekazywać tym obiektom informacje niezbędne do realizacji zleconego zadania, jednak zakres przekazywanych danych powinien być ograniczony do minimum.
- Jeśli istnieje zadanie, którego nikt nie jest w stanie zrealizować, należy utworzyć nową klasę (wraz z odpowiednią kartą CRC) lub rozszerzyć zakres odpowiedzialności jednej z istniejących klas (zmodyfikować jej kartę CRC).
- Jeśli karta CRC jest pełna, należy stworzyć jeszcze jedną klasę (z nową kartą CRC), która przejmie część obowiązków dotychczasowej, przeciążonej klasy. Złożoność poszczególnych klas jest ograniczona rozmiarem karty CRC.
- Staram się utożsamiać z dziedziną problemu (obsługą konta bankowego, zakupami internetowymi itp.) zarówno na poziomie słownictwa, jak i procesów. Oznacza to, że próbuję modelować zdarzenia dokładnie tak, jak prawdziwi eksperci w danej dziedzinie rozwiązują analizowany problem. Udaję, że nie istnieje coś takiego jak komputer. Trzeba przyznać, że dosyć rzadko spotyka się na takich kursach osoby, które zwracają się do siebie, stosując polecenia „getX"; w praktyce metody `get` i `set` są więc niemal zupełnie zbędne.

Kiedy już zakończysz konwersację rozwiązującą dany problem, włącz dyktafon i spróbuj ją powtórzyć lub odtwórz ją z pamięci na kartce papieru. Otrzymany tekst lub nagranie będzie swoistym „modelem dynamicznym" programu. Ostateczny zbiór kart CRC jest „modelem statycznym" programu. Po wykonaniu wielu prób i naniesieniu odpowiednich poprawek można w ten sposób rozwiązywać niemal wszystkie problemy.

Opisany przeze mnie proces tak naprawdę *jest* procesem projektowania obiektowego, tyle że uproszczonym na potrzeby środowiska zajęć dydaktycznych. Niektórzy projektują nawet rzeczywiste programy z wykorzystaniem kart CRC, ale technika ta nie sprawdza się w przypadku większych, niebanalnych rozwiązań. Znacznie częściej projektanci budują modele dynamiczne i statyczne w języku UML, stosując któryś z formalnych procesów (RUP, Crystal, a nawet pewne odmiany Extreme Programming). Kluczowe znaczenie ma fakt, iż system obiektowy jest w istocie konwersacją pomiędzy obiektami. Jeśli przez chwilę się nad tym zastanowisz, dojdziesz do przekonania, że metody `get` i `set` najzwyczajniej w świecie nie są potrzebne w takiej konwersacji. Z tego samego względu metody `get` i `set` nie powinny występować w Twoim kodzie, jeśli przed przystąpieniem do jego budowy przeprowadziłeś proces projektowania z należytą starannością.

Tak długo, jak długo to tylko możliwe modelowanie nie powinno wykraczać poza „dziedzinę problemu" (zgodnie z tym, o czym wspomniałem w ostatniej regule). Tym, co wpędza większość ludzi w kłopoty jest przekonanie o modelowaniu dziedziny w czasie, gdy tak naprawdę proces modelowania ma miejsce na poziomie implementacji. Jeśli Twój system przesyłania komunikatów nie korzysta ze słownictwa typowego dla danej dziedziny problemu — jeśli Twój język nie jest zrozumiały dla przeciętnego użytkownika tego programu — to tak naprawdę modelujesz ten system na poziomie implementacji. W takich środowiskach jak komputery (lub, co gorsza, bazy danych lub narzędzia wspomagające budowę interfejsów użytkownika) nie ma miejsca na ten poziom modelowania.

Przykładowo, w modelowaniu CRC wymagane jest utrzymywanie konwersacji w obszarze dziedziny problemu właśnie przez stosowanie takiego słownictwa i procesów, którymi posługiwaliby się prawdziwi użytkownicy końcowi. W ten sposób można zapewnić wierne odwzorowanie modelowanej dziedziny przez system przesyłania komunikatów. Baza danych jest tylko elementem wewnętrznym, który jest wykorzystywany przez niektóre klasy w roli mechanizmu utrwalania danych, i który w ogóle nie powinien występować w początkowym modelu.

Jeśli utrzymasz strukturę komunikatów w obszarze dziedziny problemu, najprawdopodobniej uda Ci się wyeliminować zdecydowaną większość metod `get` i `set`, ponieważ tego typu polecenia po prostu nie występują w komunikacji pomiędzy ekspertami w danej dziedzinie, którzy rozwiązują rzeczywiste problemy.

Kiedy stosowanie akcesorów i mutatorów jest uzasadnione?

Jeśli musisz przekazywać informacje pomiędzy obiektami, powinieneś hermetycznie zamykać te informacje w innych obiektach. Funkcja `get`, która zwraca obiekt klasy `Money`, jest kompletnie nieprzydatna z punktu widzenia obiektu oczekującego wartości `double`.

Najlepszym rozwiązaniem byłoby oczywiście stosowanie metod zwracających obiekty, które implementują określone interfejsy, ponieważ takie interfejsy izolowałyby obiekty wywołujące od ewentualnych zmian implementacji klasy. Tego typu metod (zwracających referencje do interfejsów) nie należy traktować jak typowych metod zwracających, ponieważ nie pełnią one funkcji mechanizmów zapewniających dostęp do określonych

pól. Jeśli zmieni się wewnętrzna implementacja klasy udostępniającej dane, konieczna będzie taka modyfikacja definicji zwracanego obiektu, która uwzględni wprowadzone zmiany. Możliwe jest nawet zwracanie obiektów innych klas, pod warunkiem, że nowe obiekty implementują ten sam (oczekiwany po stronie wywołującego) interfejs. Zewnętrzny kod, który wykorzystuje taki obiekt za pośrednictwem jego interfejsu jest więc skutecznie chroniony.

Ogólnie, staram się jednak unikać nawet tych stosunkowo niegroźnych form akcesorów do zwracania tylko egzemplarzy klas stanowiących *kluczowe abstrakcje* systemu. (Jeśli nazwa klasy lub interfejsu odpowiada opisowi problemu na poziomie dziedzinowym, to traktuję tę klasę lub interfejs jak kluczową abstrakcję systemu).

Komunikaty powinny zawierać (w postaci swoich argumentów) możliwie niewielkie ilości danych, lepszym rozwiązaniem jest jednak „upychanie" danych w obiektach niż ich „wyciąganie" z obiektów. Innymi słowy, lepiej gdy delegujemy zadanie do innego obiektu (przekazując mu kilka niezbędnych informacji), niż wywołujemy jedną z jego metod do uzyskania potrzebnych danych (które należy jeszcze przetworzyć). Nie twierdzę, że zwracanie wartości jest czymś złym, ale dopóki jest to możliwe, powinieneś zwracać albo obiekty z ściśle ukrywające swoje implementacje, albo wartości logiczne, które nie mówią absolutnie niczego na temat implementacji. W przypadku terminala ATM lepszym rozwiązaniem będzie skierowanie do bazy danych pytania: „Czy mogę wypłacić Billowi 20 dolarów?" (dla którego centrala może wygenerować odpowiedź logiczną), niż wysłanie żądania: „Podaj mi stan rachunku Billa" i podjęcie decyzji w warunkach lokalnych.

Istnieje jeden poważny wyjątek od reguły unikania metod zwracających i ustawiających. Wszystkie systemy obiektowe zawierają coś, co nazywam „graniczną warstwą proceduralną". Przykładowo, zdecydowana większość programów obiektowych działa w proceduralnych systemach operacyjnych lub komunikuje się z proceduralnymi bazami danych. Interfejsy łączące te zewnętrzne podsystemy proceduralne z natury rzeczy są uniwersalne. Projektant biblioteki JDBC nie ma przecież pojęcia o tym, co będziesz robił ze swoją bazą danych, zatem zaprojektowane przez niego klasy muszą być wysoce elastyczne. Innym przykładem biblioteki „warstwy granicznej" są klasy obsługujące interfejsy użytkownika (np. należące do biblioteki Swing języka Java). Projektanci biblioteki Swing nie wiedzą, jak ich klasy będą używane, zatem stworzone przez nich rozwiązania są bardzo uniwersalne. W normalnych warunkach zapewnianie elastyczności (a więc tworzenie niepotrzebnych funkcji) jest o tyle niewłaściwe, że wymaga znacznych nakładów czasowych. Tak czy inaczej, elastyczność jest nieodłączną cechą interfejsów granicznych, zatem klasy tej warstwy pełne są metod akcesorów i mutatorów. Projektanci tego typu rozwiązań rzeczywiście nie mają wyboru.

W praktyce problem nieznajomości przyszłych zastosowań kodu dotyczy wszystkich pakietów Javy. Wyeliminowanie wszystkich akcesorów i mutatorów jest niezwykle trudne, jeśli nie są znane przyszłe zastosowania obiektów danej klasy. Gdy weźmie się pod uwagę tę istotną przeszkodę, trzeba przyznać, że projektanci Javy wykonali kawał dobrej roboty, ukrywając tyle szczegółów implementacyjnych, ile było możliwe. Nie twierdzę, że wszystkie decyzje podjęte podczas projektowania biblioteki JDBC mają zastosowanie w Twoim kodzie. Tak nie jest. Warto jednak pamiętać, że Twoja sytuacja jest dużo lepsza — *wiesz*, jak poszczególne klasy będą wykorzystywane, zatem nie musisz tracić czasu na zapewnianie niepotrzebnej elastyczności.

Muszę też wspomnieć o wartościach stałych, które często są udostępniane bezpośrednio jako publiczne składowe obiektów. Oto moje rady w tej kwestii:

- Nie rób tego, jeśli nie musisz. Na przykład lepszym rozwiązaniem jest skalowanie listy w sposób dostosowujący jej rozmiar do zawartości niż udostępnianie stałej `MAX_SIZE`.
- Używaj nowego (wprowadzonego w JDK 1.5) mechanizmu `enum` wszędzie tam, gdzie jest to możliwe. Takie rozwiązanie jest dużo lepsze od kategorycznego deklarowania i inicjalizowania wartości typu `static final int`. Alternatywnym rozwiązaniem jest stosowanie wzorca typów wyliczeniowych zapewniającego bezpieczeństwo typów, który opisano w książce *Efektywne programowanie w języku Java* autorstwa Joshuy Blocha (Helion, 2002).

  Zasadniczym problemem jest zdefiniowanie typu wyliczeniowego w następujący sposób:

```
private static class Format{ private Format(); }
public static final Format SINGLE_LINE  = null;
public static final Format POPUP_DIALOG = new Format();
public static final Format PANEL        = new Format();

public displayYourselfAs( Format how )
{   // Wyświetla bieżącą wartość kalendarza we
    // wskazanym formacie.
}
```

  Ponieważ argumentem metody `displayYourselfAs(…)` jest obiekt klasy `Format`, i ponieważ mogą istnieć tylko dwa egzemplarze tej klasy (i trzy referencje do tych egzemplarzy), nie możesz przekazać do tej metody błędnej wartości. Gdybyś jednak użył następującego, zdecydowanie bardziej popularnego rozwiązania opartego na wyliczeniu typu `int`:

```
public static final int SINGLE_LINE  = 0;
public static final int POPUP_DIALOG = 1;
public static final int PANEL        = 2;

public displayYourselfAs( int how )
{   //…
}
```

  mógłbyś przekazywać w formie argumentu metody `displayYourselfAs(…)` wartości zupełnie nonsensowne (np. `-1`). Bloch poświęcił temu zagadnieniu dziesięć stron, więc nie pozostaje mi nic innego jak odesłać Cię do jego książki.

- Jeśli musisz udostępnić stałą, upewnij się, że rzeczywiście jest to stała. Dostępne w Javie słowo kluczowe `final` gwarantuje, że dana referencja nie będzie modyfikowana, zatem nigdy nie wskaże na inną strukturę, co jednak wcale nie oznacza, że równie dobrze chroniony jest obiekt wskazywany przez tę referencję. Jeśli dany obiekt jest wykorzystywany w roli stałej, musisz napisać odpowiednią klasę w taki sposób, aby modyfikowanie jej obiektów po prostu nie było możliwe. (W Javie o takich klasach mówi się, że są *niezmienne*, jednak poza deklarowaniem wszystkich pól takiej klasy ze słowem kluczowym `final` język ten nie oferuje żadnych dodatkowych mechanizmów gwarantowania niezmienności. Jedynym sposobem jest programowanie klas w taki właśnie sposób).

Przeanalizujmy teraz dostępną w Javie klasę `Color`. Kiedy już utworzysz obiekt tej klasy, nie możesz zmienić reprezentowanego przez niego koloru, ponieważ klasa `Color` nie udostępnia żadnych metod zmieniających kolor. Przyjrzyj się następującemu fragmentowi:

```
public static final Color background = Color.RED;
//...
c.darken();
```

Wywołanie metody `darken()` nie modyfikuje obiektu wskazywanego przez stałą `background`; zamiast tego metoda zwraca obiekt klasy `Color`, którego odcień jest ciemniejszy od koloru oryginalnego. Powyższy kod tak naprawdę niczego nie robi, ponieważ zwrócony obiekt klasy `Color` nie jest nigdzie umieszczany, zatem nie możesz teraz wykonać operacji:

```
background = c.darken();
```

ponieważ pole `background` zostało zadeklarowane ze słowem kluczowym `final`.

I wreszcie zdarza się, że jeden obiekt pełni funkcję „opiekuna" wielu innych obiektów. Przykładowo, dostępna w Javie klasa `Collection` zawiera cały zbiór obiektów, które zostały do niej przekazane z zewnątrz. Ponieważ słowa `get` i `set` są często wykorzystywane w nazwach metod odpowiedzialnych wyłącznie za umieszczanie obiektów w obiekcie opiekuna oraz wyciąganie tych obiektów z tego zbiorczego obiektu, metody te nie odsłaniają informacji o sposobie działania obiektu. Rozwiązanie to jest więc zupełnie poprawne. Ogólnie, jeśli przekazujesz jakieś dane do zbiorczego obiektu, oczekiwanie odnośnie do możliwości ponownego pobrania zapisanych danych z tego obiektu jest czymś normalnym.

Skrajnymi przykładami „opiekunów" informacji są bazy danych, choć twórcy ich interfejsów poszli jeszcze dalej w kierunku metod `get` i `set`, ponieważ baza danych w zasadzie stała się rozwiązaniem proceduralnym (wielkim zbiorem informacji) i istotnym elementem wspominanej już „warstwy granicznej". Efektem takiego podejścia jest brak możliwości uzyskiwania dostępu do proceduralnych baz danych przy użyciu technik typowych dla rozwiązań obiektowych. Stosowanie metod `get` i `set` jest nieuniknione. Możesz jednak (i powinieneś) zamykać wywołania proceduralne kierowane do warstwy bazy danych w funkcjonujących na wyższym poziomie obiektach dziedzinowych, po czym pisać swój kod na bazie interfejsów tych obiektów. W kodzie obiektów pośredniczących możesz umieścić dowolną potrzebną liczbę wywołań `get` i `set` kierowanych do bazy danych. Większa część Twojego programu nie będzie jednak miała dostępu do mechanizmu obsługującego bazę danych, ponieważ w komunikacji z tym proceduralnym podsystemem będą pośredniczyły specjalne obiekty wyższego poziomu.

Podsumowanie problematyki metod zwracających i ustawiających

Spróbujmy teraz podsumować nasze wnioski i obserwacje: *Nie* twierdzę, że zwracanie wartości jest złe, że informacje nie mogą być przekazywane pomiędzy poszczególnymi składnikami systemu, lub że można wyeliminować z programu wszystkie akcesory i mu-

tatory. Informacje muszą przepływać w systemie, ponieważ brak takiego przepływu oznaczałby całkowity paraliż programu. Wszelkie przekazywane informacje powinny jednak być hermetycznie zamykane w obiektach, które ukrywają swoje implementacje przed pozostałymi elementami programu.

Oto najważniejsze wnioski:

- Możliwości konserwacji programu są odwrotnie proporcjonalne do ilości danych przekazywanych pomiędzy obiektami.
- Ujawnianie szczegółów implementacyjnych negatywnie wpływa na możliwość konserwacji kodu. Zanim dodasz do definicji klasy metodę akcesora lub mutatora, upewnij się, że jest ona rzeczywiście niezbędna.
- Klasy, które bezpośrednio modelują system na poziomie dziedzinowym są niekiedy nazywane *obiektami biznesowymi* i prawie nigdy nie potrzebują akcesorów i mutatorów. Program można traktować jak szeroki zbiór uniwersalnych bibliotek, które należy połączyć zgodnie z zaleceniem o braku metod zwracających i ustawiających, oraz klas dziedzinowych, które powinny zawierać i izolować całą implementację. Obecność metod `get` i `set` na tym poziomie pokazuje, że proces projektowania nie został przeprowadzony prawidłowo (w szczególności, zabrakło należytej staranności w fazie modelowania dynamicznego).
- Utrzymywanie procesu projektowania w zakresie dziedziny problemu (w ramach określonego środowiska „biznesowego") tak długo, jak długo to tylko możliwe, pozwala na zaprojektowanie systemu przesyłania komunikatów pozbawionego metod zwracających i ustawiających, ponieważ w większości przypadków polecenia `get` i `set` po prostu nie występują w dziedzinie problemu.
- Im bardziej zbliżysz się do proceduralnej granicy systemu obiektowego (interfejsu bazy danych, klas obsługujących interfejs użytkownika itp.), tym trudniej Ci będzie ukryć szczegóły implementacyjne swojego kodu. Rozważnym posunięciem jest w takim przypadku użycie akcesorów i mutatorów w specjalnie utworzonej warstwie granicznej.
- Także uniwersalne biblioteki i klasy nie mogą w pełni ukrywać swoich implementacji, zatem zawsze oferują akcesory i mutatory.
- Czasem nie warto tracić czasu na pełne (hermetyczne) izolowanie implementacji. Wystarczy wspomnieć tak banalne klasy jak `Point` czy `Dimension`. Podobnie, model izolowania prywatnych klas, które są tworzone w ramach implementacji innych klas (np. klasa `Node` zdefiniowana jako prywatna klasa wewnętrzna klasy `Tree`), często może być znacznie mniej restrykcyjny. Z drugiej strony, warto pamiętać o problemach powodowanych przez pola `System.in`, `System.out` i `System.err`, kiedy wprowadzono klasy `Reader` i `Writer`. Czy potrafiłbyś np. dodać jednostki (centymetry, metry, itp.) do klasy `Dimension`?

Na konferencji JavaOne (myślę, że w roku 1991) poproszono Jamesa Goslinga o podzielenie się z słuchaczami możliwie krótką radą na temat programowania. Gosling odpowiedział (nie jest to cytat), że możliwość konserwacji kodu jest odwrotnie proporcjonalna

do ilości danych przemieszczających się pomiędzy obiektami. Oznacza to, że choć nie możesz całkowicie wyeliminować przesyłania danych (szczególnie w środowiskach z wieloma obiektami, które stale ze sobą współpracują celem wykonania pewnych zadań), powinieneś przynajmniej próbować minimalizować przepływ informacji.

Kiedy już muszę przekazać jakąś informację z jednego obiektu do drugiego, staram się to robić z uwzględnieniem następującej pary reguł:

- Przesyłaj obiekty (najlepiej opierając się na implementowanych przez nie interfejsach), nie surowe dane.
- Stosuj model „upychania" danych, zamiast modelu „wyciągania" danych. Przykładowo, obiekt może delegować zadanie do jednego ze swoich obiektów współpracujących (przekazując mu niewielką ilość informacji niezbędnych do realizacji zleconego zadania). Rozwiązanie alternatywne, w którym obiekt współpracujący „wyciąga" informacje od obiektu delegującego za pośrednictwem metod zwracających, jest z wielu względów gorsze. Wzorzec wagi piórkowej bazuje właśnie na modelu „upychania".

Przekształcenie systemu komunikacji z modelu „upychania" w model „wyciągania" często jest tylko kwestią odpowiedniego przekierowania komunikatów.

Ocena możliwości konserwowania kodu nie jest binarna — osobiście przywiązuję do tego problemu bardzo dużą wagę, ponieważ konserwacja programu tak naprawdę rozpoczyna się na kilka sekund po jego napisaniu. Kod zbudowany z uwzględnieniem możliwości jego konserwowania zwykle jest nie tylko bardziej zrozumiały, ale też bardziej niezawodny.

Tak czy inaczej, musisz zdecydować, w którym miejscu rozległej skali możliwości konserwacji chcesz usytuować swój program. Biblioteki Javy (a przynajmniej ich zdecydowana większość) są dobrym przykładem kompromisu pomiędzy takimi możliwościami a stosunkowo uniwersalną funkcjonalnością. Autorzy pakietów języka programowania Java ukryli tyle szczegółów implementacyjnych, ile tylko zdołali (warto pamiętać, że ich biblioteki musiały zapewniać nie tylko wyjątkową elastyczność, ale także odpowiednie graniczne interfejsy proceduralne). Ceną, jaką musieli zapłacić, są utrudnienia we wprowadzaniu strukturalnych zmian do takich bibliotek jak Swing, ponieważ zbyt wiele istniejących programów już teraz jest uzależnionych od szczegółów implementacyjnych tych bibliotek.

Nie wszystkie biblioteki Javy odkrywają swoje implementacje. Wystarczy wspomnieć o interfejsach API biblioteki Crypto (w pakiecie *javax.crypto*) oraz klasach `URL` i `URLConnection`, które nie odsłaniają niemal żadnych informacji i jednocześnie są wyjątkowo elastyczne. Także klasy serwletów są dobrym przykładem hermetycznego zamknięcia implementacji, co i tak nie wyklucza przekazywania informacji (choć twórcy tych klas mogli pójść jeszcze krok dalej i opracować dodatkową warstwę abstrakcji umożliwiającą budowę kodu HTML).

Kiedy więc widzisz metody, których nazwy rozpoczynają się od słów `get` lub `set`, powinieneś się zastanowić, co to właściwie oznacza. Zadaj sobie pytanie, czy taki prze-

pływ danych jest rzeczywiście konieczny. Czy możesz tak zmodyfikować system przesyłania komunikatów, aby informacje były przekazywane w formie mniej szczegółowych struktur, co mogłoby wyeliminować ten tryb przenoszenia danych? Czy możesz przekazywać te same informacje w postaci argumentów komunikatów, zamiast w postaci osobnych komunikatów? Czy alternatywna architektura rzeczywiście umożliwi lepsze ukrywanie implementacji? Jeśli jednak nie ma alternatywy dla istniejącego rozwiązania, musisz się z tym pogodzić.