Projektowanie zorientowane obiektowo. Wzorce projektowe. Wydanie II

Autorzy: Alan Shalloway, James R. Trott
Tłumaczenie: Piotr Rajca
ISBN: 83-7361-782-5
Tytuł oryginału: Design Patterns Explained A New Perspective on Object-Oriented Design, 2nd Edition
Format: B5, stron: 368

Zmień podejście do programowania – zastosuj wzorce projektowe

- Skorzystaj z metod modelowania obiektowego w języku UML
- Poznaj różne typy wzorców projektowych
- Wykorzystaj wzorce projektowe w swoich programach

Wzorce projektowe to modele rozwiązań wielu zagadnień programistycznych, oparte na zasadach programowania obiektowego. Zastosowanie ich w projektach informatycznych zapewnia szybszą i bardziej efektywną pracę zarówno podczas projektowania i tworzenia oprogramowania, jak i na etapie jego wdrożenia. Sprawne korzystanie z wzorców projektowych wiąże się jednak z koniecznością poznania metod modelowania obiektowego, zrozumienia zasad obiektowości i umiejętności podzielenia projektowanego systemu na komponenty.

Książka „Programowanie zorientowane obiektowo. Wzorce projektowe. Wydanie drugie" to przewodnik po wzorcach projektowych, przedstawiający je od strony najbardziej istotnej dla programisty – od strony praktycznej. Przykłady w języku Java, diagramy UML i wyczerpujące komentarze – wszystko to sprawia, że po przeczytaniu tej książki staniesz się ekspertem w dziedzinie wzorców projektowych i będziesz wykorzystywać je we wszystkich swoich projektach.

- Zasady obiektowości
- Modelowanie obiektowe w języku UML
- Standardowe rozwiązania obiektowe
- Wprowadzenie do wzorców projektowych
- Zasady stosowania wzorców projektowych
- Katalog wzorców projektowych
- Projektowanie i programowanie z zastosowaniem wzorców projektowych

Korzystając z wzorców projektowych, zwiększysz szybkość i efektywność swojej pracy nad aplikacjami.

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

Spis treści

Rozdział 8.

Poszerzamy horyzonty

Przegląd

W rozdziale

W poprzednich rozdziałach omówiłem trzy podstawowe koncepcje, na których opiera się projektowanie obiektowe: obiekty, hermetyzację oraz klasy abstrakcyjne. Właściwe zrozumienie tych pojęć przez projektanta jest niezwykle istotne. Tradycyjne sposoby ich rozumienia mają wiele ograniczeń, dlatego też w niniejszym rozdziale powrócę raz jeszcze do omawianej wcześniej problematyki. Moją intencją będzie przedstawienie nowych sposobów rozumienia projektowania obiektowego, które wynikają z perspektywy wzorców projektowych. Niestety, tradycyjne sposoby mają bardzo duże ograniczenia.

W rozdziale ponownie zastanowię się nad zagadnieniami przedstawionymi we wcześniejszej części książki, jak również zaprezentuję kilka nowych tematów. Chciałbym przedstawić czytelnikowi nowy sposób spojrzenia na projektowanie obiektowe, perspektywę, która wyłania się dzięki zrozumieniu wzorców projektowych. Następnie opiszę kluczowe cechy kodu o wysokiej jakości. Znaczenie tych cech podkreślają propagatorzy i zwolennicy programowania inteligentnego (ang. *agile coding*), czyli tworzenia kodu zgodnie z zasadami programowania ekstremalnego (ang. *extreme programming*, programowania bazującego na testowaniu). Co ciekawe, te same cechy występują także we wzorcach projektowych, a jeśli będziemy postępować zgodnie z zasadami i metodologią wzorców projektowych, to pojawią się one w sposób naturalny. Mam nadzieję, że prezentując te cechy zarówno pod kątem programowania inteligentnego, jak i wzorców projektowych, wypełnię lukę występującą pomiędzy tymi dwoma podejściami do projektowania.

Niniejszy rozdział:

- przedstawia i porównuje tradycyjny sposób rozumienia obiektów (jako zestawu danych i metod) z nowym sposobem (jako bytów o określonej odpowiedzialności),
- przedstawia i porównuje tradycyjny sposób rozumienia hermetyzacji (jako ukrywania danych) z nowym sposobem (jako możliwości ukrycia w ogóle); szczególnie istotne będzie tu zrozumienie tego, że hermetyzacja służyć może także jako sposób ukrycia różnic w zachowaniu obiektów,

- przedstawia i porównuje różne sposoby obsługi różnic w zachowaniu,
- przedstawia i porównuje tradycyjny sposób wykorzystania dziedziczenia (służący specjalizacji oraz ponownemu wykorzystaniu istniejącego kodu) z nowym sposobem (polegającym na wykorzystaniu dziedziczenia w celu klasyfikacji obiektów); pokazuje również, że sposoby te umożliwiają zawarcie zmienności w zachowaniu obiektów,
- opisuje analizę wspólności i zmienności,
- przedstawia to, jak perspektywy koncepcji, specyfikacji oraz implementacji mają się do klas abstrakcyjnych i ich klas pochodnych,
- porównuje wzorce projektowe oraz programowanie inteligentne; choć początkowo może się wydawać, iż oba te podejścia nie są ze sobą zgodne, to okazuje się jednak, że zwracają one uwagę na podobne jakości programowania — nadmiarowość, czytelność oraz łatwość testowania.

Podziękowanie

Przedstawiona przeze mnie nowa perspektywa obiektowości nie jest zupełnie oryginalna. Stosowali ją z pewnością projektanci poszukujący *wzorców projektowych*. Jest także zgodna z wynikami prac Christophera Alexandra, Jima Copliena (do jego pracy będę się odwoływać w dalszej części rozdziału) oraz Bandy Czworga[1].

Mimo to perspektywa obiektowości nie doczekała się dotąd takiego przedstawienia, jakie zamieszczam w niniejszym rozdziale książki. Powstało ono na podstawie analizy wzorców projektowych i sposobu ich opisu przez innych autorów.

Pisząc tutaj o „nowej" perspektywie obiektowości mam na myśli to, że przedstawiony dalej sposób rozumienia obiektowości będzie prawdopodobnie nowością dla wielu projektantów. Podobnie jak był dla mnie, kiedy po raz pierwszy zapoznawałem się z tematyką wzorców projektowych.

Obiekty — w rozumieniu tradycyjnym i nowym

Rozumienie tradycyjne: dane oraz metody

Tradycyjnie przez obiekty rozumiemy dane oraz operujące na nich metody. Jeden z moich wykładowców nazwał je też kiedyś „inteligentnymi danymi", gdyż chciał odróżnić je od bazy danych. Obiekty są zatem postrzegane jako inteligentny sposób obsługi danych: „Zacznijmy od danych opisujących stan dziedziny problemu, dodajmy do nich metody operujące na tych danych (czyli niezbędne działanie) i *voilà* — mamy gotowe obiekty!". Jednak jest to zbyt uproszczony sposób patrzenia na obiekty, można by rzec — sposób jednowymiarowy. Taki sposób widzenia obiektów mieści się jednak w perspektywie implementacji.

[1] Gdyż pisząc niniejszą książkę, udało mi się poznać kilka osób zajmujących się tworzeniem programów w języku Smalltalk. Niemal wszystkie one miały takie samo podejście do projektowania obiektowego jak to, które prezentuję w niniejszej książce.

Nowe rozumienie: byty posiadające odpowiedzialność

Bardziej przydatna okazuje się tu definicja obiektu powstająca w perspektywie koncepcji — jako *bytu o określonej odpowiedzialności*. Odpowiedzialność ta określa z kolei sposób zachowania obiektu. Dlatego też czasami możemy w skrócie powiedzieć, że obiekt reprezentuje byt o określonym zachowaniu.

Zaletą nowej definicji jest to, że pomaga ona skoncentrować się na *zadaniach* obiektu, a nie na sposobie ich implementacji. Dzięki temu w procesie tworzenia oprogramowania możemy wyróżnić dwa etapy:

1. wstępnego projektu — na etapie tym możemy uniknąć zajmowania się szczegółami implementacji.
2. implementacji projektu.

Skoncentrowanie uwagi na tym, co obiekt ma robić, pozwala także nie przejmować się zbyt wcześnie szczegółami jego implementacji. Pozwala na ukrycie szczegółów tej implementacji. To z kolei pomaga w pisaniu oprogramowania, które w przyszłości będzie można łatwiej modyfikować… oczywiście jeśli zajdzie taka konieczność.

Jest to możliwe dzięki temu, iż zwracając uwagę na działanie obiektu, koncentrujemy się jedynie na jego interfejsie publicznym, czyli na „oknie komunikacyjnym", za pomocą którego można poprosić obiekt o wykonanie pewnej czynności. Dysponując dobrym interfejsem, można „poprosić" obiekt o wykonanie dowolnej czynności mieszczącej się w granicach jego odpowiedzialności i jednocześnie mieć pewność, że obiekt ją wykonana. Nie trzeba przy tym dysponować żadnymi informacjami odnośnie zdarzeń zachodzących *wewnątrz* obiektu. Nie trzeba wiedzieć, w jaki sposób obiekt wykorzysta przekazane do niego informacje ani jak zdobędzie inne dane, które są mu potrzebne. Przekazujemy odpowiedzialność obiektowi i więcej nic nas nie interesuje.

Zastanówmy się na przykład nad obiektem klasy `Figura`, którego odpowiedzialność będzie stanowić:

- przechowanie informacji o jego położeniu na ekranie,
- narysowanie własnej reprezentacji na ekranie,
- usunięcie reprezentacji z ekranu.

Istnienie tych obowiązków określa wprost zestaw potrzebnych metod:

- `pobierzPolozenie(...)`
- `rysujFigure(...)`
- `usunFigure(...)`

Nie określam przy tym żadnych szczegółów wewnętrznej implementacji obiektu, a jedynie wymieniam jego obowiązki. Obiekt może przechowywać odpowiednie atrybuty lub posiadać dodatkowe metody, które wyznaczą odpowiednie wartości (na przykład na podstawie informacji zawartych w innych obiektach). Obiekt klasy `Figura` może więc zawierać atrybuty określające jego położenie lub pobierać te informacje na przykład z obiektu reprezentującego bazę danych. W ten sposób uzyskujemy wysoką elastyczność ułatwiającą osiągnięcie zadań projektowania (bądź zmianę kodu, jeśli cele ulegną zmianie).

Czytelnik z pewnością zauważy też, że koncentracja na motywacji (a nie na implementacji) jest koncepcją powtarzającą się we wzorcach projektowych. Wynika to z faktu, iż użycie interfejsu do ukrycia implementacji w zasadniczy sposób oddziela ją od obiektów, które z niej korzystają.

Proponuję, by czytelnik przyjął zaprezentowany tu sposób widzenia obiektów. Rezultatem takiej decyzji będzie lepsza jakość tworzonych rozwiązań.

Hermetyzacja — w rozumieniu tradycyjnym i nowym

Mój obiektowy parasol

Podczas wykładów poświęconych wzorcom projektowym często zadaję moim studentom pytanie: „Kto z Państwa spotkał się z definicją hermetyzacji mówiącą o ukrywaniu danych?". Prawie wszyscy podnoszą w odpowiedzi rękę.

Następnie opowiadam im historię mojego parasola. Proszę pamiętać, że mieszkam w Seattle, które posiada — nieco przesadzoną — opinię wyjątkowo deszczowej okolicy. Prawdą jest jednak, że od jesieni do wiosny jest tutaj dość mokro i wtedy parasole i kurtki z kapturem należą do artykułów pierwszej potrzeby.

Opowiem teraz o moim wielkim parasolu. Jest tak duży, że oprócz mnie mogą się pod nim zmieścić jeszcze trzy, a nawet cztery osoby! Kiedy już jesteśmy w jego wnętrzu, czyli poza zasięgiem deszczu, możemy się za jego pomocą przemieszczać. Dodatkowo zabawia nas w tym czasie jego system stereofoniczny, a klimatyzacja zapewnia odpowiednią temperaturę. Prawda, że to niezwykły parasol?

Jest przy tym bardzo wygodny w użyciu. Nie muszę go ze sobą nosić, bo zawsze czeka na mnie na zewnątrz. Wyposażony jest ponadto w koła, żeby łatwiej można się było przemieszczać. Ale nie muszę go pchać ani ciągnąć, ponieważ posiada własny napęd. Korzystam z niego nawet wtedy, gdy nie pada. Jeśli świeci słońce i chcę nacieszyć się jego promieniami, otwieram górną część parasola (powód, dla którego używam parasola nawet wtedy, gdy nie pada, nie jest dla mnie jasny).

Mieszkańcy Seattle używają setki tysięcy podobnych parasoli w przeróżnych kolorach.

Większość ludzi nazywa je jednak *samochodami*.

Sam jednak częściej myślę o moim samochodzie jak o parasolu, ponieważ zwykle chroni mnie przed deszczem. Czekając na kogoś na dworze często siadam pod moim „parasolem", aby nie zmoknąć.

Definicje mogą narzucać ograniczenia

Jednak samochód nie jest parasolem. Możemy go wykorzystywać jako schronienie przed deszczem, ale jest to dość ograniczony sposób wykorzystania możliwości, jakie daje samochód. Podobnie jest z hermetyzacją — nie służy ona jedynie do ukrywania danych. Taki sposób myślenia o hermetyzacji ogranicza moje możliwości jako projektanta.

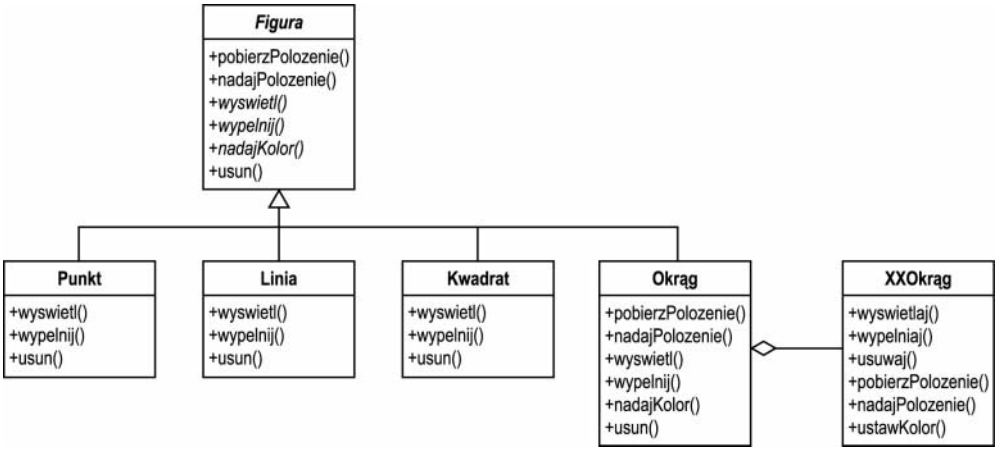
W jaki sposób myśleć o hermetyzacji

O hermetyzacji powinno myśleć się jak o ukrywaniu w ogóle. Innymi słowy — hermetyzacja *może* służyć do ukrycia danych. Ale może także ukrywać:

- sposób implementacji,
- klasy pochodne,
- szczegóły projektowe,
- reguły tworzenia obiektów.

We wcześniejszych rozważaniach dotyczących ukrywania implementacji w zasadzie „hermetyzowałem" ją. Aby posunąć się jeszcze dalej, przeanalizujmy diagram przedstawiony na rysunku 8.1, który został po raz pierwszy zamieszczony w rozdziale 7., zatytułowanym „Wzorzec adaptera". Klasy `Punkt`, `Linia`, `Kwadrat` oraz `Okrag` dziedziczą po klasie `Figura`. Dodatkowo klasa `Okrag` „opakowuje" lub zawiera klasę `XXOkrag`.

Rysunek 8.1 przedstawia kilka rodzajów hermetyzacji.



Rysunek 8.1. *Dostosowanie klasy XXOkrag za pomocą klasy Okrag*

Kilka poziomów hermetyzacji

Diagram ten przedstawia wiele sposobów zastosowania hermetyzacji:

- **Hermetyzację danych** — dane wewnątrz obiektów klas `Punkt`, `Linia` oraz `Kwadrat` są ukryte przed obiektami innych klas.
- **Hermetyzację metod** — na przykład metoda `pobierzPolozenie` w klasie `Okrag`.
- **Hermetyzację innych obiektów** — jedynie obiekt klasy `Okrag` posiada dostęp do zawartego w nim obiektu klasy `XXOkrag`.
- **Hermetyzację typów** — użytkownicy klasy `Figura` nie wiedzą o istnieniu klas `Punkt`, `Linia`, `Kwadrat`.

Hermetyzację typów uzyskuje się zatem w przypadku, gdy istnieje klasa abstrakcyjna mająca kilka klas pochodnych (lub interfejs wraz z jego implementacjami) wykorzystywanych w oparciu o zasady polimorfizmu. Użytkownik korzystający z tej klasy

abstrakcyjnej nie zna typu klasy pochodnej obiektu, którym się w danej chwili posługuje. To właśnie ten rodzaj hermetyzacji ma zazwyczaj na myśli Banda Czworga.

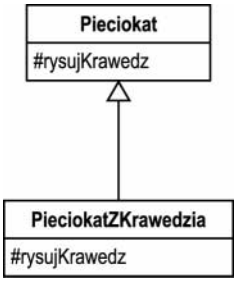
Zalety tej nowej definicji

Rozumienie hermetyzacji w szerszy sposób przyczynia się do uzyskania lepszej struktury programu. Hermetyzacja ułatwia określenie interfejsów, na których opiera się projekt. Ukrywając za pomocą klasy `Figura` istnienie klas reprezentujących poszczególne rodzaje figur, można później dodawać ich kolejne rodzaje bez obawy o to, że będzie to wymagać zmian w programie użytkownika. Podobnie — ukrywając istnienie obiektu klasy `XXOkrag` wewnątrz klasy `Okrag`, można później zmienić w dowolny sposób implementację rysowania okręgu.

Dziedziczenie jako pojęcie a dziedziczenie jako sposób wielokrotnego zastosowania

W początkowym okresie (tuż po zaprezentowaniu paradygmatu obiektowego) uważano, że jedną z jego najważniejszych zalet jest możliwość ponownego wykorzystania istniejącego kodu poprzez tworzenie klas pochodnych za pomocą dziedziczenia z istniejących klas bazowych. W ten sposób powstał termin *specjalizacja*, który służy do określenia procesu tworzenia klas pochodnych (dlatego też klasy pochodne nazywa się czasem *klasami wyspecjalizowanymi*, a klasy bazowe — *klasami ogólnymi*).

Nie zamierzam tutaj podważać słuszności takiego twierdzenia. Proponuję jednak wykorzystanie dziedziczenia w sposób, który uważam za bardziej doskonały. Załóżmy, na przykład, że chciałbym posługiwać się pięciokątem. Definiuję zatem klasę `Pieciokat`, która będzie zawierać stan nowej figury oraz metody pozwalające na jej wyświetlenie, usunięcie itd. Nieco później okazuje się, że potrzebny mi jest pięciokąt ze specjalnymi krawędziami. Mogę zatem użyć klasy `Pieciokat` i na jej podstawie stworzyć bardziej wyspecjalizowaną klasę pochodną dysponująca niezbędnym algorytmem wyświetlania krawędzi (rysunek 8.2).



Rysunek 8.2. *Klasa PieciokatZKrawedzia dziedziczy po klasie Pieciokat*

Był to przykład zastosowania dziedziczenia w celu specjalizacji. Wykorzystałem klasę `Pieciokat`, aby stworzyć nową klasę — `PieciokatZKrawedzia`. Rozwiązanie to spisuje się dobrze, choć przysparza trzech problemów opisanych w tabeli 8.1.

Innym sposobem zastosowania dziedziczenia jest klasyfikacja klas pod kątem identycznego zachowania. Zagadnienie to rozwinę w dalszej części rozdziału.

Tabela 8.1. *Problemy, jakich przysparza zastosowanie dziedziczenia w celu specjalizacji.*

Problem	Opis
Może przyczyniać się do występowania niskiego stopnia spójności.	Zastanówmy się, co by się stało, gdyby istniało wiele różnych typów krawędzi? Okazuje się, że w takim przypadku klasa `Pieciokat` (oraz jej klasy pochodne) nie opisuje już wyłącznie samej figury, lecz także jej krawędzie, a to sprawia, iż klasa ta musi zajmować się dodatkowymi problemami. Co więcej, w klasie mogą się także pojawić inne zmienne aspekty (na przykład rodzaj wypełnienia pięciokąta).
Ogranicza możliwości wielokrotnego stosowania kodu.	Jeśli stworzę w klasie `Pieciokat` (i jej klasach pochodnych) kod obsługujący różne rodzaje krawędzi, to w jaki sposób będę mógł z niego skorzystać w innych klasach? Zadanie to byłoby bardzo trudne, gdyż za każdym razem zmienia się kontekst, a co więcej, gdyż kod obsługujący znajduje się w klasie `Pieciokat` i raczej nie będzie dostępny poza nią.
Utrudnia obsługę zmian.	Metoda specjalizacji w celu wielokrotnego zastosowania doskonale nadaje się do przedstawiania w klasie, gdyż można ją zademonstrować i przejść do dalszych zagadnień, zanim ktokolwiek zdąży zapytać, co się stanie, gdy pojawi się możliwość modyfikacji jakiegoś innego czynnika. Na przykład co zrobić, jeśli pojawią się dwa różne rodzaje cieniowania? Aby je obsłużyć, trzeba by stworzyć nowe, bardziej wyspecjalizowane wersje klasy `Pieciokat` (co oznaczałoby częściowe powielenie kodu).

Określ zmienność i hermetyzuj ją

Wzorce projektowe wykorzystujące dziedziczenie w celu sklasyfikowania odmiennego zachowania

Autorzy książki *Design Patterns: Elements of Reusable Object-Oriented Software* sugerują, co następuje:

> *Spróbujmy określić, co jest zmienną w naszym projekcie. Takie podejście stanowi przeciwieństwo koncentrowania się na przyczynach zmian w projekcie. Zamiast zastanawiać się, co może spowodować wprowadzenie zmian do projektu, skoncentrujmy się na tym, co możemy zmienić bez konieczności modyfikacji projektu. Skoncentrujmy się zatem na hermetyzacji tego, co ulega zmianie, czyli sposobie stosowanym przez wiele wzorców projektowych*[2].

Osobiście preferuję nieco inne ujęcie tej samej kwestii: *Znajdź, co się zmienia i hermetyzuj to*.

Takie stwierdzenie, może wydać się czytelnikowi mało zrozumiałe, jeśli nadal myśleć będzie o hermetyzacji jak o ukrywaniu danych. Stanie się dużo bardziej czytelne, jeśli czytelnik pomyśli o hermetyzacji jako o ukrywaniu klas pochodnych za pomocą klasy abstrakcyjnej lub interfejsu — czyli o „hermetyzacji typu”[3]. Udostępnienie referencji

[2] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Boston: Addison-Wesley, 1995, s. 29.

[3] Ogólnie rzecz biorąc, to właśnie o tym rodzaju hermetyzacji myśli Banda Czworga, używając terminu „hermetyzacja”.

do takiej abstrakcyjnej klasy lub interfejsu (agregacja) ukrywa klasy pochodne reprezentujące różnice w sposobie działania. Innymi słowy, pewna klasa posiada referencję do klasy abstrakcyjnej lub do interfejsu posiadających więcej niż jedną klasę pochodną. Jednak te klasy pochodne są ukryte (hermetyzowane) przez klasę, która ich używa.

Wiele wzorców projektowych stosuje hermetyzację w celu utworzenia warstw pomiędzy obiektami, co umożliwia wprowadzanie zmian po jednej ze stron warstwy bez wpływu na obiekty znajdujące się po przeciwnej stronie warstwy. Jest to możliwe dzięki wprowadzeniu przez wzorzec niskiego stopnia powiązania pomiędzy obiektami po obu stronach warstwy.

Zawieranie zmienności danych a zmienności zachowania

Sposób ten stanowi podstawę działania wzorca mostu, który przedstawię w rozdziale 10. zatytułowanym „Wzorzec mostu". Wcześniej chciałbym jednak omówić pewien błąd, który często popełniają projektanci.

Przypuśćmy, że pracuję nad projektem, który tworzy modele przeznaczone do opisu różnych cech zwierząt. Wymagania będą w tym przypadku określone następująco:

- zwierzęta mogą posiadać różną liczbę nóg,
- obiekty reprezentujące zwierzęta muszą umożliwić przechowanie tej informacji i jej uzyskanie,
- różne zwierzęta mogą poruszać się w różny sposób,
- obiekty reprezentujące zwierzęta muszą umożliwić uzyskanie informacji o tym, ile czasu zajmie im pokonanie określonego dystansu na danym terenie.

Typowym sposobem, w jaki programista poradzi sobie z problemem różnej ilości nóg, będzie utworzenie zmiennej składowej wewnątrz obiektu, która przechowywać będzie odpowiednią wartość, a także metod umożliwiających nadanie wartości zmiennej i pobranie jej wartości. Jednak — aby uporać się z problemem zmienności w zachowaniu obiektów — potrzebne będzie inne rozwiązanie.

Przypuśćmy, że określone są dwa sposoby poruszania się: chodzenie i latanie. Dla każdego z nich potrzebny będzie osobny fragment kodu, gdyż sama zmienna niczego tutaj nie rozwiąże (choć można jej użyć w celu określenia, jaki sposób poruszania się jest dostępny). W tym wypadku programista wybierze raczej jedno z dwu rozwiązań:

- utworzenie zmiennej składowej, która przechowywać będzie informację o sposobie poruszania się zwierzęcia,
- utworzenie osobnej klasy pochodnej klasy bazowej `Zwierze` dla reprezentacji zwierząt, które chodzą, i osobnej klasy dla tych, które latają.

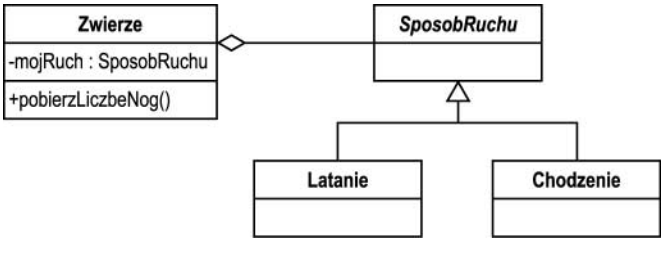
Okazuje się jednak, że w sytuacjach, gdy problem staje się złożony, to oba te rozwiązania zawodzą. Doskonale spełniają swe zadania, gdy istnieje tylko jeden zmienny czynnik (sposób poruszania się), jednak co się stanie, gdy liczba tych czynników wzrośnie? Na przykład co w sytuacji, jeśli pojawią się orły (latające drapieżniki), lwy (drapieżniki poruszające się na lądzie), wróble (ptaki roślinożerne) oraz krowy (zwierzęta roślinożerne poruszające się po lądzie)? Wykorzystanie instrukcji wyboru do określania typu zwierzęcia spowodowałoby skojarzenie sposobów poruszania się oraz odżywiania — czyli

czynników, które nie wydają się być ze sobą połączone. Z kolei wykorzystanie dziedziczenia do obsługi każdej z sytuacji wyjątkowych prowadzi do ogromnego wzrostu ilości klas. Poza tym co się stanie, jeśli zwierzęta raz przejawiają jeden sposób zachowania, a w innych przypadkach zachowują się inaczej (na przykład większość ptaków potrafi zarówno latać, jak i poruszać się po lądzie)?

Istnieje jeszcze inny problem. Tworzenie klas obsługujących coraz to więcej czynników zmiennych (na przykład wykorzystując w tym celu instrukcje wyboru) może doprowadzić do zmniejszenia spójności kodu. Oznacza to, że im więcej przypadków szczególnych obsługuje klasa, tym trudniej jest zrozumieć jej kod.

Obsługa zmienności działania poprzez zastosowanie obiektów

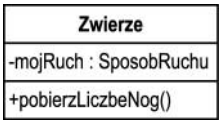
Innym rozwiązaniem może okazać się umieszczenie wewnątrz obiektu klasy `Zwierze` obiektu określającego sposób poruszania się, co ilustruje diagram pokazany na rysunku 8.3.



Rysunek 8.3. *Obiekt klasy Zwierze zawiera obiekt klasy SposobRuchu*

To nie jest żadna przesada

Rozwiązanie to może na pierwszy rzut oka wyglądać nadmiarowo. Jednak w praktyce oznacza ono jedynie tyle, że obiekt klasy `Zwierze` zawiera odpowiedni obiekt określający sposób jego poruszania się. Jest więc analogiczne do rozwiązania, w którym zmienną wykorzystujemy do przechowania informacji o liczbie nóg zwierzęcia (z tą różnicą, że w tym przypadku zmienna składowa reprezentuje różnicę w zachowaniu, a nie w liczbie). Może jedynie wydawać się, że oba te rozwiązania się różnią, choćby na podstawie różnic w diagramach przedstawionych na rysunkach 8.3 i 8.4.



Rysunek 8.4. *Obiekt zawierający atrybuty*

Porównanie obu rozwiązań

Wielu projektantów uważa, że pomiędzy zawieraniem przez obiekt innego obiektu, a zawieraniem przez obiekt atrybutów istnieje różnica. Jednak mimo że atrybuty są zmiennymi typów prostych (na przykład `double`, `integer`) i nie przypominają obiektów, to są nimi z punktu widzenia projektowania obiektowego. Pamiętajmy, że w programowaniu obiektowym *wszystko* stanowi obiekt (nawet podstawowe typy danych, których zachowanie określa arytmetyka). Specyficzna składnia posługiwania się tymi obiektami (na przykład `x+y` odpowiadająca `x.add(y)`) ukrywa jedynie fakt, iż są to obiekty o określonym zachowaniu.

W ten sposób rozwiązanie zastosowane w przypadku zmienności atrybutów i rozwiązanie w przypadku zmienności zachowania okazują się do siebie podobne. Najłatwiej będzie to pokazać na przykładzie. Załóżmy, że opracować muszę system obsługi punktu sprzedaży. Kluczowy element tego systemu stanowić będzie faktura. Na fakturze tej znajdzie się całkowita wartość zakupu. Początkowo dla jej reprezentacji mógłbym użyć typu prostego `double`. Jeśli jednak system będzie musiał wystawiać faktury w różnych walutach, to szybko pojawi się problem odpowiedniej konwersji. Dlatego też zdecyduję się raczej utworzyć klasę `Pieniadz`, która przechowywać będzie informacje o kwocie i jej walucie. Tak więc suma na fakturze będzie teraz manifestacją obiektu klasy `Pieniadz`.

Choć może wydawać się na początku, że jedynym zadaniem obiektu klasy `Pieniadz` jest przechowanie odpowiedniej informacji, to jednak szybko okaże się, że zgodnie z zasadą odpowiedzialności obiekty tej klasy muszą posiadać także metody służące konwersji pomiędzy różnymi walutami. Jak się okazuje, zadanie konwersji nie sprowadza się tylko do przechowania w obiekcie kolejnej informacji (o aktualnym przeliczniku walut).

Komplikację wprowadzić może na przykład konieczność dokonywania konwersji pomiędzy walutami na podstawie ich kursów pochodzących z przeszłości. W takim przypadku atrybut można by zastąpić klasą `Waluta`. Dodawanie zachowań do klasy `Pieniadz` lub `Waluta` dodaje je także do klasy `Rachunek`, która zależy od umieszczonych w niej obiektów `Pieniadz` (a zatem także i obiektów `Waluta`). Niemniej jednak takie rozwiązanie ani nie powoduje zwiększenia stopnia złożoności klasy `Rachunek`, ani nie wymaga wprowadzania w niej jakichkolwiek zmian.

Strategię polegającą na uzyskiwaniu określonego zachowania obiektu w zależności od rodzaju zawieranego obiektu zademonstruję omawiając kilka następnych wzorców projektowych.

Analiza wspólności i zmienności a klasy abstrakcyjne

Analiza wspólności i zmienności

Książka Copliena omawiająca problem analizy wspólności i zmienności, pokazuje, jak odnajdywać w dziedzinie problemu czynniki zmienne oraz elementy wspólne: „Określ, *gdzie* („analiza wspólności") oraz *jak* („analiza zmienności") elementy się od siebie różnią".

Analiza wspólności

Coplien stwierdza, iż: „Analiza wspólności polega na poszukiwaniu wspólnych elementów, które pozwalają zrozumieć, na czym polega podobieństwo członków tej samej rodziny"[4]. Pod pojęciem „członków rodziny" Coplien rozumie elementy, które są ze sobą powiązane ze względu na sytuację, w jakiej się pojawiają, lub funkcje, jakie wykonują. Proces odnajdywania cech wspólnych definiuje rodzinę, do której należą elementy (a zatem, także, jakie są różnice pomiędzy nimi). Na przykład, gdyby ktoś pokazał nam flamaster do pisania na tablicy, pióro oraz ołówek, to moglibyśmy stwierdzić, iż ich wspólną cechę jest

[4] Coplien J., *Multi-Paradigm Design for C++*, Boston: Addison-Wesley, 1998, str. 63.

przeznaczenie — wszystko są to przedmioty służące do pisania. Proces, jaki wykonaliśmy, aby określić wszystkie te przedmioty w identyczny sposób, nazywamy analizą wspólności. Dysponując cechami wspólnymi (przedmioty do pisania), łatwiej można określić, czym poszczególne przedmioty różnią się od siebie (na czym się pisze, kształt przedmiotu i tak dalej).

Analiza zmienności

Analiza zmienności ma na celu określenie, czym poszczególni członkowie rodziny różnią się od siebie. Te odmienności mają sens wyłącznie w odniesieniu do elementów, dla których określono cechy wspólne:

> *Analiza wspólności poszukuje struktury, która jest niezmienna, natomiast analiza zmienności poszukuje struktury, która może się zmieniać. Analiza zmienności ma sens wyłącznie w kontekście zdefiniowanym przez odpowiednią analizę wspólności… W odniesieniu do architektury analiza wspólności zapewnia jej długowieczność, natomiast analiza zmienności — przydatność*[5].

Innymi słowy, jeśli czynnikiem zmiennym są konkretne klasy należące do dziedziny problemu, to czynniki wspólne definiują te pojęcia dziedziny, które łączą te klasy ze sobą. Pojęcia wspólne będą reprezentowane przez klasy abstrakcyjne. Różnice wskazane przez analizę zmienności będą implementowane przez konkretne klasy (to znaczy przez klasy pochodne klasy abstrakcyjnej).

Nowy paradygmat znajdowania obiektów

Często niedoświadczeni projektanci programów obiektowych są instruowani, aby analizować dziedzinę problemu oraz „odnajdywać istniejące rzeczowniki i tworzyć klasy, które będą je reprezentować, a następnie odnajdywać czasowniki (czyli akcje) i implementować je poprzez dodawanie metod do wcześniej utworzonych obiektów". Taki proces, polegający na skoncentrowaniu uwagi na rzeczownikach i czasownikach, zazwyczaj prowadzi do powstawania większych hierarchii klas, niż można by sobie tego życzyć. Sugeruję, by podstawowym narzędziem podczas tworzenia obiektów była analiza wspólności i zmienności, gdyż metoda ta jest lepsza od wyróżniania rzeczowników i czasowników (jest ona częściowo zgodna z metodą postulowaną przez Copliena).

Projektowanie obiektowe obejmuje wszystkie trzy perspektywy

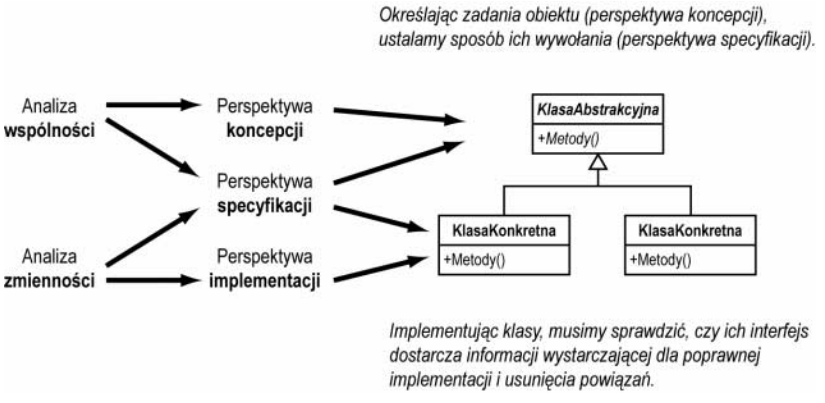
Rysunek 8.5 obrazuje związki zachodzące pomiędzy:

- analizą wspólności i zmienności,
- perspektywami koncepcji, specyfikacji oraz implementacji,
- klasą abstrakcyjną, jej interfejsem i klasami pochodnymi.

Teraz specyfikacja pozwala na lepsze zrozumienie klas abstrakcyjnych

Jak pokazano na rysunku 8.5, analiza wspólności związana jest z warstwą koncepcyjną dziedziny zastosowań, a analiza zmienności odnosi się do warstwy implementacji (czyli specyficznych przypadków problemu).

[5] Ibidem, strony 60 i 64.



Rysunek 8.5. *Związki pomiędzy analizą wspólności i zmienności, perspektywami i klasą abstrakcyjną*

Warstwa specyfikacji znajduje się pośrodku. Zarówno analiza wspólności, jak i zmienności jest z nią związana. Warstwa specyfikacji określa sposób komunikacji z obiektami, które są koncepcyjnie podobne. Natomiast poszczególne obiekty reprezentują zmienność problemu. W warstwie implementacji specyfikacja przyjmuje postać klasy abstrakcyjnej bądź interfejsu.

W nowej perspektywie projektowania obiektowego możemy wyróżnić związki przedstawione w tabeli 8.2.

Tabela 8.2. *Zalety zastosowania klas abstrakcyjnych do specjalizacji*

Związek	Omówienie
Klasa abstrakcyjna a główne pojęcie łączące klasy	Klasa abstrakcyjna stanowi kluczowe pojęcie łączące klasy pochodne i definiuje część wspólną problemu.
Część wspólna a określenie używanych klas abstrakcyjnych	Część wspólna problemu definiuje klasę abstrakcyjną.
Część zmienna a klasy pochodne	Zmienność, którą możemy zidentyfikować *wewnątrz* części wspólnej, określa klasy pochodne klasy abstrakcyjnej.
Specyfikacja a interfejs klasy abstrakcyjnej	Interfejs klasy abstrakcyjnej — a tym samym jej klas pochodnych — określony jest w warstwie specyfikacji.

Proces projektowania klas upraszcza się w ten sposób do procedury złożonej z dwu etapów przedstawionych w tabeli 8.3.

Tabela 8.3. *Dwuetapowa procedura projektowania*

Definicja	Pytanie
Klasa abstrakcyjna (część wspólna)	Jak powinien wyglądać *interfejs*, by mógł umożliwiać realizację wszystkich *odpowiedzialności* tej klasy?
Klasy pochodne	W jaki sposób powinna zostać zaimplementowana część zmienna problemu w ramach danej specyfikacji?

Związek pomiędzy perspektywą specyfikacji i perspektywą koncepcji jest więc następujący: *specyfikacja określa interfejs potrzebny do obsługi wszystkich przypadków danego problemu (czyli część wspólną określoną przez perspektywę koncepcji)*.

Związek pomiędzy perspektywą specyfikacji i perspektywą implementacji możemy natomiast określić: *biorąc pod uwagę określoną specyfikację i ustalając, w jaki sposób należy zaimplementować poszczególne przypadki (czyli część zmienną)*.

Cechy programowania inteligentnego

Projektowanie metodą „od góry do dołu" a projektowanie „w trakcie pracy"

Podejście do projektowania wykorzystujące wzorce projektowe często określa się jako „projektowanie od góry do dołu". Zaleca ono rozpoczynanie projektowania od najbardziej ogólnych pojęć i sukcesywne uwzględnianie coraz większej ilości szczegółów.

Istnieje także podejście alternatywne, postulowane przez zasady programowania ekstremalnego, które wydaje się stać w całkowitej sprzeczności z metodą przedstawioną powyżej. Programowanie ekstremalne koncentruje się na realizacji niewielkich etapów oraz weryfikację ich poprawności. Całościowy obraz rozwiązania wyłania się na podstawie tych etapów.

Osobiście uważam, że zasady programowania ekstremalnego oraz metody projektowania z wykorzystaniem wzorców projektowych nie są względem siebie sprzeczne, lecz raczej się uzupełniają. Obu tych metod można użyć w celu osiągnięcia tego samego celu — utworzenia efektywnego, solidnego i elastycznego kodu. Ale jak to jest możliwe? Sądzę, iż wynika to z faktu, że zasady, na których bazują obie te metody, są pokrewne.

Wnioski ze stosowania programowania inteligentnego

Ponieważ stosunkowo wcześnie zacząłem stosować praktyki programowania inteligentnego, dlatego też musiałem rozstrzygnąć pewien problem:

- z powodzeniem stosowałem projektowanie metodą „od góry do dołu",
- stosowanie zasad programowania inteligentnego pozwoliło mi na ograniczenie projektowania wcześniejszą metodą (a czasami nawet na całkowite jej uniknięcie),
- uzyskiwane rezultaty były jeszcze lepsze.

Mój dylemat polegał na tym, iż byłem świadom, że wzorce projektowe przyczyniły się do mych sukcesów i nie chciałem rezygnować z ich stosowania. Jednak metody programowania inteligentnego, którymi pragnąłem się posługiwać, nie zalecały takiego postępowania. Pomimo to czułem, że obie metody projektowania muszą mieć jakieś cechy wspólne — programowanie inteligentne wymaga kodu zapewniającego dużą łatwość modyfikacji, a wzorce projektowe — elastycznego kodu. Być może różnica polegała raczej na samej stosowanej metodzie niż na efektach, jakie pozwalała uzyskać.

Ostatecznie udało mi się rozwiązać mój problem, gdy zauważyłem, że obie metody wymuszają tworzenie kodu o tych samych cechach, a różnią się jedynie sposobami postępowania. Różne cechy kodu są w rzeczywistości ściśle ze sobą powiązane. Na przykład, jeśli metoda jest hermetyzowana, to w efekcie jest także odseparowana od pozostałych fragmentów programu. Praktyki zalecane przez programowanie inteligentne koncentrowały się na innych cechach niż te, o których wspominałem wcześniej. Jednak cechy te były ściśle powiązane z cechami kodu, który tworzyłem, posługując się wcześniejszymi metodami projektowania. Tymi dodatkowymi cechami są: (1) brak powtarzalności kodu, (2) czytelność, (3) łatwość testowania (przy czym podana kolejność cech nie jest odzwierciedleniem ich ważności).

Brak powtarzalności kodu

Niezwykle ważną strategią tworzenia kodu, którą należy stosować jest implementowanie konkretnej reguły tylko w jednym miejscu. Od bardzo dawna mantrą programistów obiektowych było stwierdzenie: „Jedna reguła, jedno miejsce". Reprezentuje ono najlepsze praktyki projektowe. Całkiem niedawno Kent Beck nazwał ten sposób projektowania „regułą jedynego wystąpienia"[6].

Zdefiniował ją jako element narzucanych ograniczeń:

1. System (rozumiany jako połączenie kodu i testów) musi przekazywać wszystko, co chcemy przekazać.
2. System nie może zawierać powtarzającego się kodu (oba te punkty tworzą regułę jedynego wystąpienia).

Innymi słowy, jeśli istnieje jakaś reguła określająca sposób wykonywania pewnej operacji, to należy ją zaimplementować tylko jeden raz. Zazwyczaj wymaga to stworzenia kilku niewielkich metod. Dodatkowy koszt takiego postępowania jest przeważnie minimalny, jednak pozwala uniknąć powtarzania kodu i bardzo często chroni przed przyszłymi problemami. Powielanie jest niekorzystne nie tylko ze względu na większą ilość kodu, który należy wpisać, lecz także dlatego, iż jeśli w przyszłości trzeba będzie coś zmienić, to można zapomnieć wprowadzić modyfikacje we wszystkich niezbędnych miejscach.

Nie jestem bynajmniej purystą, niemniej jednak uważam, że jeśli jest jakaś zasada, której zawsze należy przestrzegać, to jest to właśnie ta zasada. Istnieje bardzo silny związek pomiędzy powielaniem kodu oraz powiązaniami. Jeśli w programie występuje powtarzający się kod, to istnieje bardzo duże prawdopodobieństwo, że w razie konieczności zmiany fragmentu tego programu trzeba będzie wprowadzić zmiany także w jego innych miejscach. Wynika to z faktu, że powtarzające się fragmenty kodu programu są ze sobą powiązane.

Co ciekawe, w celu wyeliminowania powtarzania się kodu wystarczy postępować zgodnie z praktykami projektowania na podstawie interfejsu, a następnie wydzielić fragmenty zmienne i zapewnić wysoki stopień spójności. Jest to możliwe dzięki temu, iż kod będzie umieszczony nie w kilku, lecz w jednym miejscu. Aby uniknąć silnych powiązań, należy hermetyzować kod i precyzyjnie zdefiniować interfejs pozwalający na jego stosowanie.

[6] Ang. „once and only once rule". Beck K., *Extreme Programming Explained: Embrace Change*, Boston: Addison-Wesley, 2000, strony 108 – 109.

Czytelność

Czytelność jest kolejną cechą kodu, której zachowanie postulują zasady programowania inteligentnego. Czytelność jest związana z wysokim stopniem spójności. Niemniej jednak Ron Jeffries (propagator programowania ekstremalnego), przedstawiając zasadę „programowania intencyjnego”[7], posuwa się jeszcze o krok dalej. Najprościej rzecz ujmując, zasada ta stwierdza, iż jeśli podczas pisania programu należy stworzyć pewną funkcję, trzeba udać, że funkcja ta już istnieje, nadać jej nazwę „określającą jej przeznaczenie”[8], umieścić w kodzie jej wywołanie i zabrać się do dalszej pracy (zaimplementować kod funkcji później). Innymi słowy, tworzenie programu sprowadza się do napisania serii wywołań funkcji, których nazwy w czytelny sposób określają ich przeznaczenie.

Takie postępowanie pozwala na tworzenie czytelnego kodu, gdyż na poziomie większego modułu osoba analizująca kod łatwo może zrozumieć jego przeznaczenie. Do takiego postępowania zachęca także Martin Fowler, stwierdzając: „Za każdym razem, gdy poczujemy chęć skomentowania jakiegoś fragmentu kodu, zmieńmy go na funkcję”[9]. W efekcie tworzone metody są krótsze i bardziej zwięzłe (a przez to także i bardziej spójne).

Programowanie intencyjne jest bardo podobne do metody stosowanej podczas korzystania z wzorców projektowych — projektowania według interfejsu. Podając „nazwę określającą przeznaczenie” metody, tworzymy jej interfejs i nie przejmujemy się jej implementacją. Także i w tym przypadku wydaje się, że programowanie inteligentne oraz wzorce projektowe stosują podobne sposoby zapewniania wysokiej jakości kodu.

Łatwość testowania

Łatwość testowania jest niezwykle ważną cechą dobrego kodu. Jej zapewnienie jest jedyny z kluczowych założeń zasad programowania inteligentnego. Nim zacznę szczegółowo opisywać to zagadnienie, muszę jednak wyjaśnić różnice pomiędzy „łatwością testowania” a zasadą pisania testów przez stworzeniem kodu zalecaną w programowaniu ekstremalnym.

Jednym z unikalnych założeń programowania ekstremalnego jest tworzenie testów przed stworzeniem właściwego kodu. Postępowanie takie ma kilka celów:

- W efekcie uzyskujemy zbiór zautomatyzowanych testów.
- Jesteśmy zmuszeni do projektowania według interfejsu, a nie według implementacji, dzięki czemu powstające metody są lepiej hermetyzowane i cechują się mniejszym stopniem powiązań.
- Skoncentrowanie się na testach wymusza skoncentrowanie się na fragmentach kodu, które można przetestować, a to zwiększa stopień spójności i zmniejsza wzajemne powiązania różnych fragmentów kodu.

Osobiście określam kod łatwy do testowania mianem *testowalnego*. Jest to kod, który można przetestować niezależnie od pozostałych fragmentów tworzonego programu oraz bez konieczności zwracania uwagi na jego powiązania z innymi fragmentami kodu.

---

[7] Jeffries R., Anderson A., Hendrickson C., *Extreme Programming Installed*, Boston: Addison-Wesley, 2001, strony 73 – 74.

[8] Czyli nazwę, która w ścisły i precyzyjny sposób wyjaśni przeznaczenie funkcji oraz zakres jej obowiązków.

[9] Fowler M., *Refactoring: Improving the Design of Existing Code*, Boston: Addison-Wesley Longman, 1999, str. 77.

Zasada tworzenia testów przed przystąpieniem do pisania kodu leżąca u podstaw programowania ekstremalnego nieodmiennie prowadzi do powstawania kodu, który można łatwo testować[10].

Łatwość testowania w ścisły sposób łączy się z innymi praktykami:

- **Spójność** kodu ułatwia jego testowanie, gdyż dany fragment kodu dotyczy tylko jednej operacji.
- Kod **o niskim stopniu powiązań** jest łatwiejszy do testowania w porównaniu z kodem, w którym występują ścisłe powiązania, gdyż jest on niemal pozbawiony interakcji z innymi fragmentami kodu.
- Fakt **powtarzania się kodu** nie ma wpływu na łatwość testowania, lecz wymusza wykonywanie większej ilości testów. Oznacza to, że wraz ze wzrostem ilości powtarzającego się kodu zmniejsza się łatwość testowania całego systemu.
- Kod o **dużej czytelności** jest jednocześnie łatwiejszy do testowania, gdyż nazwy metod oraz ich parametry w precyzyjny sposób określają ich przeznaczenie.
- Kod **hermetyzowany** jest łatwiejszy do testowania, gdyż jest on w bardzo niewielkim stopniu powiązany z innymi fragmentami kodu.

Oto przykład potwierdzający powyższe stwierdzenia. Miałem kiedyś klienta, z którym, przed przeprowadzeniem kursu pt. „Efektywna analiza i projektowanie obiektowe", omawiałem zagadnienia testowania. Powiedział mi, bym nie poświęcał zbyt dużo uwagi testom jednostkowym, gdyż ma z nimi złe doświadczenia. Kiedy spytałem, co się stało, odpowiedział, że podczas prób zastosowania testów jednostkowych przy okazji tworzenia wcześniejszego projektu okazało się, iż jest to bardzo trudne zadanie. Aby napisać testy, trzeba było napisać specjalne narzędzia umożliwiające tworzenie obiektów, które chciał testować w prosty i szybki sposób, a obiekty te były powiązane z innymi obiektami.

W odpowiedzi zapytałem, czy przed przystąpieniem do pisania kodu zastanowił się nad sposobem, w jaki ten kod będzie testowany. Mój rozmówca odparł, iż nie zastanawiał się nad tym. Zapytałem wtedy, czy gdyby *uwzględnił* sposób testowania kodu, napisałby go w inny sposób. Mój rozmówca zamilkł i uświadomił sobie, że gdyby zastanowił się nad sposobem testowania kodu, mógłby poprawić jakość swojego projektu.

Wielu programistów posuwa się jeszcze o krok dalej i, tworząc kod, w całości bazuje na testach. Metodologia ta jest określana jako „programowanie w oparciu o testy" (ang. *test-driven developnent*, w skrócie TDD), a jej przedstawienie wykracza poza ramy niniejszej książki. Osobiście stosunkowo często stosowałem tę metodę i uważam, że jest ona doskonała. Podobnie jak inne inteligentne metody także i programowanie w oparciu o testy początkowo wydaje się być sprzeczne z wzorcami projektowymi, jednak tak nie jest. Metoda ta opiera się na tych samych zasadach co wzorce, a jedynie samo podejście do tworzenia kodu jest w niej inne.

[10] Ze względu na fakt, iż niniejsza książka jest poświęcona wzorcom projektowym, nie będę w niej opisywał doskonałych zasad stosowania testów jednostkowych i projektowania w oparciu o testy.

Podsumowanie

W rozdziale

Tradycyjny sposób rozumienia obiektów, hermetyzacji oraz dziedziczenia jest stosunkowo ograniczony. Możliwości zastosowania hermetyzacji stają się dużo szersze i wykraczają poza ukrywanie danych dzięki rozszerzeniu jej definicji na ukrywanie w ogólności. Hermetyzacja służyć może do tworzenia warstw obiektów, co umożliwia wprowadzanie zmian po jednej ze stron warstwy pozostających bez wpływu na obiekty po drugiej stronie warstwy.

Dziedziczenie natomiast lepiej jest stosować jako sposób organizacji klas konkretnych powiązanych w warstwie koncepcji niż tylko jako środek służący specjalizacji.

Koncepcja zastosowania obiektów dla reprezentacji zmienności zachowania innych obiektów nie różni się od powszechnej praktyki stosowania zmiennych składowych dla reprezentacji zmienności danych. W obu przypadkach wykorzystywana jest hermetyzacja zawartego obiektu bądź zmiennej, co umożliwia bezproblemową rozbudowę.

Analiza wspólności i zmienności pozwala na bardziej efektywne określanie obiektów występujących w dziedzinie problemu niż metoda bazująca na poszukiwaniu rzeczowników i czasowników.

Cechy kodu tworzonego w oparciu o metody zalecane przez programowanie inteligentne, a w szczególności przez programowanie ekstremalne, dokładnie odpowiadają cechom, które staramy się uzyskać, tworząc kod o wysokim stopniu spójności i hermetyzacji oraz niewielkich powiązaniach.

Pytania kontrolne

Obserwacje

1. Jaki jest poprawny sposób myślenia o hermetyzacji?
2. Jakie są trzy perspektywy analizy problemu? (Być może, odpowiadając na to pytanie, czytelnik będzie musiał zajrzeć do rozdziału 1., „Obiektowość”.)

Interpretacje

1. Obiekty można wyobrażać sobie na dwa sposoby: jako „dane z metodami” oraz „byty posiadające określoną odpowiedzialność”.
   - ♦ Co sprawia, że ten drugi sposób wyobrażania sobie obiektów jest lepszy od pierwszego?
   - ♦ Jakie dodatkowe aspekty obiektów można dzięki niemu zrozumieć?

2. Czy jeden obiekt może zawierać inne obiekty? Czy taki obiekt umieszczony wewnątrz innego obiektu w jakikolwiek sposób różni się od składowej zmiennej?
3. Jakie jest znaczenie zalecenia: „znajdź to, co się zmienia, i hermetyzuj to”? Proszę podać przykład.
4. Proszę podać związki pomiędzy analizą wspólności i zmienności oraz trzema perspektywami, z jakich można analizować obiekty.
5. Klasa abstrakcyjna odpowiada „głównemu pojęciu łączącemu”. Co to oznacza?
6. „Analiza zmienności wyjaśnia, czym różnią się od siebie poszczególni członkowie rodziny. Zmienność ma sens wyłącznie w granicach określonej cechy wspólnej”.
   - ♦ Co to oznacza?
   - ♦ Jakie typy obiektów są używane do reprezentacji wspólnych pojęć?
   - ♦ Jakie typy obiektów są używane do reprezentacji zmienności?

Opinie i zastosowania

1. Dlaczego koncentrowanie się na motywacjach jest lepsze od koncentrowania się na implementacji? Proszę podać przykłady sytuacji, w których takie podejście okazało się pomocne.
2. Z góry przyjęte sposoby pojmowania ograniczają nasze możliwości rozumienia pojęć. Stwierdzenie to okazało się prawdziwe w odniesieniu do hermetyzacji. Czy czytelnik jest w stanie podać przykład, gdy z góry przyjęte wyobrażenia miały wpływ na zrozumienie wymagań projektu? Jakie były skutki i jak udało się rozwiązać problemy?
3. Termin *dziedziczenie* określa zarówno sytuację, w której tworzona jest nowa klasa, wyspecjalizowana klasa dziedzicząca po pewnej klasie nieabstrakcyjnej, jak również, gdy na podstawie abstrakcyjnej klasy bazowej zostaje utworzona klasa stanowiąca zupełnie nową implementację. Czy nie byłoby lepiej, gdyby istniały dwa odrębne terminy opisujące te pojęcia?
4. W jaki sposób można zastosować analizę stałości i zmienności, aby ułatwić sobie opracowywanie sposobów modyfikacji systemu?
5. Ważne jest, by zmienności poszukiwać jak najwcześniej oraz jak najczęściej. Czy czytelnik uważa, iż stwierdzenie to jest prawdziwe? Proszę uzasadnić odpowiedź. Dlaczego odnajdywanie zmienności może pomóc unikać problemów?
6. Analiza zmienności i stałości jest jednym z podstawowych narzędzi służących do określania obiektów, znacznie lepszym od metody polegającej na wyszukiwaniu rzeczowników. Czy czytelnik zgadza się z tym stwierdzeniem? Proszę uzasadnić odpowiedź.
7. W niniejszym rozdziale starałem się przedstawić nowe spojrzenie na obiekty. Czy mi się to udało? Proszę uzasadnić odpowiedź.