Michael_Feathers

PRACA Z ZASTANYM KODEM

NAJLEPSZE TECHNIKI

NAUCZ SIĘ PRACOWAĆ NA GOTOWYCH PROJEKTACH!

Helion

Tytuł oryginału: Working Effectively with Legacy Code

Tłumaczenie: Ireneusz Jakóbik

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-246-8317-8

Authorized translation from the English language edition, entitled: WORKING EFFECTIVELY WITH LEGACY CODE; ISBN 0131177052; by Michael C. Feathers; published by Pearson Education, Inc, publishing as Prentice Hall.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/prazak.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/prazak
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Rozdział 9.

Nie mogę umieścić tej klasy w jarzmie testowym

Będzie ciężko. Gdyby utworzenie instancji klasy w jarzmie testowym zawsze było łatwe, książka ta byłaby o wiele krótsza. Niestety, często zadanie to jest trudne.

Oto cztery najczęściej występujące problemy, które napotykamy:

1. Nie da się w prosty sposób utworzyć obiektów klasy.
2. Nie da się w prosty sposób przeprowadzić procesu budowy jarzma testowego z umieszczoną w nim klasą.
3. Korzystanie z konstruktora, którego potrzebujemy użyć, wywołuje skutki uboczne.
4. Konstruktor wykonuje sporo pracy, a my musimy ją rozpoznać.

W rozdziale tym zajmiemy się serią przykładów, które skupiają się na tych problemach, z uwzględnieniem różnych języków. Istnieje więcej niż tylko jeden sposób poradzenia sobie z każdym z tych problemów. Zaznajomienie się z tymi przykładami jest jednak niezłą metodą na poznanie całego arsenału technik usuwania zależności i nauczenia się, które z nich wybrać i jak je stosować w określonych sytuacjach.

Przypadek irytującego parametru

Kiedy muszę wprowadzić zmianę w cudzym systemie, zwykle początkowo jestem nastawiony bardzo optymistycznie. Nie mam pojęcia dlaczego. Na ile tylko mogę, próbuję być realistą, ale optymizm zawsze się przebija. „Hej”, mówię do siebie (albo do kolegi), „wygląda na to, że będzie łatwo. Musimy tylko `coś tam` trochę `stentegować`, i po robocie”. Wszystko to brzmi prosto, kiedy się o tym mówi, aż bierzemy się do klasy `CośTam` (czymkolwiek by ona była) i się jej przyglądamy. „No dobra. Musimy więc dodać metodę tutaj, zmienić inną metodę tam i oczywiście wrzucić całość do jarzma testowego”. W tym

momencie zaczynam nieco wątpić. „Motyla noga! Wygląda na to, że najprostszy konstruktor w tej klasie przyjmuje trzy parametry, ale — dodaję optymistycznie — być może utworzenie obiektu wcale nie będzie takie trudne".

Zajmijmy się przykładem i zobaczmy, czy mój optymizm ma podstawy, czy jest tylko mechanizmem obronnym.

W kodzie systemu realizującego płatności znajduje się nieprzetestowana klasa Javy o nazwie `CreditValidator`.

```
public class CreditValidator
{
    public CreditValidator(RGHConnection connection,
                           CreditMaster master,
                           String validatorID) {
        ...
    }
    Certificate validateCustomer(Customer customer)
           throws InvalidCredit {
        ...
    }
    ...
}
```

Jedną z wielu funkcji tej klasy jest informowanie nas, czy klienci mają otwarty kredyt. Jeśli tak, otrzymujemy certyfikat informujący o wysokości kredytu. Jeśli nie, klasa zgłasza wyjątek.

Nasza misja, o ile tylko ją przyjmiemy, polega na dodaniu do tej klasy nowej metody. Metoda będzie nosić nazwę `getValidationPercent`, a jej zadaniem będzie informowanie nas o odsetku udanych wywołań metody `validateCustomer` w czasie działania walidatora.

Od czego zaczniemy?

Kiedy musimy utworzyć obiekt w jarzmie testowym, często najlepszym podejściem jest po prostu próba jego utworzenia. Moglibyśmy przeprowadzić rozległą analizę, aby dowiedzieć się, dlaczego będzie (albo też nie będzie) to łatwe bądź trudne, ale równie proste będzie utworzenie klasy testowej jUnit, wprowadzenie do niej kodu i skompilowanie go.

```
public void testCreate() {
    CreditValidator validator = new CreditValidator();
}
```

> Najlepszym sposobem na stwierdzenie, czy będziesz mieć kłopoty podczas tworzenia instancji klasy w jarzmie testowym, jest po prostu próba jej utworzenia. Napisz przypadek testowy i spróbuj w jego ramach utworzyć obiekt. Kompilator powie Ci, czego potrzebujesz, aby odnieść sukces.

To jest test konstrukcyjny. Testy konstrukcyjne wyglądają trochę dziwnie. Kiedy piszę taki test, zwykle nie umieszczam w nim asercji. Po prostu próbuję skonstruować obiekt. Później, gdy mam już możliwość tworzenia obiektów w jarzmie testowym, zwykle pozbywam się takiego testu albo zmieniam jego nazwę, żebym mógł go wykorzystać do sprawdzenia czegoś ważniejszego.

Wróćmy jednak do naszego przykładu.

Do konstruktora nie dodaliśmy jeszcze żadnych argumentów, więc kompilator narzeka. Mówi nam, że dla klasy `CreditValidator` nie ma domyślnego konstruktora. Szperając w kodzie, odkrywamy, że potrzebujemy klasy `RGHConnection`, klasy `CreditMaster` oraz hasła. Klasy te mają tylko po jednym konstruktorze i wyglądają następująco:

```
public class RGHConnection
{
    public RGHConnection(int port, String Name, string passwd)
         throws IOException {
        ...
    }
}

public class CreditMaster
{
    public CreditMaster(String filename, boolean isLocal) {
        ...
    }
}
```

Kiedy konstruowany jest obiekt klasy `RGHConnection`, łączy się on z serwerem. Podczas połączenia pobierane są z serwera wszystkie dane potrzebne do zweryfikowania kredytu klienta.

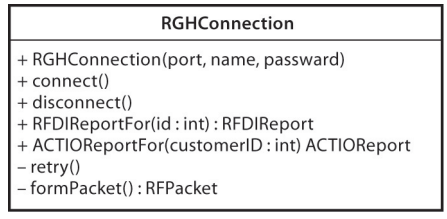
Druga klasa, `CreditMaster`, przekazuje nam informacje polityczne, z których korzystamy, podejmując decyzje kredytowe. Podczas konstruowania obiekt klasy `CreditMaster` wczytuje informacje z pliku i zapisuje je w pamięci, abyśmy mogli z nich skorzystać.

Wygląda więc na to, że umieszczenie tej klasy w jarzmie testowym będzie dość łatwe, prawda? Nie tak szybko. Możemy napisać test, ale czy damy radę z nim pracować?

```
public void testCreate() throws Exception {
    RGHConnection connection = new RGHConnection(DEFAULT_PORT,
                                                 "admin", "rii8ii9s");
    CreditMaster master = new CreditMaster("crm2.mas", true);
    CreditValidator validator = new CreditValidator(
                                        connection, master, "a");
}
```

Okazuje się, że nawiązywanie przez metodę `RGHConnection` połączenia z serwerem w czasie testu nie jest dobrym pomysłem. Zabiera to sporo czasu, a serwer nie zawsze odpowiada. Z kolei klasa `CreditMaster` nie stwarza problemów. Kiedy tworzymy jej instancję, plik jest wczytywany szybko, poza tym jest on tylko do odczytu, w związku z czym nie musimy się obawiać, że testy go uszkodzą.

Kiedy chcemy utworzyć walidator, prawdziwą przeszkodę stanowi klasa `RGHConnection` — jest ona **irytującym parametrem**. Gdybyśmy mogli utworzyć jakiś rodzaj fałszywego obiektu tej klasy i sprawić, że `CreditValidator` uwierzy, iż komunikuje się z autentycznym obiektem, moglibyśmy uniknąć całej masy problemów związanych z połączeniem. Spójrzmy na metody, które udostępnia klasa `RGHConnection` (rysunek 9.1).

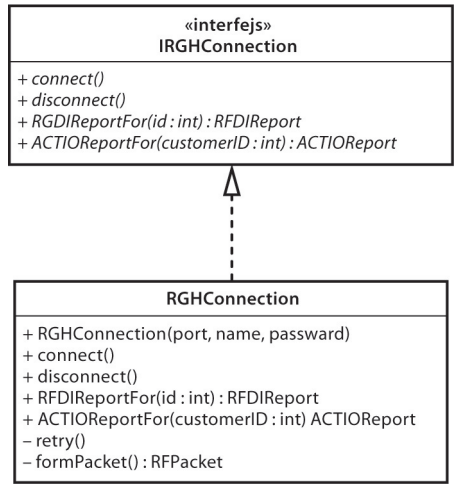


Rysunek 9.1. *Klasa RGHConnection*

Wygląda na to, że klasa `RGHConnection` zawiera zbiór metod, które obsługują mechanizm nawiązywania połączenia: `connect`, `disconnect` i `retry`, a także kilka metod biznesowych, takich jak `RFDIReportFor` i `ACTIOReportFor`. Pisząc naszą nową metodę dla klasy `CreditValidator`, będziemy musieli wywołać metodę `RFDIReportFor`, aby uzyskać wszystkie potrzebne nam informacje. Zazwyczaj dane te pochodzą z serwera, ale ponieważ chcemy uniknąć nawiązywania rzeczywistego połączenia, będziemy musieli znaleźć jakiś sposób na ich udostępnienie przez nas.

W tym przypadku najlepszą metodą na utworzenie fałszywego obiektu będzie **wyodrębnienie interfejsu** (361) w odniesieniu do klasy `RGHConnection`. Jeśli dysponujesz narzędziem, które wspiera refaktoryzację, prawdopodobnie udostępnia ono **wyodrębnianie interfejsu**. Jeżeli Twoje środowisko programistyczne nie wspiera tej techniki, pamiętaj, że **wyodrębnianie interfejsu** jest na tyle proste, że można je wykonać ręcznie.

Po **wyodrębnieniu interfejsu** (361) otrzymujemy taką strukturę, jak pokazano na rysunku 9.2.



Rysunek 9.2. *Klasa RGHConnection po wyodrębnieniu interfejsu*

Możemy przystąpić do pisania testów, tworząc niedużą fałszywą klasę, która udostępnia potrzebne nam raporty:

```
public class FakeConnection implements IRGHConnection
{
```

```
    public RFDIReport report;

    public void connect() {}
    public void disconnect() {}
    public RFDIReport RFDIReportFor(int id) { return report; }
    public ACTIOReport ACTIOReportFor(int customerID) { return null; }
}
```

Mając tę klasę, możemy rozpocząć tworzenie takich testów:

```
void testNoSuccess() throws Exception {
    CreditMaster master = new CreditMaster("crm2.mas", true);
    IRGHConnection connection = new FakeConnection();
    CreditValidator validator = new CreditValidator(
                                        connection, master, "a");
    connection.report = new RFDIReport(...);

    Certificate result = validator.validateCustomer(new Customer(...));

    assertEquals(Certificate.VALID, result.getStatus());
}
```

Klasa `FakeConnection` jest trochę dziwna. Jak często w ogóle piszemy metody, które nie mają żadnego ciała i tylko zwracają pustą wartość? Co gorsza, ma ona publiczną zmienną, której każdy może nadać taką wartość, jaką tylko zechce. Wydaje się, że klasa ta narusza wszelkie obowiązujące reguły. Otóż w rzeczywistości ich wcale nie narusza. Reguły dla klas umożliwiających przeprowadzanie testów są inne. Kod klasy `FakeConnection` nie jest kodem produkcyjnym. Nigdy nie zostanie uruchomiony w naszej pełnej aplikacji — będzie działać wyłącznie w jarzmie testowym.

Teraz, kiedy możemy już utworzyć walidator, mamy możliwość napisania metody `getValidationPercent`. Oto test, który ją weryfikuje:

```
void testAllPassed100Percent() throws Exception {
    CreditMaster master = new CreditMaster("crm2.mas", true);
    IRGHConnection connection = new FakeConnection("admin", "rii8ii9s");
    CreditValidator validator = new CreditValidator(
         connection, master, "a");
    connection.report = new RFDIReport(...);
    Certificate result = validator.validateCustomer(new Customer(...));
    assertEquals(100.0, validator.getValidationPercent(), THRESHOLD);
}
```

Kod testowy a kod produkcyjny

Kod testowy nie musi spełniać tych samych standardów co kod produkcyjny. Zazwyczaj nie mam nic przeciwko naruszaniu zasady hermetyzacji poprzez tworzenie zmiennych publicznych, jeśli uprości to pisanie testów. Kod testowy powinien być jednak przejrzysty; powinien być łatwy do zrozumienia i prosty w modyfikowaniu.

Spójrz na testy `testNoSuccess` i `testAllPassed100Percent` w tym przykładzie. Czy zawierają one powielony kod? Tak. Powtórzone są trzy pierwsze wiersze. Powinny one zostać wyodrębnione i umieszczone w jednym miejscu — metodzie `setUp()` tej klasy testowej.

Test ten sprawdza, czy procent uwierzytelnienia wynosi mniej więcej 100.0, gdy otrzymujemy pojedynczy, ważny certyfikat kredytu.

Test działa poprawnie, ale pisząc kod metody `getValidationPercent`, zauważamy coś interesującego. Okazuje się, że nie będzie ona w ogóle używać metody `CreditMaster`. Dlaczego więc ją piszemy i przekazujemy do obiektu klasy `CreditValidator`? Może wcale nie musimy tego robić? Instancję klasy `CreditValidator` moglibyśmy utworzyć w naszym teście następująco:

```
CreditValidator validator = new CreditValidator(connection, null, "a");
```

Czy jeszcze się nie pogubiłeś?

Sposób, w jaki ludzie reagują na tego typu kod, sporo mówi o systemach, z którymi oni pracują. Jeżeli widząc ten kod, powiedziałeś: „O, świetnie. Do konstruktora przekazywana jest wartość `null`. W naszym systemie ciągle tak robimy", prawdopodobnie zajmujesz się dość paskudnym systemem. Zapewne wszędzie masz w nim porozrzucane instrukcje sprawdzające występowanie pustej wartości i pełno kodu warunkowego określającego, co można, a co trzeba z nią zrobić. Z drugiej jednak strony, jeśli spojrzałeś na powyższy kod i stwierdziłeś: „Z tym facetem chyba jest coś nie tak! Przekazywanie w systemie wartości `null`? Czy on w ogóle ma o czymkolwiek pojęcie?" — otóż tym z Was z tej drugiej grupy (a przynajmniej tym, którzy nadal to czytają i nie zamknęli z rozmachem tej książki w księgarni) — chciałbym powiedzieć coś takiego: pamiętajcie, że robimy to tylko w testach. Najgorsze, co się może stać, to to, że jakiś fragment kodu spróbuje skorzystać z tej zmiennej. W naszym przypadku środowisko uruchomieniowe Javy zgłosi wyjątek. Ponieważ jarzmo wyłapuje wszystkie wyjątki zgłaszane podczas testów, dość szybko dowiemy się, czy jakiś parametr jest w ogóle używany.

Przekazywanie pustej wartości

Kiedy piszesz testy, a pewien obiekt wymaga parametru, który jest trudny do utworzenia, rozważ przekazanie po prostu pustej wartości. Jeżeli parametr ten zostanie użyty podczas wykonywania się testu, kod zgłosi wyjątek, który zostanie wychwycony przez jarzmo testowe. Jeśli musisz mieć zachowanie, które istotnie wymaga obiektu, to będziesz mógł go skonstruować i przekazać jako parametr.

Przekazywanie pustej wartości jest w niektórych językach bardzo wygodną techniką. Dobrze się ona sprawdza w Javie i C# oraz niemal każdym języku, który zgłasza wyjątek, gdy w czasie działania programu zostanie użyta pusta referencja. Z tego wynika, że przekazywanie pustej wartości nie jest dobrym pomysłem w przypadku C i C++, chyba że masz pewność, iż program uruchomieniowy wykryje błędy związane z pustymi wskaźnikami. W przeciwnym razie będziesz mieć do czynienia z testami, które się tajemniczo wysypują — o ile będziesz mieć szczęście. Jeśli zabraknie Ci szczęścia, Twoje testy będą po prostu bezobjawowo i beznadziejnie złe. W czasie działania będą niszczyć pamięć, a Ty nigdy się o tym nie dowiesz.

Kiedy pracuję w Javie, często zaczynam od następującego testu, a parametry uzupełniam w miarę potrzeb.

```
public void testCreate() {
    CreditValidator validator = new CreditValidator(null, null, "a");
}
```

Najważniejsze do zapamiętania jest to, aby nie przekazywać pustej wartości w kodzie produkcyjnym, chyba że nie masz innego wyboru. Wiem, że niektóre biblioteki tego od Ciebie oczekują, ale kiedy piszesz nowy kod, masz lepszą alternatywę. Jeśli kusi Cię użycie pustej wartości w kodzie produkcyjnym, znajdź miejsca, w których są one zwracane i pobierane, po czym weź pod uwagę inne rozwiązanie. Zastanów się nad użyciem **wzorca pusty obiekt**.

Wzorzec pusty obiekt

Wzorzec pusty obiekt to sposób na uniknięcie użycia pustych wartości w programach. Mamy na przykład metodę, która zwraca dane pracownika po otrzymaniu jego numeru identyfikacyjnego. Co powinno zostać zwrócone, jeśli nie ma pracownika o podanym numerze?

```
for(Iterator it = idList.iterator(); it.hasNext(); ) {
    EmployeeID id = (EmployeeID)it.next();
    Employee e = finder.getEmployeeForID(id);
    e.pay();
}
```

Mamy kilka możliwości. Możemy podjąć decyzję o zgłaszaniu wyjątków, dzięki czemu nie będzie trzeba niczego zwracać, ale takie rozwiązanie zmusiłoby klienty do jawnej obsługi błędów. Moglibyśmy także zwracać pustą wartość, lecz wówczas klienty musieliby jawnie sprawdzać, czy jej nie otrzymują.

Jest jeszcze trzecie rozwiązanie. Czy powyższy kod tak naprawdę sprawdza, czy istnieje pracownik, któremu należy zapłacić? Czy musi to robić? A gdybyśmy tak mieli klasę o nazwie `NullEmployee`? Instancja tej klasy nie ma nazwiska ani adresu, a kiedy polecisz jej dokonanie wypłaty, po prostu nic nie zrobi.

W takich kontekstach puste obiekty mogą być przydatne — pomagają one chronić klienty przed jawną obsługą błędów. Chociaż puste obiekty są pomocne, powinieneś zachować ostrożność, gdy z nich korzystasz. Oto przykład niewłaściwego sposobu liczenia pracowników, którym wypłacono wynagrodzenie:

```
int employeesPaid = 0;
for(Iterator it = idList.iterator(); it.hasNext(); ) {
    EmployeeID id = (EmployeeID)it.next();
    Employee e = finder.getEmployeeForID(id);
    e.pay();
    mployeesPaid++; // błąd!
}
```

Jeśli któryś ze zwróconych pracowników jest pracownikiem pustym, to zliczenie będzie błędne.

Puste obiekty są przydatne zwłaszcza wtedy, gdy klient nie musi sprawdzać, czy dana operacja się powiodła. W wielu przypadkach możemy tak dopracować nasz projekt, abyśmy mieli do czynienia z takim właśnie rozwiązaniem.

Przekazanie pustej wartości i **wyodrębnienie interfejsu** (361) to dwa sposoby na poradzenie sobie z irytującymi parametrami. Czasami można jednak skorzystać z jeszcze innej możliwości. Jeżeli sprawiająca problemy zależność w parametrze nie jest bezpośrednio zakodowana w swoim konstruktorze, w celu pozbycia się jej możemy skorzystać z techniki **tworzenia podklasy i przesłaniania metody** (398). W tym przypadku rozwiązanie takie byłoby możliwe. Jeśli konstruktor klasy `RGHConnection` używa metody `connect` w celu nawiązania połączenia, moglibyśmy pozbyć się zależności, przesłaniając wywołanie `connect()` w testowanej podklasie. **Tworzenie podklasy i przesłanianie metody** (398) może być bardzo przydatnym sposobem usuwania zależności, ale musimy mieć pewność, że nie zmieniamy zachowania, które chcemy przetestować, gdy z niego korzystamy.

Przypadek ukrytej zależności

Niektóre klasy bywają podstępne. Patrzymy na nie, znajdujemy konstruktor, którego chcemy użyć, i próbujemy go wywołać. Wtedy trach! Napotykamy przeszkodę. Jedną z najczęściej występujących przeszkód jest **ukryta zależność** — konstruktor korzysta z zasobów, do których nie mamy łatwego dostępu w naszym jarzmie testowym. Zobaczymy taką sytuację w następnym przykładzie; źle zaprojektowaną klasę w C++, która obsługuje listę mailingową:

```
class mailing_list_dispatcher
{
public:
       mailing_list_dispatcher ();
    virtual ~mailing_list_dispatcher;

    void send_message(const std::string& message);
    void add_recipient(const mail_txm_id id,
          const mail_address& address);
    ...

    private:
        mail_service *service;
        int status;
};
```

Oto fragment konstruktora tej klasy. Alokuje ona obiekt `mail_service` za pomocą instrukcji `new` na liście inicjatora konstruktora. To kiepski styl, ale potem jest jeszcze gorzej. Konstruktor wykonuje sporo szczegółowej pracy z tym obiektem; korzysta też z magicznej liczby 12. Co ma oznaczać to 12?

```
mailing_list_dispatcher::mailing_list_dispatcher()
: service(new mail_service), status(MAIL_OKAY)
{
    const int client_type = 12;
    service->connect();
    if (service->get_status() == MS_AVAILABLE) {
```

```
        service->register(this, client_type, MARK_MESSAGES_OFF);
        service->set_param(client_type, ML_NOBOUNCE | ML_REPEATOFF);
    }
    else
        status = MAIL_OFFLINE;
    ...
}
```

Podczas testu możemy utworzyć instancję tej klasy, ale chyba nie przyniesie nam ona większych korzyści. Przede wszystkim musimy połączyć się z bibliotekami pocztowymi i skonfigurować system pocztowy, aby obsługiwał rejestrowanie. Jeśli w trakcie testów użyjemy funkcji `send_message`, to naprawdę wyślemy maile do ludzi. Automatyczne testowanie tej funkcjonalności będzie trudne, chyba że skonfigurujemy specjalną skrzynkę pocztową i będziemy się z nią regularnie łączyć, czekając na nadejście wiadomości. Takie rozwiązanie byłoby dobre podczas całościowych testów systemu, ale wszystko, co chcemy teraz zrobić, to tylko dodać do klasy kilka przetestowanych funkcjonalności, więc sposób ten byłby lekką przesadą. Jak moglibyśmy utworzyć prosty obiekt w celu dodania jakiejś nowej funkcjonalności?

Podstawowy problem w tym przypadku polega na tym, że zależność od obiektu `mail_service` jest ukryta w konstruktorze `mailing_list_dispatcher`. Gdyby istniał jakiś sposób na zastąpienie tego obiektu fałszywką, moglibyśmy dokonać rozpoznania, posługując się fałszywym obiektem i uzyskać informację zwrotną podczas modyfikowania klasy.

Jedną z technik, które możemy wykorzystać, jest **parametryzacja konstruktora** (377). Przy jej pomocy wyciągamy na zewnątrz zależność istniejącą w konstruktorze, przekazując ją do konstruktora.

Oto jak wygląda kod konstruktora po **sparametryzowaniu konstruktora** (377):

```
mailing_list_dispatcher::mailing_list_dispatcher(mail_service *service)
: status(MAIL_OKAY)
{
    const int client_type = 12;
    service->connect();
    if (service->get_status() == MS_AVAILABLE) {
        service->register(this, client_type, MARK_MESSAGES_OFF);
        service->set_param(client_type, ML_NOBOUNCE | ML_REPEATOFF);
    }
    else
        status = MAIL_OFFLINE;
    ...
}
```

Jedyna różnica tak naprawdę sprowadza się do tego, że obiekt `mail_service` jest tworzony poza klasą i do niej przekazywany. Być może nie wygląda to na znaczne usprawnienie, ale teraz mamy ogromne możliwości. Możemy skorzystać z **wyodrębniania interfejsu** (361), aby uzyskać interfejs dla `mail_service`. Jeden z implementerów tego interfejsu może być klasą produkcyjną, która faktycznie wysyła maile. Inny może być fałszywą klasą, rozpoznającą to, co robimy podczas testów, oraz informującą nas, że istotnie to się stało.

Parametryzacja konstruktora (377) jest bardzo wygodnym sposobem na przesunięcie zależności z konstruktora na zewnątrz, jednak programiści nie biorą go zbyt często pod uwagę. Jedną z przeszkód stanowi fakt, że wiele osób sądzi, iż w celu przekazania nowego parametru konieczna będzie zmiana wszystkich klientów klasy, co jednak nie jest prawdą. Możemy sobie z tym poradzić następująco. Najpierw wyodrębniamy ciało konstruktora i tworzymy z niego nową metodę o nazwie `initialize`. W przeciwieństwie do większości innych sposobów wyodrębniania metod możemy to całkiem bezpiecznie zrobić bez testów, gdyż w trakcie tej czynności **zachowujemy sygnatury** (314).

```
void mailing_list_dispatcher::initialize(mail_service *service)
{
    status = MAIL_OKAY;
    const int client_type = 12;
    service.connect();
    if (service->get_status() == MS_AVAILABLE) {
        service->register(this, client_type, MARK_MESSAGES_OFF);
        service->set_param(client_type, ML_NOBOUNCE | ML_REPEATOFF);
    }
    else
        status = MAIL_OFFLINE;
    ...
}

mailing_list_dispatcher::mailing_list_dispatcher(mail_service *service)
{
    initialize(service);
}
```

Teraz możemy udostępnić konstruktor, który ma oryginalną sygnaturę. Testy mogą wywoływać konstruktor sparametryzowany przez `mail_service`, natomiast klienty mogą wywoływać go w poniższy sposób; nie muszą wiedzieć, że coś uległo zmianie.

```
mailing_list_dispatcher::mailing_list_dispatcher()
{
    initialize(new mail_service);
}
```

Tego typu refaktoryzacja jest jeszcze łatwiejsza w takich językach jak C# i Java, ponieważ możemy w nich wywoływać konstruktory z poziomu innych konstruktorów.

Gdybyśmy na przykład robili coś podobnego w C#, wynikowy kod mógłby wyglądać następująco:

```
public class MailingListDispatcher
{
    public MailingListDispatcher()
    : this(new MailService())
    {}

    public MailingListDispatcher(MailService service) {
        ...
    }
}
```

Z zależnościami ukrytymi w konstruktorach można sobie radzić za pomocą wielu technik. Zwykle możemy zastosować **wyodrębnianie i przesłanianie gettera** (353), **wyodrębnianie i przesłanianie metody wytwórczej** (351) oraz **zastępowanie zmiennej instancji** (401), najbardziej jednak lubię korzystać z **parametryzacji konstruktora** (377). Kiedy w konstruktorze tworzony jest obiekt i nie ma on żadnych zależności konstrukcyjnych, łatwą do zastosowania techniką będzie właśnie **parametryzacja konstruktora**.

Przypadek konstrukcyjnego kłębowiska

Parametryzacja konstruktora (377) jest jedną z najprostszych technik, z których możemy skorzystać w celu usunięcia zależności ukrytych w konstruktorze. Jest też techniką, po którą często sięgam w pierwszej kolejności. Niestety, nie zawsze jest ona najlepszym wyborem. Jeśli konstruktor tworzy sporą liczbę obiektów lub ma dostęp do wielu zmiennych globalnych, możemy w rezultacie uzyskać bardzo długą listę parametrów. W gorszych sytuacjach konstruktor tworzy kilka obiektów, po czym używa ich do utworzenia kolejnych obiektów, tak jak poniżej:

```
class WatercolorPane
{
public:
    WatercolorPane(Form *border, WashBrush *brush, Pattern *backdrop)
   {
        ...
        anteriorPanel = new Panel(border);
        anteriorPanel->setBorderColor(brush->getForeColor());
        backgroundPanel = new Panel(border, backdrop);
        cursor = new FocusWidget(brush, backgroundPanel);
        ...
    }
    ...
}
```

Gdybyśmy chcieli zrobić rozpoznanie, posługując się obiektem `cursor`, mielibyśmy problem. Obiekt ten jest osadzony w kłębowisku tworzonych obiektów. Moglibyśmy spróbować przenieść poza klasę cały kod użyty do tworzenia kursora. W dalszej kolejności klient mógłby utworzyć kursor i przekazać go jako argument. Nie będzie to jednak bezpieczne, jeśli nie mamy na miejscu testów, poza tym rozwiązanie takie byłoby sporym utrudnieniem dla klientów tej klasy.

Jeśli dysponujemy narzędziem refaktoryzującym, które bezpiecznie wyodrębnia metody, możemy skorzystać z techniki **wyodrębniania i przesłaniania metody wytwórczej** (351) w odniesieniu do kodu konstruktora, co jednak nie sprawdzi się we wszystkich językach. Możemy tak postąpić w Javie i C#, ale już C++ nie pozwala wywoływać funkcji wirtualnych w konstruktorach w celu odwołania się do wirtualnych funkcji zdefiniowanych w klasach pochodnych. Poza tym tak w ogóle to nie jest dobry pomysł. Funkcje w klasach pochodnych często zakładają, że mogą korzystać ze zmiennych w ich klasie bazowej.

Dopóki konstruktor klasy bazowej nie zakończy w pełni swojej pracy, istnieje ryzyko, że przesłonięta funkcja, która go wywołuje, może uzyskać dostęp do niezainicjalizowanej zmiennej.

Inną opcją jest **zastępowanie zmiennej instancji** (401). Piszemy setter dla klasy, który umożliwi nam podstawienie innej instancji po skonstruowaniu obiektu.

```
class WatercolorPane
{
public:
    WatercolorPane(Form *border, WashBrush *brush, Pattern *backdrop)
    {
        ...
        anteriorPanel = new Panel(border);
        anteriorPanel->setBorderColor(brush->getForeColor());
        backgroundPanel = new Panel(border, backdrop);

        cursor = new FocusWidget(brush, backgroundPanel);
        ...
    }

    void supersedeCursor(FocusWidget *newCursor)
    {
        delete cursor;
        cursor = newCursor;
    }
}
```

Musimy być bardzo ostrożni, gdy stosujemy tę technikę refaktoryzacji w C++. Kiedy zastępujemy obiekt, powinniśmy pozbyć się jego starej instancji. Często oznacza to, że musimy skorzystać z operatora `delete` w celu wywołania destruktora i zniszczenia pamięci obiektu. Kiedy to robimy, musimy wiedzieć, co robi destruktor i czy niszczy on coś, co zostało przekazane konstruktorowi obiektu. Jeśli nie będziemy ostrożni podczas czyszczenia pamięci, możemy spowodować pewne subtelne błędy.

W większości innych języków **zastępowanie zmiennej instancji** (401) jest dość proste w użyciu. Oto wynik zastosowania tej techniki zapisany w Javie. Nie musimy robić nic szczególnego, aby pozbyć się obiektu, do którego odwołuje się `cursor` — proces odśmiecania pamięci i tak w końcu się go pozbędzie. Powinniśmy jednak być szczególnie uważni, aby nie korzystać z tej metody w kodzie produkcyjnym. Jeżeli obiekty, które zastępujemy, zarządzają innymi zasobami, możemy spowodować całkiem poważne problemy związane z dostępem do zasobów.

```
void supersedeCursor(FocusWidget newCursor) {
    cursor = newCursor;
}
```

Teraz, gdy mamy już zastępczą metodę, możemy podjąć się próby utworzenia instancji `FocusWidget` poza klasą i przekazać go do obiektu po jego skonstruowaniu. Ponieważ potrzebujemy przeprowadzić rozpoznanie, w odniesieniu do klasy `FocusWidget` możemy skorzystać z techniki **wyodrębniania interfejsu** (361) albo **wyodrębniania implementera**

(356) i utworzyć fałszywy obiekt do przekazania. Z pewnością będzie to łatwiejsze niż tworzenie obiektu `FocusWidget` w konstruktorze.

```
TEST(renderBorder, WatercolorPane)
{
    ...
    TestingFocusWidget *widget = new TestingFocusWidget;
    WatercolorPane pane(form, border, backdrop);

    pane.supersedeCursor(widget);

    LONGS_EQUAL(0, pane.getComponentCount());
}
```

Nie lubię korzystać z techniki **zastępowania zmiennej instancji** (401). Używam jej, kiedy nie mam już innego wyjścia. Prawdopodobieństwo pojawienia się problemów z zarządzaniem zasobami jest zbyt duże. Mimo to korzystam z niej od czasu do czasu w C++. Często wolałbym zastosować **wyodrębnianie i przesłanianie metody wytwórczej** (351), co jednak jest niemożliwe w przypadku konstruktorów w języku C++. Z tego też powodu rzadko uciekam się do **zastępowania zmiennej instancji** (401).

Przypadek irytującej zależności globalnej

Od lat ludzie w branży programistycznej narzekają, że nie ma na rynku większej liczby komponentów wielokrotnego użytku. Wraz z upływem czasu sytuacja polepszyła się — istnieje wiele komercyjnych oraz otwartych platform, ale w zasadzie z wielu z nich tak naprawdę nie korzystamy; to raczej one używają naszego kodu. Platformy te często zarządzają cyklem życia aplikacji, a my piszemy kod wypełniający puste miejsca. Możemy to zaobserwować we wszystkich rodzajach platform, od ASP.NET aż do Java Struts. W taki sposób działa nawet xUnit — piszemy klasy testowe, a on je wywołuje i wyświetla wyniki ich działania.

Platformy rozwiązują wiele problemów i nadają nam impet, gdy rozpoczynamy nowe projekty, ale to nie takiego rodzaju wielokrotnego wykorzystania oczekiwano we wczesnych latach rozwoju oprogramowania. Wielokrotne użycie w starym stylu ma miejsce wtedy, gdy znajdujemy jakąś klasę lub zbiór klas, których chcemy użyć w naszej aplikacji, i po prostu to robimy. Dodajemy je do naszego projektu i z nich korzystamy. Byłoby miło, gdybyśmy mogli tak robić rutynowo, ale szczerze mówiąc — myślę, że sami siebie oszukujemy nawet wtedy, gdy tylko myślimy o takim rodzaju wielokrotnego użycia, skoro nie potrafimy wyciągnąć z pierwszej lepszej aplikacji dowolnej klasy i oddzielnie jej skompilować w jarzmie testowym bez konieczności wykonania całej masy pracy (ale zrzędzę).

Wiele różnych rodzajów zależności może utrudniać tworzenie i używanie klas na platformach testowych, a jedną z najgorszych, z jakimi możemy mieć do czynienia, jest użycie zmiennych globalnych. W prostszych przypadkach w celu ominięcia takich zależności możemy posłużyć się **parametryzacją konstruktora** (377), **parametryzacją metody**

(381) oraz **wyodrębnianiem i przesłanianiem wywołania** (349), ale czasami zależności globalne są tak wszechobecne, że prościej będzie rozprawić się z nimi u źródła. Z taką sytuacją spotkamy się w następnym przykładzie, którym jest aplikacja w Javie, rejestrująca pozwolenia na budowę w pewnej agencji rządowej. Oto jedna z jej głównych klas:

```
public class Facility
{
private Permit basePermit;

    public Facility(int facilityCode, String owner, PermitNotice notice)
              throws PermitViolation {

    Permit associatedPermit =
        PermitRepository.getInstance().findAssociatedPermit(notice);

    if (associatedPermit.isValid() && !notice.isValid()) {
        basePermit = associatedPermit;
    }
    else if (!notice.isValid()) {
        Permit permit = new Permit(notice);
        permit.validate();
        basePermit = permit;
    }
    else
        throw new PermitViolation(permit);
    }
   ...
}
```

Chcielibyśmy utworzyć klasę `Facility` w jarzmie testowym, w związku z czym zaczynamy od próby utworzenia jej obiektu:

```
public void testCreate() {
    PermitNotice notice = new PermitNotice(0, "a");
    Facility facility = new Facility(Facility.RESIDENCE, "b", notice);
}
```

Test kompiluje się prawidłowo, ale kiedy zaczynamy pisać kolejne testy, zauważamy pewien problem. Konstruktor korzysta z klasy o nazwie `PermitRepository` i aby nasze testy zostały poprawnie skonfigurowane, musi zostać zainicjalizowany określonym zestawem pozwoleń. Przebiegłe, co? Oto ta problematyczna instrukcja w konstruktorze:

```
Permit associatedPermit =
           PermitRepository.getInstance().findAssociatedPermit(notice);
```

Moglibyśmy ominąć tę przeszkodę, parametryzując konstruktor, ale w tej aplikacji nie jest to odosobniony przypadek. Istnieje jeszcze 10 innych klas, które zawierają mniej więcej taki sam wiersz kodu. Tkwi on w konstruktorach, metodach zwykłych i statycznych. Możemy tylko wyobrazić sobie, ile czasu byśmy poświęcili, walcząc z tym problemem w bazie kodu.

Jeżeli uczyłeś się kiedyś o wzorcach projektowych, być może rozpoznałeś w tym przykładzie **wzorzec projektowy singleton** (370). Metoda `getInstance` klasy `PermitRepository`

jest metodą statyczną, której zadaniem jest zwrócenie jedynej instancji klasy `PermitRepository`, która może istnieć w naszej aplikacji. Pole przechowujące tę instancję także jest statyczne i znajduje się w tej klasie.

W Javie wzorzec singleton jest jednym z mechanizmów używanych do tworzenia zmiennych globalnych. Zazwyczaj wykorzystywanie zmiennych globalnych jest złym pomysłem z kilku powodów. Jednym z nich jest nieprzejrzystość. Kiedy oglądamy fragment kodu, dobrze byłoby wiedzieć, na co może on wpływać. Na przykład w Javie — jeśli chcemy zrozumieć, jaki wpływ wywiera na różne elementy poniższy kod — musimy zajrzeć do kilku zaledwie miejsc.

```
Account example = new Account();
example.deposit(1);
int balance = example.getBalance();
```

Wiemy, że obiekt klasy `Account` może mieć wpływ na elementy, które przekazujemy do konstruktora `Account`, ale my niczego nie przekazujemy. Obiekty klasy `Account` mogą też wpływać na obiekty, które przekazujemy do metody jako parametry, ale w tym przypadku nie przekazujemy nic, co mogłoby ulec zmianie, a jedynie liczbę całkowitą. W miejscu tym przypisujemy zwracaną wartość `getBalance` zmiennej i tak naprawdę jest to jedyny element, który może być zmieniany przez powyższe instrukcje.

Kiedy korzystamy ze zmiennych globalnych, sytuacja zostaje postawiona na głowie. Możemy patrzeć na użycie takich klas jak `Account` i nie mieć pojęcia, czy ma ona dostęp i modyfikuje zmienne zadeklarowane w jakimś innym miejscu programu. Nie trzeba nadmieniać, że z tego powodu programy mogą być trudniejsze do zrozumienia.

Trudnym zadaniem w naszej sytuacji jest konieczność określenia, które zmienne globalne zostały użyte w klasie, i nadanie im odpowiednich wartości na potrzeby testu. Do tego musimy to robić przed każdym testem, jeśli konfiguracja testów ma się zmieniać. Zadanie to jest dość żmudne; musiałem je wykonywać w odniesieniu do całej masy systemów, aby można było je poddać testom, i wraz z upływem czasu nie stało się ono nawet na jotę bardziej ekscytujące.

Powróćmy jednak do naszego przykładu.

`PermitRepository` jest singletonem. Z tego względu jest on szczególnie trudny do sfałszowania. Cała koncepcja kryjąca się za wzorcem singletona polega na uniemożliwieniu tworzenia więcej niż jednej jego instancji w danej aplikacji. Takie rozwiązanie może sprawdzać się w kodzie produkcyjnym, ale w przypadku testowania każdy test w zestawie powinien być w pewnym sensie miniaplikacją — powinien być całkowicie odizolowany od pozostałych testów. Aby zatem uruchomić w jarzmie testowym kod zawierający singletony, musimy osłabić własność singletona. Oto jak to zrobimy.

Najpierw do klasy singletona dodamy nową metodę statyczną. Pozwoli nam ona zastąpić statyczną instancję w singletonie. Nazwiemy ją `setTestingInstance`.

```
public class PermitRepository
{
    private static PermitRepository instance = null;
```

```
    private PermitRepository() {}

    public static void setTestingInstance(PermitRepository newInstance)
    {
        instance = newInstance;
    }

    public static PermitRepository getInstance()
    {
        if (instance == null) {
            instance = new PermitRepository();
        }
        return instance;
    }
    public Permit findAssociatedPermit(PermitNotice notice) {
        ...
    }
    ...
}
```

Teraz, gdy mamy już setter, możemy utworzyć testową instancję klasy `PermitRepository` i nadać jej wartość. W naszej testowej metodzie `setUp` chcielibyśmy dodać następujący kod:

```
public void setUp() {
    PermitRepository repository = new PermitRepository();
    ...
    // w tym miejscu dodaj zezwolenia do repozytorium
    ...
    PermitRepository.setTestingInstance(repository);
}
```

Wprowadzanie statycznego settera (370) nie jest jedynym sposobem na poradzenie sobie z taką sytuacją. Oto inne podejście. Do metody `resetForTesting()` możemy dodać singleton, który wygląda następująco:

```
public class PermitRepository
{
    ...
    public void resetForTesting() {
        instance = null;
    }
    ...
}
```

Jeśli wywołamy tę metodę z naszej metody testowej `setUp` (a dobrym pomysłem będzie wywołanie jej także z metody `tearDown`), będziemy tworzyć świeże singletony w każdym teście. Za każdym razem singleton będzie inicjalizował się od nowa. Schemat ten dobrze się sprawdza, kiedy metody publiczne w singletonie pozwalają konfigurować jego stan w dowolny sposób, jaki tylko będzie potrzebny podczas testowania. Jeśli singleton nie ma takich publicznych metod albo korzysta z zewnętrznych zasobów, które wpływają na jego stan, lepszym wyborem będzie **wprowadzenie statycznego settera** (370). Dzięki temu będziesz mógł tworzyć podklasy singletona, przesłaniać metody, usuwać zależności i dodawać metody publiczne do podklas, aby poprawnie konfigurować ich stan.

Czy to zadziała? Jeszcze nie. Kiedy programiści korzystają ze **wzorca projektowego singleton** (370), często ich konstruktor klasy singletona jest prywatny, i mają ku temu dobry powód. Jest to najbardziej przejrzysty sposób zagwarantowania, że nikt spoza tej klasy nie będzie mógł utworzyć kolejnej instancji singletona.

W tym momencie pojawia się konflikt między dwoma założeniami naszego projektu. Chcemy mieć pewność, że w systemie istnieje tylko jedna instancja klasy `PermitRepository`, ale chcemy też dysponować systemem, w którym klasy można testować niezależnie od siebie. Czy uda nam się osiągnąć jednocześnie oba te cele?

Cofnijmy się na chwilę. Dlaczego chcemy mieć w systemie tylko jedną instancję klasy? Odpowiedź będzie się różnić w zależności od systemu, ale oto kilka najczęściej spotykanych powodów:

1. **Modelujemy rzeczywisty świat, a w rzeczywistym świecie istnieje tylko jedna taka rzecz.** Właśnie takie są niektóre systemy kontrolujące sprzęt. Programiści tworzą klasę dla każdego urządzenia, które musi być kontrolowane. Wychodzą z założenia, że jeśli istnieje tylko po jednym urządzeniu, każde z nich powinno być singletonem. Podobnie sprawy się mają w przypadku baz danych. W naszej agencji istnieje tylko jeden zbiór pozwoleń, a zatem element zapewniający do nich dostęp powinien być singletonem.
2. **Jeśli utworzymy dwie takie rzeczy, możemy znaleźć się w poważnych opałach.** Sytuacja taka również często ma miejsce w dziedzinie sterowania urządzeniami. Wyobraź sobie przypadkowe utworzenie dwóch kontrolerów prętów uranowych i umożliwienie dwóm różnym częściom programu sterowanie nimi w tym samym czasie bez wzajemnej wiedzy o sobie.
3. **Jeśli ktoś utworzy dwie takie rzeczy, będziemy zużywać zbyt wiele zasobów.** To zdarza się często. Zasoby mogą być obiektami fizycznymi, takimi jak miejsce na dysku albo zużycie pamięci, ale mogą być także abstrakcyjne, jak na przykład liczba licencji na oprogramowanie.

Takie są powody, dla których wymusza się istnienie pojedynczych instancji, ale nie są to główne powody, dla których singletony są używane. Programiści często tworzą singletony, ponieważ chcą mieć zmienne globalne. Uważają, że przekazywanie zmiennych do miejsc, w których będą one potrzebne, jest zbyt kłopotliwe.

Jeśli mamy singletona z tej drugiej przyczyny, to naprawdę nie ma powodu do zachowania jego własności. Nasz konstruktor może mieć zakres chroniony, publiczny albo pakietu, a przy tym nadal będziemy dysponować przyzwoitym, możliwym do testowania systemem. W innym przypadku i tak warto poszukać alternatywy. Jeśli zajdzie taka potrzeba, wprowadzimy inny rodzaj ochrony. Moglibyśmy dodać w naszym systemie kompilującym sprawdzanie we wszystkich plikach źródłowych, czy metoda `setTestingInstance` nie jest wywoływana przez kod nietestujący. Tak samo możemy postąpić w odniesieniu do kontroli wykonywanej w czasie działania programu. Jeśli metoda `setTestingInstance` zostanie wywołana podczas działania aplikacji, możemy podnieść alarm albo zawiesić system i poczekać na działanie ze strony użytkownika. Faktem jest, że chociaż wymuszenie

„singletonowości" nie było możliwe w wielu językach przed pojawieniem się zorientowania obiektowego, to jednak programiści zdołali utworzyć wiele bezpiecznych systemów. W końcu wszystko sprowadza się do odpowiedzialnego projektu i kodowania.

Jeżeli naruszenie własności singletona nie stanowi większego problemu, możemy zdać się na regułę stosowaną przez zespół. Przykładowo każdy w zespole powinien zrozumieć, że w aplikacji mamy jedną instancję bazy danych i że nie powinniśmy mieć kolejnej.

Aby osłabić właściwość singletona w klasie `PermitRepository`, możemy przekształcić konstruktor w publiczny. Takie rozwiązanie będzie nas satysfakcjonować, dopóki publiczne metody tej klasy pozwalają nam robić wszystko, czego potrzebujemy w celu skonfigurowania naszego repozytorium na potrzeby testów. Jeśli na przykład w klasie `PermitRepository` znajduje się metoda o nazwie `addPermit`, umożliwiająca dodawanie pozwoleń, które będą nam potrzebne w testach, być może wystarczy po prostu, że umożliwimy sobie tworzenie repozytoriów i użyjemy ich w naszych testach. W innym przypadku możemy nie mieć potrzebnego nam dostępu lub — co gorsza — singleton może robić rzeczy, co do których nie chcielibyśmy, aby się działy w jarzmie testowym, takie jak komunikowanie się w tle z bazą danych. W takich okolicznościach możemy **utworzyć podklasę i przesłonić metodę** (398), po czym utworzyć klasy pochodne, które ułatwią nam testowanie.

Oto przykład z naszego systemu pozwoleń. Oprócz metod i zmiennych, które sprawiają, że `PermitRepository` jest singletonem, mamy także następującą metodę:

```
public class PermitRepository
{
    ...
    public Permit findAssociatedPermit(PermitNotice notice) {
        // otwórz bazę danych z pozwoleniami
        ...

        // wybierz na podstawie wartości w powiadomieniu
        ...

        // sprawdź, czy mamy tylko jedno pasujące pozwolenie; jeśli nie, zgłoś błąd
        ...

        // zwróć pasujące pozwolenie
        ...
    }
}
```

Jeśli chcemy uniknąć komunikacji z bazą danych, możemy utworzyć następującą podklasę klasy `PermitRepository`:

```
public class TestingPermitRepository extends PermitRepository
{
    private Map permits = new HashMap();

    public void addAssociatedPermit(PermitNotice notice, permit) {
        permits.put(notice, permit);
    }

    public Permit findAssociatedPermit(PermitNotice notice) {
```

```
        return (Permit)permits.get(notice);
    }
}
```

Kiedy tak zrobimy, będziemy mogli zachować część własności singletona. Ponieważ korzystamy z podklasy klasy `PermitRepository`, sprawimy, że będzie chroniony raczej nasz konstruktor klasy `PermitRepository` niż publiczny. Dzięki temu zabezpieczymy się przed utworzeniem więcej niż jednej instancji tej klasy, chociaż będziemy mogli tworzyć jej podklasy.

```
public class PermitRepository
{
    private static PermitRepository instance = null;

    protected PermitRepository() {}

    public static void setTestingInstance(PermitRepository newInstance)
   {
        instance = newInstance;
    }

    public static PermitRepository getInstance()
    {
        if (instance == null) {
            instance = new PermitRepository();
        }
        return instance;
    }
    public Permit findAssociatedPermit(PermitNotice notice)

    {
        ...
    }
    ...
}
```

W wielu przypadkach możemy skorzystać z **tworzenia podklasy i przesłaniania metody** (398) — takiego jak powyżej — aby wstawić na miejsce fałszywy singleton. Innym razem zależności będą do tego stopnia rozbudowane, że łatwiej będzie **wyodrębnić interfejs** (361) względem singletona i zmienić wszystkie referencje w aplikacji, aby używały nazwy interfejsu. Może to kosztować sporo pracy, ale w celu dokonania takich zmian moglibyśmy skorzystać ze **wsparcia kompilatora** (317). Po wyodrębnieniu klasa `PermitRepository` będzie wyglądać następująco:

```
public class PermitRepository implements IPermitRepository
{
    private static IPermitRepository instance = null;

    protected PermitRepository() {}

    public static void setTestingInstance(IPermitRepository newInstance)
    {
        instance = newInstance;
```

```
    }

    public static IPermitRepository getInstance()
    {
        if (instance == null) {
            instance = new PermitRepository();
        }
        return instance;
    }

    public Permit findAssociatedPermit(PermitNotice notice)
    {
        ...
    }
    ...
}
```

Interfejs `IPermitRepository` będzie mieć sygnatury wszystkich publicznych, niestatycznych metod klasy `PermitRepository`.

```
public interface IPermitRepository
{
    Permit findAssociatedPermit(PermitNotice notice);
    ...
}
```

Jeśli używasz języka, który jest wyposażony w narzędzie do refaktoryzacji, mógłbyś wykonać takie wyodrębnienie interfejsu automatycznie. Jeśli Twój język nie ma tej możliwości, łatwiej będzie skorzystać z techniki **wyodrębniania implementera** (356).

Cały ten proces refaktoryzacji nosi nazwę **wprowadzania statycznego settera** (370). Jest to technika, z której możemy skorzystać, aby rozmieścić testy mimo istnienia rozległych zależności globalnych. Niestety, niespecjalnie nadaje się ona do obchodzenia globalnych zależności. Jeśli musisz poradzić sobie z tym problemem, posłuż się **parametryzacją metody** (381) i **parametryzacją konstruktora** (377). Za pomocą tych technik refaktoryzacji zmienisz referencję globalną na zmienną tymczasową w metodzie lub na pole w obiekcie. Wada **parametryzacji metody** (381) polega na tym, że w wyniku jej zastosowania możesz uzyskać wiele dodatkowych metod, które będą rozpraszać osoby próbujące zrozumieć klasy. Z kolei wada **parametryzacji konstruktora** (377) jest taka, że każdy obiekt korzystający ze zmiennej globalnej otrzymuje w rezultacie dodatkowe pole. Pole to będzie musiało zostać przekazane do konstruktora, przez co klasa tworząca obiekt także będzie musiała mieć dostęp do instancji. Jeżeli to dodatkowe pole będzie potrzebne w zbyt wielu obiektach, może to znacząco wpłynąć na ilość pamięci używanej przez aplikację, chociaż często wskazuje to na inne problemy w projekcie.

Przyjrzyjmy się najgorszemu przypadkowi. Mamy aplikację z kilkoma setkami klas, które podczas działania programu tworzą tysiące obiektów, a każdy z tych obiektów wymaga dostępu do bazy danych. Pierwsze pytanie, które przychodzi mi na myśl, nawet bez spojrzenia na aplikację, brzmi: dlaczego? Jeśli system robi jeszcze coś innego oprócz komunikowania się z bazą danych, można przeprowadzić jego refaktoryzację, dzięki czemu

część klas zajmie się tymi innymi rzeczami, a pozostałe klasy będą zapisywać i pobierać dane z bazy. Kiedy podejmujemy skoordynowane działania w celu rozdzielenia odpowiedzialności w aplikacji, zależności stają się lokalne — referencja do bazy danych w każdym obiekcie nie będzie potrzebna. Niektóre obiekty będą zapełniane danymi pochodzącymi z bazy, a inne będą przeprowadzać obliczenia na danych, które otrzymały za pośrednictwem swoich konstruktorów.

W ramach ćwiczeń wybierz sobie w dużej aplikacji zmienną globalną i poszukaj jej. W większości przypadków zmienne globalne są globalnie dostępne, ale rzadko są globalnie używane. Korzysta się z nich w relatywnie niewielu miejscach. Wyobraź sobie, jak moglibyśmy przekazać taki obiekt do obiektów, które go potrzebują, gdyby nie mógł on być zmienną globalną. W jaki sposób dokonalibyśmy refaktoryzacji tego programu? Czy istnieją odpowiedzialności, które moglibyśmy wydzielić ze zbiorów klas, aby ograniczyć ich globalny zasięg?

Jeśli znajdziesz globalną zmienną, która rzeczywiście jest używana we wszystkich miejscach, oznacza to, że w Twoim kodzie nie ma żadnego podziału na warstwy. Zajrzyj do rozdziałów 15., „Cała moja aplikacja to wywołania API”, i 17., „Moja aplikacja nie ma struktury”.

Przypadek straszliwych zależności dyrektyw include

C++ był moim pierwszym językiem zorientowanym obiektowo i muszę przyznać, że czułem się bardzo dumny, kiedy nauczyłem się wielu jego szczegółów i zawiłości. Zdominował on branżę, gdyż w swoim czasie stanowił niezwykle praktyczne rozwiązanie wielu dokuczliwych problemów. Komputery są zbyt wolne? Proszę bardzo, oto język, w którym wszystko jest opcjonalne. Możesz mieć całą wydajność czystego C, jeśli będziesz używać tylko jego funkcji. Nie możesz namówić swoich ludzi do korzystania z języka zorientowanego obiektowo? Proszę bardzo, oto kompilator C++; w kodzie C możesz dopisać fragment w C++ i uczyć się zorientowania obiektowego w trakcie programowania.

Chociaż C++ był przez pewien czas bardzo popularny, w końcu ustąpił miejsca na rzecz Javy oraz paru innych, nowszych języków. W pewnym stopniu przyczyną była konieczność zachowania wstecznej zgodności z C, ale o wiele większy wpływ wywarł wymóg uproszczenia pracy z językami programowania. Zespoły pracujące w C++ regularnie przekonywały się, że domyślna konfiguracja tego języka nieszczególnie sprawdza się podczas konserwacji i że muszą poza nią wykraczać, aby system był elastyczny i podatny na wprowadzanie zmian.

Jeden z aspektów C++, wywodzący się z C, który jest szczególnie kłopotliwy, to sposób, w jaki jedna część programu dowiaduje się o innej części. W Javie i C#, gdy klasa w jednym pliku musi skorzystać z klasy w drugim pliku, stosujemy import lub posługujemy się dyrektywą `using`, aby definicja klasy stała się dostępna. Kompilator szuka tej klasy i sprawdza, czy była już ona kompilowana. Jeśli nie, kompiluje ją. Jeżeli klasa była już kompilowana,

kompilator odczytuje ze skompilowanego pliku niewielki fragment, pobierając tylko tyle informacji, ile jest potrzebnych do zagwarantowania, że wszystkie metody wymagane przez nową klasę znajdą się na miejscu.

Kompilatory C++ zazwyczaj nie stosują tego typu optymalizacji. Jeśli w C++ klasa musi coś wiedzieć o innej klasie, deklaracja tej drugiej klasy (w innym pliku) jest dołączana w formie tekstowej do pliku, który potrzebuje tych informacji. Taki proces może być powolny. Kompilator musi powtórnie przeanalizować deklarację i zbudować jej wewnętrzną reprezentację za każdym razem, kiedy ją napotyka. Co gorsza, mechanizm dołączania jest podatny na nadużycia. Plik może dołączać plik, który dołącza kolejny plik itd. W przypadku projektów, przy których programiści nie unikali takiego rozwiązania, nietrudno o znalezienie małych plików, które koniec końców dołączają tysiące wierszy kodu. Programiści zastanawiają się, dlaczego kompilacja trwa tak długo, ale ponieważ instrukcje dołączające są rozsiane po całym systemie, trudno jest wskazać konkretny plik i zrozumieć, dlaczego zajmuje to tyle czasu.

Można odnieść wrażenie, że czepiam się C++, ale tak nie jest. To ważny język i stworzono w nim niewiarygodnie dużo kodu, ale poprawna z nim praca wymaga szczególnej staranności.

Uzyskanie instancji klasy C++ w jarzmie testowym może być trudne w przypadku cudzego kodu. Jeden z problemów, które prawie od razu napotykamy, to zależności nagłówkowe. Które pliki nagłówkowe są nam potrzebne, aby w jarzmie testowym utworzyć klasę?

Oto część deklaracji obszernej klasy C++ o nazwie `Scheduler`. Zawiera ona ponad 200 metod, ale tutaj pokazałem jakieś 5 z nich. Nie dość, że klasa ta jest ogromna, to jeszcze charakteryzuje się silnymi i złożonymi zależnościami od wielu innych klas. Jak moglibyśmy poddać klasę `Scheduler` testom?

```
#ifndef SCHEDULER_H
#define SCHEDULER_H

#include "Meeting.h"
#include "MailDaemon.h"
...
#include "SchedulerDisplay.h"
#include "DayTime.h"

class Scheduler
{
public:
        Scheduler(const string& owner);
        ~Scheduler();

    void addEvent(Event *event);
    bool hasEvents(Date date);
    bool performConsistencyCheck(string& message);
    ...
};
#endif
```

Poza innymi elementami klasa `Scheduler` korzysta też z plików *Meeting*, *MailDemon*, *Event*, *SchedulerDisplay* i *DayTime*. Gdy chcemy utworzyć testy dla obiektów klasy `Scheduler`, najprostsze, co możemy zrobić, to próba utworzenia ich w tym samym katalogu, w nowym pliku o nazwie *SchedulerTests*. Dlaczego chcemy mieć testy w tym samym katalogu? Przy obecności preprocesora tak będzie łatwiej. Jeżeli w projekcie nie użyto ścieżek, umożliwiających dołączanie plików w spójny sposób, czekałoby nas sporo pracy, gdybyśmy chcieli umieścić nasze testy w innych katalogach.

```
#include "TestHarness.h"
#include "Scheduler.h"

TEST(create,Scheduler)
{
    Scheduler scheduler("fred");
}
```

Jeśli utworzymy plik i po prostu wpiszemy taką deklarację obiektu do testu, natkniemy się na problem z dołączaniem plików nagłówkowych. Aby skompilować klasę `Scheduler`, musimy mieć pewność, że kompilator i konsolidator wiedzą wszystko na temat elementów, które są tej klasie potrzebne, a także wszystko na temat elementów potrzebnych tym elementom itd. Na szczęście system przekazuje nam sporą liczbę komunikatów o błędach i szczegółowo nas o nich informuje.

W prostszych przypadkach plik *Scheduler.h* zawiera wszystko, co jest nam potrzebne do utworzenia klasy `Scheduler`, ale w niektórych przypadkach plik nagłówkowy nie obejmuje wszystkiego. Aby utworzyć obiekt i z niego korzystać, będziemy musieli dołączyć dodatkowe pliki.

Moglibyśmy po prostu skopiować wszystkie dyrektywy `#include` z pliku źródłowego z klasą `Scheduler`, ale być może nie będziemy potrzebować każdego z tych plików. Najlepszą taktyką byłoby dodawanie ich po jednym i podjęcie decyzji, czy ta konkretna zależność jest nam tak naprawdę niezbędna.

W idealnym świecie najprostsze byłoby dołączanie po kolei wszystkich potrzebnych nam plików, aż przestałyby się pojawiać błędy kompilacji, lecz takie postępowanie mogłoby doprowadzić do zamętu w naszym kodzie. Jeśli istnieje długi ciąg przechodnich zależności, zapewne w rezultacie dołączylibyśmy o wiele więcej, niż naprawdę potrzebujemy. Nawet jeśli ciąg zależności nie jest zbyt długi, mogłoby się okazać, że zależymy od elementów, z którymi praca w jarzmie testowym jest bardzo trudna. W naszym przykładzie klasa `SchedulerDisplay` jest jedną z takich zależności. Nie pokazuję tego tutaj, ale sięga do niej konstruktor w klasie `Scheduler`. Tego rodzaju zależności możemy się pozbyć następująco:

```
#include "TestHarness.h"
#include "Scheduler.h"

void SchedulerDisplay::displayEntry(const string& entyDescription)
{
}

TEST(create,Scheduler)
```

```
{
    Scheduler scheduler("fred");
}
```

Wprowadziliśmy w tym miejscu alternatywną definicję `SchedulerDisplay::display ↪Entry`. Niestety, kiedy tak zrobimy, będziemy potrzebować odrębnej kompilacji przypadków testowych zawartych w tym pliku. W programie możemy mieć tylko po jednej definicji każdej metody w klasie `SchedulerDisplay`, w związku z czym potrzebny nam będzie oddzielny program dla naszych testów tej klasy.

Na szczęście w pewnym stopniu będziemy mogli wielokrotnie korzystać z fałszywek, które utworzyliśmy w ten sposób. Zamiast umieszczać definicje klas — takich jak `SchedulerDisplay` — bezpośrednio w pliku testowym, możemy zamieścić je w oddzielnym pliku, z którego będzie można skorzystać w plikach z testami:

```
#include "TestHarness.h"
#include "Scheduler.h"
#include "Fakes.h"

TEST(create,Scheduler)
{
    Scheduler scheduler("fred");
}
```

Po kilkakrotnym wykonaniu takiego zabiegu tworzenie instancji klasy C++ w jarzmie testowym stanie się całkiem łatwe i machinalne. Istnieje jednak kilka dość poważnych wad tego rozwiązania. Musimy utworzyć odrębny program i tak naprawdę nie usuwamy żadnych zależności na poziomie języka, w związku z czym kod nie staje się przejrzystszy, gdy się ich pozbywamy. Co gorsza, powielone definicje, które umieszczamy w pliku testowym (w naszym przykładzie `SchedulerDisplay::displayEntry`), muszą być utrzymywane tak długo, jak długo zachowujemy dany zestaw testów na swoim miejscu.

Technikę tę zachowuję dla przypadków, w których mam do czynienia z bardzo dużą klasą, wykazującą poważne problemy z zależnościami. Nie jest to technika, z której można korzystać często lub w prosty sposób. Jeśli dana klasa ma zostać rozbita wraz z upływem czasu na dużą liczbę mniejszych klas, korzystne może okazać się utworzenie dla niej odrębnego programu testowego. Może on odgrywać rolę poligonu doświadczalnego, służącego do rozległej refaktoryzacji. Wraz z upływem czasu, gdy coraz więcej klas zostanie poddanych testom, będzie można pozbyć się tego programu.

Przypadek cebulowego parametru

Lubię proste konstruktory. Naprawdę. To wspaniałe, kiedy decydujesz się na utworzenie klasy, po czym po prostu wpisujesz wywołanie konstruktora i otrzymujesz sympatyczny, żywy, działający i gotowy do użycia obiekt. W wielu przypadkach tworzenie obiektów może być jednak trudne. Każdy obiekt powinien zostać skonfigurowany we właściwym stanie — stanie, który przygotuje go do dodatkowych zadań. W wielu przypadkach ozna-

cza to, że musimy mu udostępnić inne obiekty, które także muszą być poprawnie skonfigurowane. Obiekty te podczas konfiguracji mogą wymagać jeszcze innych obiektów i w rezultacie dochodzimy do tworzenia obiektów potrzebnych do utworzenia obiektów potrzebnych do utworzenia obiektów potrzebnych do utworzenia parametru dla konstruktora klasy, którą chcemy poddać testom. Obiekty wewnątrz innych obiektów — wygląda to jak jakaś wielka cebula. Oto przykład tego rodzaju problemu.

Mamy klasę wyświetlającą obiekt typu `SchedulingTask`:

```
public class SchedulingTaskPane extends SchedulerPane
{
    public SchedulingTaskPane(SchedulingTask task) {
        ...
    }
}
```

Aby ją utworzyć, musimy przekazać jej obiekt `SchedulingTask`, ale w celu jego utworzenia potrzebujemy skorzystać z jego jedynego konstruktora:

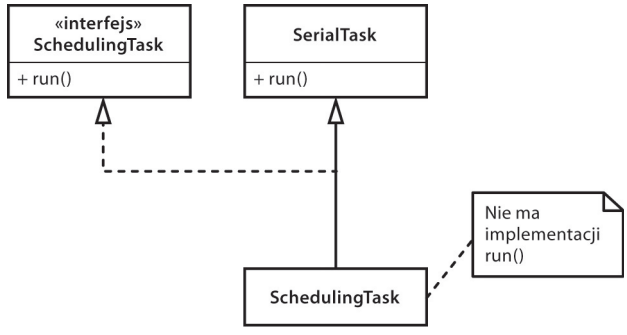
```
public class SchedulingTask extends SerialTask
{
    public SchedulingTask(Scheduler scheduler, MeetingResolver resolver)
    {
        ...
    }
}
```

Jeśli odkryjemy, że do utworzenia obiektów `Scheduler` i `MeetingResolver` potrzebujemy kolejnych obiektów, prawdopodobnie zaczniemy rwać sobie włosy z głowy. Jedyne, co nas powstrzymuje od pogrążenia się w skrajnej rozpaczy, to fakt, że musi istnieć przynajmniej jedna klasa, która nie potrzebuje jako argumentów obiektów innej klasy. W przeciwnym razie system nigdy nie dałby się skompilować.

Sposób na poradzenie sobie z taką sytuacją polega na bliższym zastanowieniu się nad tym, co chcemy osiągnąć. Musimy napisać testy, ale czego tak naprawdę potrzebujemy od parametrów przekazywanych do konstruktora? Jeśli na potrzeby naszych testów nie potrzebujemy niczego, to możemy **przekazać wartość pustą** (126). Jeżeli potrzebujemy tylko pewnego, elementarnego zachowania, możemy z najbliższej zależności **wyodrębnić interfejs** (361) albo **wyodrębnić implementer** (356) i skorzystać z otrzymanego interfejsu w celu utworzenia fałszywego obiektu. W naszym przypadku najbliższą zależnością klasy `SchedulingTaskPane` jest `SchedulingTask`. Jeśli uda nam się utworzyć fałszywy obiekt klasy `SchedulingTask`, będziemy w stanie utworzyć instancję klasy `SchedulingTaskPane`.

Niestety, klasa `SchedulingTask` dziedziczy po klasie `SerialTask`, a jedyne, co robi, to przesłonięcie kilku metod chronionych; wszystkie metody publiczne znajdują się w klasie `SerialTask`. Czy w odniesieniu do klasy `SchedulingTask` możemy skorzystać z **wyodrębniania interfejsu** (361)? A może powinniśmy zastosować tę technikę także do klasy `SerialTask`? W Javie nie musimy tego robić. Możemy utworzyć interfejs dla klasy `SchedulingTask`, który zawiera również metody klasy `SerialTask`.

Nasza wynikowa hierarchia wygląda jak na rysunku 9.3.



Rysunek 9.3. *Interfejs SchedulingTask*

W tym przypadku mamy szczęście, że korzystamy z Javy. Niestety, w C++ nie mamy możliwości obsługiwania takich przypadków, gdyż w języku tym nie istnieją samodzielne interfejsy. Są one zwykle implementowane w klasach zawierających jedynie funkcje czysto wirtualne. Gdyby przykład ten został przełożony na C++, interfejs `SchedulingTask` stałby się klasą abstrakcyjną, ponieważ dziedziczy funkcje wirtualne po klasie `SchedulingTask`. Aby utworzyć instancję klasy `SchedulingTask`, musielibyśmy udostępnić w niej ciało metody `run()`, które odsyłałoby do metody `run()` w klasie `SerialTask`. Na szczęście będzie to łatwe do wykonania. Oto jak teraz wygląda kod:

```
class SerialTask
{
public:
    virtual void run();
    ...
};

class ISchedulingTask
{
public:
    virtual void run() = 0;
    ...
};

class SchedulingTask : public SerialTask, public ISchedulingTask
{
public:
    virtual void run() { SerialTask::run(); }
};
```

W dowolnym języku, w którym możemy tworzyć interfejsy lub klasy działające jak interfejsy, możemy z nich systematycznie korzystać w celu usuwania zależności.

Przypadek zaliasowanego parametru

Często, kiedy do konstruktorów przekazywane są parametry, które wchodzą nam w drogę, możemy ominąć ten problem, stosując **wyodrębnianie interfejsu** (361) lub **wyodrębnianie implementera** (356). Czasami jednak takie rozwiązanie nie jest praktyczne. Spójrzmy na inną klasę z systemu pozwoleń na budowę, z którym mieliśmy do czynienia we wcześniejszym podrozdziale:

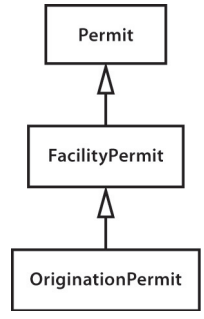
```
public class IndustrialFacility extends Facility
{
    Permit basePermit;

    public IndustrialFacility(int facilityCode, String owner,
                    OriginationPermit permit) throws PermitViolation {
        Permit associatedPermit =
            PermitRepository.GetInstance()
                            .findAssociatedFromOrigination(permit);

        if (associatedPermit.isValid() && !permit.isValid()) {
            basePermit = associatedPermit;
        }
        else if (!permit.isValid()) {
        permit.validate();
        basePermit = permit;
    }
    else
        throw new PermitViolation(permit);
    }
    ...
}
```

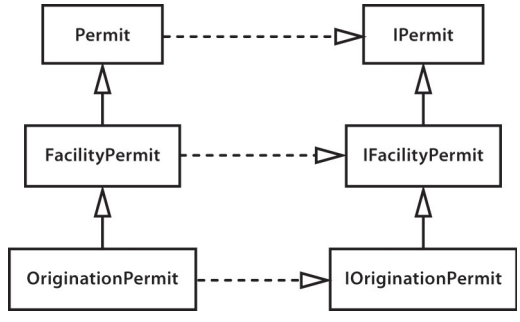
Chcielibyśmy utworzyć instancję tej klasy w jarzmie, ale na przeszkodzie stoi nam kilka problemów. Jeden z nich polega na tym, że znowu mamy do czynienia z singletonem — `PermitRepository`. Możemy ominąć ten problem, stosując techniki, które poznaliśmy we wcześniejszym podrozdziale „Przypadek irytującej zależności globalnej”. Zanim jednak rozwiążemy ten problem, napotykamy kolejny. Uzyskanie źródłowego pozwolenia, które musimy przekazać do konstruktora, jest trudne. Obiekty klasy `OriginationPermit` cechują się okropnymi zależnościami. Pierwsze, co przychodzi mi na myśl to: „Aha, żeby ominąć tę zależność, zastosuję wobec klasy `OriginationPermit` **wyodrębnianie interfejsu**”, ale nie jest to takie proste. Na rysunku 9.4 pokazano hierarchię obiektów klasy `Permit`.

Konstruktor `IndustrialFacility` przyjmuje obiekt klasy `OriginationPermit` i przechodzi do obiektu `PermitRepository`, aby zdobyć odpowiednie pozwolenie; w `PermitRepository` korzystamy z metody przyjmującej obiekt klasy `OriginationPermit` i zwracającej obiekt typu `Permit`. Jeśli repozytorium odnajdzie odpowiednie pozwolenie, zapisze je w polu `Permit`. Jeśli nie, zapisze w tym polu obiekt `OriginationPermit`. Moglibyśmy utworzyć interfejs dla klasy `OriginationPermit`, ale w niczym by nam to nie pomogło. Musielibyśmy



Rysunek 9.4. *Hierarchia obiektów klasy Permit*

polu `Permit` przypisać interfejs `IOriginationPermit`, co by nie zadziałało — w Javie interfejsy nie mogą dziedziczyć po klasach. Najbardziej oczywistym rozwiązaniem będzie utworzenie interfejsów od samej góry aż po dół i zamiana pola `Permit` na `IPermit`. Rysunek 9.5 pokazuje, jak to będzie wyglądać.



Rysunek 9.5. *Hierarchia obiektów klasy Permit z wyodrębnionymi interfejsami*

A fe! To absurdalnie dużo pracy i w ogóle nie podoba mi się kod, jaki byśmy otrzymali. Interfejsy świetnie nadają się do usuwania zależności, ale kiedy zbliżamy się do momentu, gdy między klasami a interfejsami mamy już niemal relację „jeden do jednego", projekt robi się zagracony. Nie zrozum mnie źle — gdy jesteśmy przyparci do muru, dobrze będzie zmierzać w stronę takiego projektu, ale jeśli istnieją inne możliwości, powinniśmy je rozpatrzyć. Na szczęście je mamy.

Wyodrębnianie interfejsu (361) jest tylko jednym ze sposobów na usuwanie zależności w odniesieniu do parametru. Czasami opłaca się zadać pytanie, dlaczego zależność jest niedobra. Czasem tworzenie obiektu jest uciążliwe. Niekiedy parametr wywołuje niepożądane efekty uboczne; może komunikuje się z systemem plików albo bazą danych. Innym razem wykonywanie się kodu może zabierać zbyt dużo czasu. Kiedy stosujemy **wyodrębnianie interfejsu** (361), możemy poradzić sobie z tymi wszystkimi problemami, chociaż robimy to, brutalnie odcinając jego połączenie z całą klasą. Jeśli problem kryje się tylko we fragmentach klasy, możemy przyjąć inne podejście i przeciąć połączenie tylko z kłopotliwymi fragmentami.

Przyjrzyjmy się bliżej klasie `OriginationPermit`. Nie chcemy jej używać w teście, ponieważ komunikuje się ona po kryjomu z bazą danych, kiedy każemy jej przeprowadzić autoryzację:

```
{
    ...
    public void validate() {
        // połącz się z bazą danych
        ...
        // pobierz informację o autoryzacji
        ...
        // ustaw flagę autoryzacji
        ...
        // zamknij bazę danych
        ...
    }
}
```

Nie chcemy tego robić w teście — musielibyśmy wprowadzić do bazy danych jakieś fałszywe wpisy, co zdenerwowałoby jej administratora. Gdyby się o tym dowiedział, przyszłoby nam postawić mu obiad, ale i tak byłby poirytowany. I bez tego jego praca jest wystarczająco trudna.

Inną strategią, którą moglibyśmy przyjąć, jest **utworzenie podklasy i przesłonięcie metody** (398). Możemy utworzyć klasę o nazwie `FakeOriginationPermit`, która udostępnia metodę ułatwiającą zmianę flagi uwierzytelnienia. Następnie w podklasach moglibyśmy przesłonić metodę `validate` i podczas testowania klasy `IndustrialFacility` przestawić flagę uwierzytelniania w taki sposób, jaki tylko będzie nam potrzebny. Oto całkiem dobry, pierwszy test:

```
public void testHasPermits() {
    class AlwaysValidPermit extends FakeOriginationPermit
    {
        public void validate() {
        // ustaw flagę uwierzytelnienia
        becomeValid();
     }
    };

    Facility facility = new IndustrialFacility(Facility.HT_1, "b",
                                               new AlwaysValidPermit());
    assertTrue(facility.hasPermits());
}
```

W wielu językach możemy tworzyć takie klasy „w locie” za pomocą metod. Chociaż nie lubię tego często robić w kodzie produkcyjnym, sposób ten jest całkiem wygodny podczas testowania. Bardzo prosto możemy tworzyć przypadki specjalne.

Tworzenie podklasy i przesłanianie metody (398) jest pomocne podczas usuwania zależności dotyczących parametrów, ale czasami faktoryzacja metod w klasie nie jest idealnym rozwiązaniem tego problemu. Mieliśmy szczęście, że zależności, które nam przeszkadzały, były odizolowane w metodzie `validate`. W najgorszym przypadku są

one splecione z potrzebną nam logiką, a my najpierw musimy wyodrębnić metody. Może to być proste, gdy mamy narzędzie refaktoryzujące. Jeśli nie dysponujemy takim narzędziem, przydatne mogą okazać się niektóre z technik opisanych w rozdziale 22., „Muszę zmienić monstrualną metodę, lecz nie mogę napisać do niej testów”.

Skorowidz

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Z

Ź

Programiści uwielbiają brać udział w nowych projektach, być świadkami ewolucji kodu, mieć wpływ na wybór narzędzi i projektować ich architekturę. Niestety, w ogromnej większości przypadków muszą pracować z kodem mającym sporo lat i pisanym przez wiele osób. Jak sobie poradzić w takim środowisku? Jak dobrać techniki pracy do gotowego kodu? Na te i inne podobne pytania odpowiada ten wyjątkowy podręcznik.

Dzięki niemu dowiesz się, jak wprowadzać zmiany w zastanym kodzie, tworzyć testy automatyczne oraz modyfikować architekturę rozwiązania. Ponadto poznasz najlepsze techniki pracy z projektami niezorientowanymi obiektowo oraz przekonasz się, że można skutecznie poradzić sobie z przerośniętymi klasami i metodami. Ostatnia część książki została poświęcona technikom usuwania zależności. Ten podręcznik to lektura obowiązkowa każdego programisty. Dzięki niemu Twoja praca z zastanym kodem nabierze nowego sensu!

DZIĘKI TEJ KSIĄŻCE:

- **poradzisz sobie z zastanym kodem**
- **nauczysz się wprowadzać w nim zmiany**
- **zastosujesz testy automatyczne**
- **przeprowadzisz skuteczną refaktoryzację**

PRACA Z ZASTANYM KODEM NIE MUSI BYĆ NUŻĄCA!

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-246-8317-8

Cena: 79,00 zł

Informatyka w najlepszym wydaniu