Technologia i rozwiązania

TDD
Programowanie w Javie sterowane testami

Naucz się podstaw metodyki TDD

Viktor Farcic
Alex Garcia

Helion

[PACKT] PUBLISHING open source community experience distilled

Tytuł oryginału: Test-Driven Java Development

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-2341-4

First published in the English language under the title
„Test-Driven Java Development — (9781783987429)".

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/tddpro.zip

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

Printed in Poland.

Spis treści

4

ROZDZIAŁ

Testy jednostkowe. Koncentrowanie się na wykonywanym zadaniu, a nie na tym, co już zostało zrobione

„Jeśli chcesz utworzyć coś wyjątkowego, twój umysł musi być nieustannie skoncentrowany na najdrobniejszych szczegółach”.

— Giorgio Armani

Zgodnie z wcześniejszą obietnicą w każdym rozdziale opisana zostanie inna platforma do testowania kodu napisanego w Javie. Ten rozdział nie jest pod tym względem wyjątkiem. Do budowania specyfikacji posłuży tu platforma TestNG.

W poprzednim rozdziale zastosowano procedurę czerwone, zielone, refaktoryzacja. Wykorzystano testy jednostkowe bez dokładnego wytłumaczenia, jak te testy działają w kontekście TDD. Tutaj na podstawie informacji z poprzedniego rozdziału szczegółowo wyjaśniono, czym są testy jednostkowe i jakie jest ich miejsce w procesie budowania oprogramowania za pomocą TDD.

Celem tego rozdziału jest pokazanie, jak skoncentrować się na obecnie rozwijanej jednostce i jak ignorować lub izolować wcześniej ukończone elementy.

Gdy już opanujesz wiedzę na temat platformy TestNG i testów jednostkowych, przejdziesz bezpośrednio do wymagań związanych z następną aplikacją i przystąpisz do pisania kodu.

Oto zagadnienia omówione w tym rozdziale:

- testy jednostkowe,
- testy jednostkowe w TDD,
- platforma TestNG,
- wymagania dotyczące zdalnie sterowanego statku,
- pisanie kodu zdalnie sterowanego statku,
- podsumowanie.

Testy jednostkowe

Częste testy ręczne są akceptowalne tylko w trakcie pracy nad najmniejszymi systemami. Jedyny sposób na zastąpienie testów ręcznych to zastosowanie testów automatycznych. Są one jedyną skuteczną metodą pozwalającą skrócić czas i zmniejszyć koszty budowania, wdrażania i konserwowania aplikacji. Jeśli chcesz móc efektywnie zarządzać aplikacjami, niezwykle istotne jest, by kod rozwiązania i kod testów były jak najprostsze. Prostota jest jedną z podstawowych wartości programowania ekstremalnego (ang. *eXtreme Programming* — **XP**; *http://www.extremeprogramming.org/rules/simple.html*) oraz bardzo ważnym aspektem TDD i programowania w ogóle. Najczęściej prostotę zapewnia się dzięki podziałowi rozwiązania na małe jednostki. W Javie tymi jednostkami są metody. Ponieważ są one najmniejsze, pozwalają najszybciej uzyskać informacje zwrotne. Dlatego to właśnie metodom poświęca się najwięcej czasu w trakcie myślenia o rozwiązaniu i pracy nad nim. Odpowiednikiem metod z kodu rozwiązania są testy jednostkowe, które powinny stanowić największy procent wszystkich testów.

Czym są testy jednostkowe?

Testy jednostkowe to technika wymuszająca testowanie małych, pojedynczych i odizolowanych jednostek kodu. Tymi jednostkami są zwykle metody, choć czasem używane są klasy, a nawet całe aplikacje. Aby napisać test jednostkowy, badany kod trzeba odizolować od reszty aplikacji. Najlepiej, gdy izolacja ta jest wbudowana w kod lub gdy można ją osiągnąć za pomocą atrap (więcej o atrapach dowiesz się z rozdziału 6., „Eliminowanie zewnętrznych zależności za pomocą atrap"). Jeśli testy jednostkowe sprawdzanej metody wykraczają poza granice danej jednostki, stają się testami integracyjnymi. W takich testach trudniej jest ustalić, które fragmenty kodu są sprawdzane. Gdy test kończy się niepowodzeniem, możliwy zasięg wystąpienia usterki rośnie, przez co znalezienie źródła problemu staje się trudniejsze.

Po co stosować testy jednostkowe?

Standardowe pytanie, zadawane zwłaszcza w firmach, gdzie wykorzystuje się głównie testy ręczne, brzmi: „Dlaczego powinniśmy stosować testy jednostkowe zamiast testów funkcjonalnych lub integracyjnych?". Jest to błędnie postawione pytanie. Testy jednostkowe nie zastępują testów innego rodzaju, a jedynie pozwalają ograniczyć zakres pozostałych testów. Testy jednostkowe z natury pisze się łatwiej i szybciej niż testy innego typu. Pozwala to ograniczyć koszty i szybciej wprowadzić produkt na rynek. Ponieważ takie testy można szybko pisać i uruchamiać, znacznie szybciej wykrywane są też problemy. Im szybciej wykryjesz problem, tym tańsze będzie jego rozwiązanie. Błąd wykryty kilka minut po jego popełnieniu jest znacznie łatwiejszy do naprawienia niż ta sama usterka znaleziona po kilku dniach, tygodniach czy miesiącach.

Refaktoryzacja kodu

Refaktoryzacja kodu to proces modyfikowania struktury istniejącego kodu bez zmiany jego działania. Celem refaktoryzacji jest ulepszenie już napisanego kodu. Poprawki można wprowadzać z wielu różnych powodów. Możliwe, że chcesz zwiększyć czytelność kodu, zmniejszyć jego złożoność, ułatwić konserwację, ograniczyć koszty rozbudowywania rozwiązania i tak dalej. Niezależnie od przyczyn refaktoryzacji głównym jej celem zawsze jest ulepszenie kodu. Skutkiem realizacji tego celu jest zmniejszenie długu technicznego, co pozwala ograniczyć ilość pracy, którą trzeba wykonać z powodu nieoptymalnego projektu i kodu lub niedoskonałej architektury.

Refaktoryzacja zwykle polega na wprowadzaniu zestawu drobnych zmian bez modyfikowania pożądanego działania kodu. Ograniczenie zakresu zmian w trakcie refaktoryzacji pozwala zachować pewność, że poprawki nie naruszyły działania istniejących funkcji. Jedyny sposób na uzyskanie tej pewności polega na zastosowaniu zautomatyzowanych testów.

Jedną z istotnych zalet testów jednostkowych jest to, że są najlepszym narzędziem wspomagającym przeprowadzanie refaktoryzacji. Refaktoryzacja jest zbyt ryzykowna, jeśli nie istnieją zautomatyzowane testy potwierdzające, że aplikacja wciąż działa zgodnie z oczekiwaniami. Choć do zapewnienia pokrycia kodu testami potrzebnego w trakcie refaktoryzacji można wykorzystać testy dowolnego typu, zwykle tylko testy jednostkowe pozwalają zapewnić pożądany poziom szczegółowości.

Dlaczego nie ograniczyć się do stosowania samych testów jednostkowych?

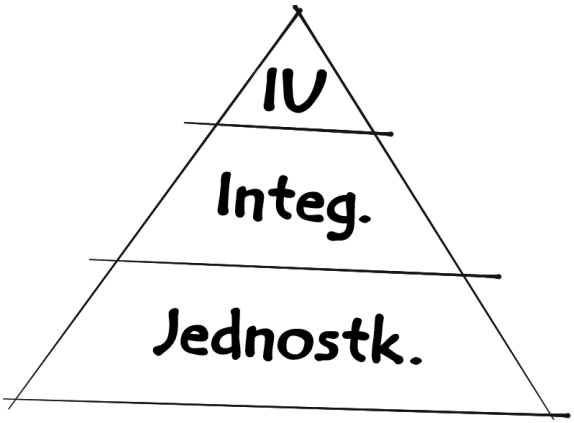
Możliwe, że zastanawiasz się, czy testy jednostkowe mogą zaspokoić wszystkie Twoje potrzeby związane z testami. Niestety, nie mogą. Choć testy jednostkowe zwykle pozwalają uwzględnić większość takich potrzeb, testy funkcjonalne i integracyjne też powinny być nieodłączną częścią Twojego zestawu narzędzi.

Inne typy testów są omówione szczegółowo w dalszych rozdziałach. Na razie warto zwrócić uwagę na kilka ważnych różnic między poszczególnymi rodzajami testów.

- **Testy jednostkowe** służą do sprawdzania niewielkich jednostek funkcyjnych. W Javie tymi jednostkami są metody. Wszystkie zewnętrzne zależności związane z wywołaniami innych klas, metod lub baz danych powinny być obsługiwane w pamięci za pomocą atrap, namiastek, „szpiegów" oraz zastępników typu fake i dummy. *Gerard Meszaros* wymyślił ogólne określenie „dublery używane w czasie testów" (ang. *test doubles*), które obejmuje wszystkie wymienione techniki (zobacz opis na stronie *http://en.wikipedia.org/wiki/Test_double*).

  Testy jednostkowe są proste, łatwo się je pisze i szybko wykonuje. Zwykle stanowią największą część w zestawie testów.
- **Testy funkcjonalne i akceptacyjne** służą do sprawdzania, czy rozwijana aplikacja działa poprawnie jako całość. Choć te dwa typy testów różnią się od siebie, używa się ich w tym samym celu. W odróżnieniu od testów jednostkowych, które sprawdzają wewnętrzną jakość kodu, testy funkcjonalne i akceptacyjne mają zapewniać, że system pracuje prawidłowo z perspektywy klienta lub użytkownika. Zazwyczaj liczba tych testów jest mniejsza niż liczba testów jednostkowych. Wynika to z dużego nakładu kosztów i wysiłku, jaki trzeba włożyć w pisanie testów tych dwóch typów i ich wykonywanie.
- **Testy integracyjne** mają sprawdzać, czy odrębne jednostki, moduły, aplikacje, a nawet całe systemy są prawidłowo zintegrowane ze sobą. Załóżmy, że korzystasz z frontonu, który na zapleczu używa interfejsu API, a ten z kolei komunikuje się z bazą danych. Testy integracyjne pozwolą sprawdzić, czy te trzy odrębne komponenty systemu są zintegrowane ze sobą i mogą się między sobą komunikować. Ponieważ wiadomo, że jednostki działają poprawnie i że wszystkie testy funkcjonalne oraz akceptacyjne zakończyły się powodzeniem, testy integracyjne zwykle stanowią najmniejszą grupę spośród trzech omawianych kategorii testów. Wynika to z tego, że testy integracyjne mają tylko zapewniać, że wszystkie elementy poprawnie ze sobą współpracują.

Widoczna na rysunku 4.1 piramida testów pokazuje, że testów jednostkowych powinno być znacznie więcej niż testów z wyższych poziomów (testów interfejsu użytkownika, testów integracyjnych i tak dalej). Z czego to wynika? Testy jednostkowe są znacznie tańsze do pisania, szybsze do uruchamiania i ponadto zapewniają wyższe pokrycie kodu. Pomyśl na przykład o funkcji rejestrowania się użytkownika. Należy sprawdzić, co się stanie, gdy pole na nazwę użytkownika jest puste, gdy pole na hasło jest puste, gdy nazwa użytkownika lub hasło ma zły format, gdy dany użytkownik już istnieje i tak dalej. Dla tej jednej funkcji można napisać dziesiątki, a nawet setki testów. Pisanie i uruchamianie takich testów z poziomu interfejsu użytkownika bywa bardzo kosztowne. Budowanie tego rodzaju testów jest czasochłonne, a ich wykonywanie zajmuje dużo czasu. Natomiast przeprowadzenie testu jednostkowego z użyciem metody, która sprawdza poprawność omawianej funkcji, wygląda inaczej. Test jest wtedy prosty, dzięki czemu można go szybko napisać i uruchomić. Jeśli wszystkie sytuacje są uwzględnione w testach jednostkowych, wystarczy wykonać jeden test integracyjny, sprawdzający, czy interfejs użytkownika wywołuje na zapleczu odpowiednią metodę. Jeżeli ją wywołuje, szczegóły jej działania w kontekście integracji nie mają znaczenia, ponieważ wiadomo, że wszystkie przypadki są sprawdzane na poziomie jednostkowym.



Rysunek 4.1. Piramida testów — testy interfejsu użytkownika (IU), integracyjne i jednostkowe

Testy jednostkowe w TDD

Jaka jest specyfika pisania testów jednostkowych w kontekście TDD? Najważniejszy jest tu *czas* pisania testów. W tradycyjnych podejściach testy jednostkowe są pisane po przygotowaniu kodu rozwiązania, w TDD natomiast testy są pisane na początku. Kolejność prac jest więc odwrócona. Tradycyjnie (bez stosowania TDD) celem testów jednostkowych jest sprawdzanie poprawności już istniejącego kodu. Zgodnie z TDD to testy jednostkowe powinny sterować tworzeniem kodu i projektu. To testy powinny definiować działanie możliwie najmniejszych jednostek. Testy są wtedy mikrowymaganiami czekającymi na spełnienie. Testy określają, co należy zrobić w następnym kroku i kiedy kod jest gotowy. W zależności od typu stosowanych testów (czy są to testy jednostkowe, funkcjonalne, czy integracyjne) zakres późniejszych działań jest inny. Gdy w TDD używane są testy jednostkowe, zakres prac jest najmniejszy i ogranicza się do metody lub, częściej, jej fragmentu. Ponadto należy wtedy stosować się do pewnych zasad projektowych, takich jak **KISS** (ang. *keep it simple stupid*, czyli „nie komplikuj, głupku"). Jeśli przygotujesz proste testy o bardzo niewielkim zakresie prac, kod pozwalający przejść te testy też zwykle będzie prosty. Dzięki wyeliminowaniu z testów zewnętrznych zależności trzeba odpowiednio zaprojektować w kodzie rozwiązania podział zadań. To tylko niektóre z wielu przykładów ilustrujących, że TDD pomaga pisać kod wyższej jakości. Tych samych korzyści nie da się uzyskać, stosując tylko testy jednostkowe. Bez TDD testy jednostkowe sprawdzają istniejący kod i nie mają wpływu na projekt.

Tak więc głównym celem testów jednostkowych w tradycyjnym podejściu (bez TDD) jest sprawdzanie poprawności gotowego kodu. Testy jednostkowe pisane na początku, w modelu TDD, służą głównie do tworzenia specyfikacji i projektu. Sprawdzanie poprawności jest tylko dodatkową korzyścią, przy czym testy mają wtedy zazwyczaj wyższą jakość niż wówczas, gdy powstają dopiero po napisaniu kodu rozwiązania.

TDD zmusza do przemyślenia wymagań i projektu, pisania przejrzystego i działającego kodu, tworzenia wykonywalnych wymagań oraz bezpiecznego i częstego refaktoryzowania rozwiązania. Ponadto uzyskujesz wysokie pokrycie kodu testami, co pozwala przeprowadzić testy regresyjne całego kodu po wprowadzeniu w nim zmian. Efektem testów jednostkowych bez TDD są tylko testy, i to często o wątpliwej jakości.

Platforma TestNG

JUnit i TestNG to dwie główne platformy do testowania kodu w Javie. W poprzednim rozdziale pisałeś już testy za pomocą platformy JUnit i, miejmy nadzieję, dobrze rozumiesz jej działanie. A co z TestNG? Ta platforma powstała w celu utworzenia ulepszonej wersji JUnit. I rzeczywiście — zawiera pewne funkcje niedostępne w JUnit.

W dalszych punktach znajdziesz przegląd różnic między tymi dwoma platformami. Autorzy tej książki starają się nie tylko wyjaśnić te różnice, ale też ocenić ich znaczenie w kontekście stosowania testów jednostkowych razem z TDD.

Adnotacja @Test

W platformach JUnit i TestNG adnotacja `@Test` jest używana do wskazywania metod uznawanych za test. W JUnit każda taka metoda musi mieć adnotację `@Test`, w TestNG natomiast można umieścić tę adnotację na poziomie klasy. Ta adnotacja na poziomie klasy sprawia, że wszystkie metody publiczne są uznawane za testy, chyba że programista doda do wybranych metod inne modyfikatory.

```
@Test
public class DirectionSpec {

    public void whenGetFromShortNameNThenReturnDirectionN() {
        Direction direction = Direction.getFromShortName('N');
        assertEquals(direction, Direction.NORTH);
    }

    public void whenGetFromShortNameWThenReturnDirectionW() {
        Direction direction = Direction.getFromShortName('W');
        assertEquals(direction, Direction.WEST);
    }
}
```

W tym przykładzie adnotacja `@Test` znajduje się nad klasą `DirectionSpec`. Dlatego metody `whenGetFromShortNameNThenReturnDirectionN` i `whenGetFromShortNameWThenReturnDirectionW` są uznawane za testy. W tym samym kodzie w platformie JUnit adnotacja `@Test` musi się znajdować przy obu metodach.

Adnotacje @BeforeSuit, @BeforeTest, @BeforeGroups, @AfterGroups, @AfterTest i @AfterSuit

Te adnotacje nie mają swoich odpowiedników w platformie JUnit. Platforma TestNG pozwala grupować testy w pakiety za pomocą konfiguracji w formacie XML. Metody z adnotacjami `@BeforeSuit` i `@AfterSuit` są uruchamiane przed testami z pakietu lub po całym pakiecie. Metody z adnotacjami `@BeforeTest` i `@AfterTest` są wykonywane przed każdą metodą testową z klasy albo po każdej takiej metodzie. Testy w platformie TestNG można też łączyć w grupy. Adnotacje `@BeforeGroup` i `@AfterGroup` umożliwiają uruchamianie metod przed pierwszym testem z grupy i po wykonaniu wszystkich testów z grupy.

Choć te adnotacje mogą być bardzo użyteczne, jeśli testy są pisane po kodzie rozwiązania, w kontekście TDD ich przydatność jest umiarkowana. Inaczej niż w tradycyjnych testach, które często planuje się i pisze w ramach odrębnego projektu, w TDD należy dodawać za każdym razem jeden prosty test. Jeszcze ważniejsze jest to, że testy mają działać szybko, dlatego nie trzeba łączyć ich w pakiety lub grupy. Gdy testy działają szybko, pomijanie którychś z nich jest marnotrawstwem. Jeśli wszystkie testy można wykonać na przykład w mniej niż 15 sekund, nie ma potrzeby uruchamiać tylko części z nich. Natomiast gdy testy działają powoli, często oznacza to, że nie odizolowano zewnętrznych zależności. Niezależnie od przyczyn wpływających na powolne wykonywanie testów rozwiązanie nie powinno polegać na wykonywaniu tylko wybranych z nich, ale na naprawieniu problemu.

Ponadto testy funkcjonalne i integracyjne są zwykle wolniejsze i wymagają rozdzielenia. Warto rozdzielić je na przykład za pomocą instrukcji z pliku *build.gradle*, tak by dla każdego typu testów używane było odrębne zadanie.

Adnotacje @BeforeClass i @AfterClass

Te adnotacje działają tak samo w platformach JUnit i TestNG. Metody z tymi adnotacjami są uruchamiane przed pierwszym testem i po ostatnim teście z danej klasy. Jedyna różnica polega na tym, że w platformie TestNG metody z tą adnotacją nie muszą być statyczne. Jest tak, ponieważ w obu platformach stosowane są odmienne podejścia w trakcie uruchamiania metod z testami. Platforma JUnit izoluje testy i uruchamia je w odrębnych egzemplarzach klasy z testami. Dlatego metody trzeba zdefiniować jako statyczne, co pozwala wielokrotnie wykorzystać je we wszystkich przebiegach testów. Platforma TestNG wykonuje wszystkie metody testowe w kontekście jednego egzemplarza klasy, metody nie muszą być więc statyczne.

Adnotacje @BeforeMethod i @AfterMethod

Adnotacje `@BeforeMethod` i `@AfterMethod` działają tak jak ich odpowiedniki z platformy JUnit. Metody z tymi adnotacjami są uruchamiane przed każdą metodą testową i po każdej takiej metodzie.

Argument w adnotacji @Test(enable = false)

Platformy JUnit i TestNG umożliwiają wyłączenie testów. W platformie JUnit służy do tego odrębna adnotacja `@Ignore`, w platformie TestNG natomiast należy użyć argumentu logicznego `enable` w adnotacji `@Test`. Oba te rozwiązania działają tak samo, a różnica polega tylko na zapisie.

Argument w adnotacji @Test(expectedExceptions = NazwaKlasy.class)

W tym punkcie platforma JUnit ma przewagę. Choć obie platformy umożliwiają wskazywanie oczekiwanych wyjątków w ten sam sposób (przy czym w platformie JUnit służący do tego argument nosi nazwę `expected`), JUnit udostępnia reguły, które pozwalają w bardziej elegancki sposób testować wyjątki. Z reguł korzystałeś już w rozdziale 2., „Narzędzia, platformy i środowiska".

Podsumowanie porównania platform TestNG i JUnit

Między omawianymi platformami występują też liczne inne różnice. By zachować zwięzłość, nie opisano ich wszystkich w tej książce. Więcej informacji znajdziesz w dokumentacji obu narzędzi.

> Więcej informacji o platformach JUnit i TestNG znajdziesz na stronach *http://junit.org/* i *http://testng.org/*.

Platforma TestNG udostępnia więcej funkcji i jest bardziej zaawansowana niż JUnit. W tym rozdziale będziesz używał platformy TestNG, dzięki czemu lepiej ją poznasz. Zauważysz jednak, że nie są tu stosowane żadne z zaawansowanych funkcji tej platformy. Wynika to z tego, że w TDD rzadko są one potrzebne w trakcie tworzenia testów jednostkowych. Testy funkcjonalne i integracyjne działają inaczej, dlatego pozwoliłyby lepiej zademonstrować przewagę platformy TestNG. Istnieją jednak narzędzia lepiej dostosowane do takich testów. Przekonasz się o tym w dalszych rozdziałach.

Której z tych platform powinieneś używać? Wybór należy do Ciebie. Do czasu zakończenia lektury tego rozdziału zdobędziesz praktyczne umiejętności posługiwania się zarówno platformą JUnit, jak i platformą TestNG.

Wymagania dotyczące zdalnie sterowanego statku

Tu wykonasz odmianę dobrze znanego ćwiczenia *Mars Rover*, opublikowanego po raz pierwszy w witrynie *Dallas Hack Club* (*http://dallashackclub.com/rover*).

Wyobraź sobie, że gdzieś po morzach pływa statek. Ponieważ żyjemy w XXI wieku, maszyną można sterować zdalnie.

> Zadanie polega na napisaniu programu, który pozwala sterować przemieszczaniem się statku po morzach.

Jako że jest to książka o TDD, a tematem tego rozdziału są testy jednostkowe, rozwiniesz aplikację za pomocą TDD, koncentrując się na testach jednostkowych. W poprzednim rozdziale poznałeś procedurę czerwone, zielone, refaktoryzacja w teorii, a także wypróbowałeś ją w praktyce. Tu wykorzystasz zdobytą wcześniej wiedzę i nauczysz się skutecznie stosować testy jednostkowe. Skoncentrujesz się na rozwijanej jednostce i dowiesz się, jak izolować i pomijać zależności, które mogą być potrzebne w danej jednostce. Następnym celem jest uwzględnianie w każdym kroku tylko jednego wymagania. Dlatego prezentowane są tu wyłącznie wymagania wysokiego poziomu. Użytkownik powinien móc zdalnie sterować statkiem zlokalizowanym w dowolnym miejscu świata.

W celu uproszczenia Ci pracy wszystkie klasy pomocnicze zostały już napisane i przetestowane. Dzięki temu możesz się skoncentrować tylko na głównym zadaniu, a ćwiczenie będzie zwięzłe.

Pisanie kodu do zdalnego sterowania statkiem

Zacznij od zaimportowania istniejącego repozytorium Git.

Przygotowywanie projektu

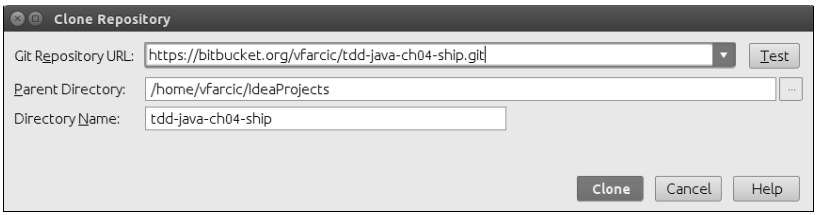
Rozpocznij od przygotowania projektu.

1. Otwórz środowisko IntelliJ IDEA. Jeśli otwarty jest istniejący projekt, wybierz opcję *File/Close Project*.

   Pojawi się ekran podobny do ekranu widocznego na rysunku 4.2.

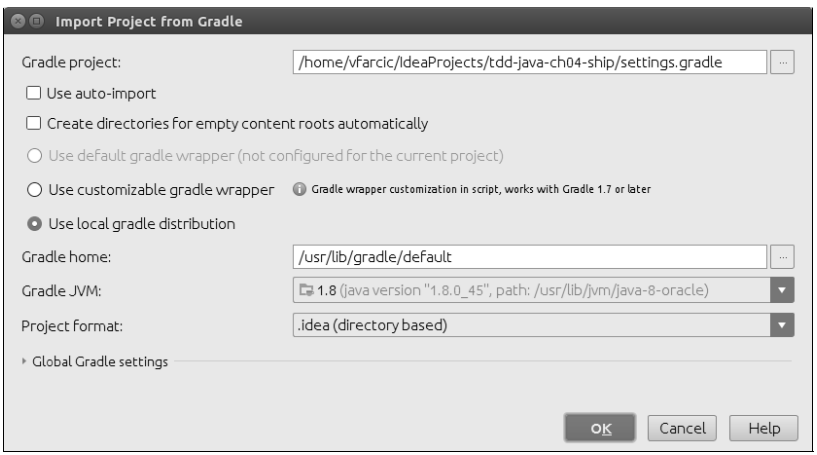
2. Aby zaimportować projekt z repozytorium Git, wybierz opcję *Check out from Version Control/Git*. Wpisz `https://bitbucket.org/vfarcic/tdd-java-ch04-ship.git` w polu *Git Repository URL* i kliknij przycisk *Clone* (zobacz rysunek 4.3).

Rysunek 4.2. Początkowy ekran środowiska IntelliJ IDEA



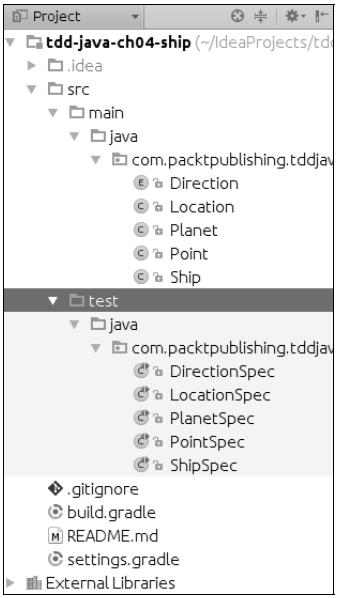
Rysunek 4.3. Importowanie projektu z repozytorium Git

3. Gdy zobaczysz pytanie o to, czy chcesz otworzyć projekt, wybierz opcję *Yes*. Następnie zobaczysz pokazane na rysunku 4.4 okno dialogowe *Import Project from Gradle*. Kliknij przycisk *OK*.



Rysunek 4.4. Otwieranie projektu

4. Środowisko IDEA potrzebuje czasu na pobranie wszystkich zależności opisanych w pliku *build.gradle*. Po zakończeniu ich wczytywania zobaczysz, że niektóre klasy i powiązane z nimi testy są już gotowe. Przedstawia to rysunek 4.5.



Rysunek 4.5. Struktura wczytanego projektu

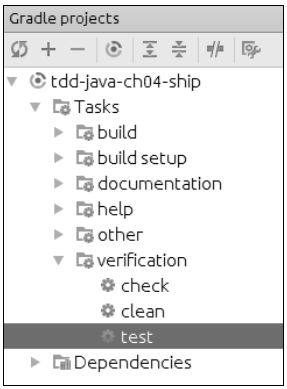
Klasy pomocnicze

Wyobraź sobie, że prace nad tym projektem rozpoczął Twój współpracownik. Jest dobrym programistą i praktykiem TDD. Ufasz, że zapewnił dobre pokrycie kodu testami, dlatego możesz polegać na jego pracy. Współpracownik nie zdążył jednak ukończyć aplikacji przed wyjazdem na urlop i to Ty musisz kontynuować rozwijanie programu. Gotowe są już wszystkie klasy pomocnicze: `Direction`, `Location`, `Planet` i `Point`. Zauważ, że przygotowane zostały też odpowiednie klasy z testami. Nazwy klas z testami utworzono przez dodanie przedrostka `Spec` do nazw sprawdzanych klas (na przykład `DirectionSpec`). Ten przedrostek zastosowano po to, by podkreślić, że testy nie tylko służą do sprawdzania poprawności kodu, ale także pełnią funkcję wykonywalnej specyfikacji.

Oprócz klas pomocniczych istnieją również klasy `Ship` (z rozwiązaniem) i `ShipSpec` (z testami używanymi jako specyfikacja). Większość czasu poświęcisz na pracę nad tymi dwoma klasami. W klasie `ShipSpec` będziesz pisał testy, a następnie dodasz kod rozwiązania do klasy `Ship`. Kolejność prac będzie więc taka sama jak wcześniej.

Ponieważ wiesz już, że testy nie tylko służą do sprawdzania poprawności kodu, ale też są wykonywalną dokumentacją, od tego miejsca używane będzie określenie *specyfikacja* zamiast *test*.

Za każdym razem, gdy napiszesz specyfikację lub odpowiadający jej kod, uruchom polecenie `gradle test` w wierszu poleceń bądź w przedstawionym na rysunku 4.6 oknie *Gradle projects* w środowisku IDEA.



Rysunek 4.6. Okno Gradle projects

Po przygotowaniu projektu możesz przejść do pierwszego wymagania.

Wymaganie nr 1

Trzeba określić bieżącą lokalizację statku, by móc go przemieścić. Ponadto trzeba wiedzieć, w którą stronę statek jest skierowany — na północ, na południe, na wschód czy na zachód. Tego właśnie dotyczy pierwsze wymaganie.

> Określone są punkt początkowy (**x, y**) statku oraz jego kierunek (**N** — ang. *north*, czyli północ, **S** — ang. *south*, czyli południe, **E** — ang. *east*, czyli wschód, lub **W** — ang. *west*, czyli zachód).

Zanim zaczniesz pracować nad wymaganiem, zapoznaj się z dostępnymi klasami pomocniczymi. Klasa `Point` przechowuje współrzędne `x` i `y`. Oto konstruktor tej klasy:

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

Dostępna jest też klasa `Direction enum` z pokazanymi poniżej wartościami:

```
public enum Direction {
    NORTH(0, 'N'),
    EAST(1, 'E'),
    SOUTH(2, 'S'),
    WEST(3, 'W'),
    NONE(4, 'X');
}
```

Istnieje również klasa `Location`, która wymaga przekazania obiektów obu wcześniej pokazanych klas jako argumentów konstruktora.

```
public Location(Point point, Direction direction) {
    this.point = point;
    this.direction = direction;
}
```

Dzięki tym klasom nie powinieneś mieć większych problemów z napisaniem testu dla pierwszego wymagania. Proces ten powinien wyglądać tak samo jak w poprzednim rozdziale.

Spróbuj samodzielnie napisać specyfikacje. Gdy je ukończysz, porównaj swój kod ze specyfikacją z tej książki. To samo zrób w trakcie pracy nad kodem rozwiązania powiązanym ze specyfikacją. Spróbuj napisać go samodzielnie, a następnie porównaj go z proponowanym rozwiązaniem z tej książki.

Specyfikacja

Oto możliwa specyfikacja dla pierwszego wymagania:

```
@Test
public class ShipSpec {

    public void whenInstantiatedThenLocationIsSet() {
        Location location = new Location(
        new Point(21, 13), Direction.NORTH);
        Ship ship = new Ship(location);
        assertEquals(ship.getLocation(), location);
    }
}
```

To było łatwe. Wystarczy sprawdzić, czy obiekt typu `Location` przekazany do konstruktora obiektu typu `Ship` jest zachowywany i czy można uzyskać dostęp do lokalizacji za pomocą gettera `location`.

> **Adnotacja @Test**
>
> Gdy w platformie TestNG adnotacja `@Test` jest ustawiona na poziomie klasy, nie trzeba określać, które metody mają pełnić funkcję testów. Wtedy wszystkie publiczne metody są uznawane za testy z platformy TestNG.

Kod rozwiązania

Kod rozwiązania powiązany ze specyfikacją jest stosunkowo łatwy do napisania. Wystarczy przypisać argument z konstruktora do zmiennej `location`.

```
public class Ship {

    private final Location location;
    public Location getLocation() {
        return location;
    }

    public Ship(Location location) {
        this.location = location;
    }
}
```

Kompletny kod źródłowy znajdziesz w odgałęzieniu `req1-location` w repozytorium `tdd-java-ch04-ship` dostępnym na stronie *https://bitbucket.org/vfarcic/tdd-java-ch04-ship/branch/req01-location*.

Refaktoryzacja

Wiesz, że obiekt typu `Ship` trzeba będzie utworzyć dla każdej specyfikacji. Dlatego można zrefaktoryzować klasę specyfikacji i dodać w niej adnotację `@BeforeMethod`. Oto nowy kod tej klasy:

```
@Test
public class ShipSpec {

    private Ship ship;
    private Location location;

    @BeforeMethod
    public void beforeTest() {
        Location location = new Location(
            new Point(21, 13), Direction.NORTH);
        ship = new Ship(location);
    }

    public void whenInstantiatedThenLocationIsSet() {
    // Location location = new Location(
    // new Point(21, 13), Direction.NORTH);
    // Ship ship = new Ship(location);
       assertEquals(ship.getLocation(), location);
    }
}
```

Nie dodano tu żadnych nowych operacji. Wystarczyło przenieść fragment kodu do adnotacji `@BeforeMethod`, by uniknąć powtórzeń, które powstałyby w trakcie pisania dalszych specyfikacji. Teraz przy każdym uruchomieniu testu utworzony zostanie obiekt typu `Ship` z argumentem `location`.

Wymaganie nr 2

Gdy wiesz już, gdzie statek się znajduje, spróbuj go przesunąć. Zacznij od możliwości przemieszczania go do przodu i do tyłu.

> Dodaj obsługę poleceń przesuwających statek naprzód (n) i wstecz (w).

Klasa pomocnicza `Location` udostępnia już metody `forward` i `backward` obsługujące takie polecenia.

```
public boolean forward() {
    ...
}
```

Specyfikacja

Co się powinno stać, gdy statek jest skierowany na północ i użytkownik przesunie go do przodu? Wartość współrzędnej `y` powinna się zmniejszyć. Inny przykład to przesunięcie do przodu statku skierowanego na wschód; w takiej sytuacji należy zwiększyć wartość współrzędnej `x`.

Pierwszą reakcją może być napisanie specyfikacji podobnych do dwóch poniższych:

```
public void givenNorthWhenMoveForwardThenYDecreases() {
    ship.moveForward();
    assertEquals(ship.getLocation().getPoint().getY(), 12);
}

public void givenEastWhenMoveForwardThenXIncreases() {
    ship.getLocation().setDirection(Direction.EAST);
    ship.moveForward();
    assertEquals(ship.getLocation().getPoint().getX(), 22);
}
```

W tym podejściu trzeba utworzyć jeszcze przynajmniej dwie specyfikacje dotyczące sytuacji, w których statek jest skierowany na południe i na zachód.

Jednak nie tak należy pisać testy jednostkowe. Większość osób zaczynających stosowanie takich testów wpada w pułapkę określania wyników końcowych wymagających znajomości wewnętrznych mechanizmów metod, klas i bibliotek używanych przez sprawdzaną metodę. To podejście prowadzi do problemów na wielu poziomach.

Gdy w sprawdzanej jednostce korzystasz z kodu zewnętrznego, powinieneś pamiętać o tym, że — przynajmniej w tym przykładzie — zewnętrzny kod jest już przetestowany. Po każdej zmianie wykonywane są wszystkie testy, dlatego wiadomo, że kod zewnętrzny działa.

> **Zawsze po wprowadzeniu zmian w kodzie rozwiązania uruchamiaj wszystkie testy**
>
> To gwarantuje, że nie pojawią się nieoczekiwane efekty uboczne spowodowane zmianami w kodzie.
>
> Za każdym razem, gdy zmodyfikujesz jakąś część kodu rozwiązania, przeprowadź wszystkie testy. W idealnych warunkach testy powinny działać szybko i być wykonywane lokalnie przez programistę. Po przesłaniu kodu do systemu kontroli wersji wszystkie testy należy przeprowadzić jeszcze raz, by się upewnić, że nie wystąpiły problemy wynikające ze scalania kodu. Jest to ważne zwłaszcza wtedy, gdy nad kodem pracuje kilka osób. Do pobrania kodu z repozytorium, skompilowania rozwiązania i wykonania testów można wykorzystać narzędzia do obsługi ciągłej integracji, takie jak Jenkins, Hudson, Travind, Bamboo lub Go-CD.

Inny problem z opisanym podejściem polega na tym, że jeśli w zewnętrznym kodzie pojawią się zmiany, trzeba będzie zmodyfikować wiele specyfikacji. Najlepiej jest, gdy zmiany są niezbędne tylko w specyfikacjach bezpośrednio powiązanych z modyfikowaną jednostką. Wyszukiwanie w innych miejscach wywołań danej jednostki bywa bardzo czasochłonne i grozi powstawaniem błędów.

Poniżej pokazano znacznie łatwiejszy, szybszy i lepszy sposób na napisanie specyfikacji dla omawianego wymagania.

```
public void whenMoveForwardThenForward() {
    Location expected = location.copy();
    expected.forward();
    ship.moveForward();
    assertEquals(ship.getLocation(), expected);
}
```

Ponieważ klasa `Location` ma już metodę `forward`, wystarczy się upewnić, że metoda ta zostanie poprawnie wywołana. Tu kod tworzy nowy obiekt `expected` typu `Location`, wywołuje jego metodę `forward` i porównuje lokalizację obiekt `expected` z lokalizacją statku po wywołaniu jego metody `moveForward`.

Zauważ, że specyfikacje służą nie tylko do sprawdzania poprawności kodu. Są również używane jako wykonywalna dokumentacja i, co najważniejsze, wspomagają myślenie oraz projektowanie. Druga wersja specyfikacji bardziej precyzyjnie określa jej przeznaczenie. W klasie `Ship` należy dodać metodę `moveForward` i wywołać w niej instrukcję `location.forward`.

Kod rozwiązania

Po przygotowaniu tak krótkiej i jasno zdefiniowanej specyfikacji napisanie kodu, który pozwala ją spełnić, powinno być łatwe.

```
public boolean moveForward() {
    return location.forward();
}
```

Specyfikacja

Po utworzeniu specyfikacji dotyczącej ruchu do przodu i napisaniu odpowiadającego jej kodu rozwiązania pora zająć się ruchem wstecz. Specyfikacja wygląda tu prawie identycznie.

```
public void whenMoveBackwardThenBackward() {
    Location expected = location.copy();
    expected.backward();
    ship.moveBackward();
    assertEquals(ship.getLocation(), expected);
}
```

Kod rozwiązania

Kod rozwiązania związany z ruchem wstecz jest, podobnie jak kod specyfikacji, łatwy do napisania.

```
public boolean moveBackward() {
    return location.backward();
}
```

Kompletny kod źródłowy dotyczący tego wymagania znajdziesz w odgałęzieniu `req02-forward-backward` w repozytorium `tdd-java-ch04-ship` dostępnym na stronie *https://bitbucket.org/vfarcic/tdd-java-ch04-ship/branch/req02-forward-backward*.

Wymaganie nr 3

Przemieszczanie statku tylko do przodu i do tyłu nie wystarczy. Potrzebna jest też możliwość sterowania statkiem i robienia zwrotów w lewo oraz w prawo.

> Dodaj polecenia powodujące obrót statku w lewo (l) i prawo (p).

Po dodaniu kodu rozwiązania dla poprzedniego wymagania nowe zadanie nie powinno sprawić Ci żadnych trudności, ponieważ można w nim wykorzystać tę samą logikę. Klasa pomocnicza `Location` zawiera już metody `turnLeft` i `turnRight`, które wykonują określone w wymaganiu operacje. Teraz wystarczy zintegrować te metody z klasą `Ship`.

Specyfikacja

Oto specyfikacja zwrotu w lewo oparta na tych samych pomysłach, które wykorzystano wcześniej:

```
public void whenTurnLeftThenLeft() {
    Location expected = location.copy();
    expected.turnLeft();
    ship.turnLeft();
    assertEquals(ship.getLocation(), expected);
}
```

Kod rozwiązania

Napisanie kodu dla nowej specyfikacji prawdopodobnie nie sprawiło Ci trudności.

```
public void turnLeft() {
    location.turnLeft();
}
```

Specyfikacja

Specyfikacja zwrotu w prawo powinna wyglądać niemal identycznie jak specyfikacja zwrotu w lewo.

```
public void whenTurnRightThenRight() {
    Location expected = location.copy();
    expected.turnRight();
    ship.turnRight();
    assertEquals(ship.getLocation(), expected);
}
```

Kod rozwiązania

Teraz zakończ prace nad omawianym wymaganiem i dodaj kod rozwiązania odpowiadający specyfikacji zwrotu w prawo.

```
public void turnRight() {
    location.turnRight();
}
```

Kompletny kod źródłowy dotyczący tego wymagania znajdziesz w odgałęzieniu `req3-left-right` w repozytorium `tdd-java-ch04-ship`. Kod ten jest dostępny na stronie *https://bitbucket.org/vfarcic/tdd-java-ch04-ship/branch/req03-left-right*.

Wymaganie nr 4

Wszystkie dotychczasowe zadania były stosunkowo łatwe, ponieważ potrzebne funkcje były dostępne w klasach pomocniczych. Dzięki temu dowiedziałeś się, że trzeba się skupić na rozwijanej jednostce, nie należy natomiast próbować testować efektów końcowych. W ten sposób możesz zbudować zaufanie. Powinieneś ufać, że opracowany przez innych kod (klasy pomocnicze) jest prawidłowy. Począwszy od wymagania nr 4, będziesz musiał zacząć wierzyć w poprawność kodu napisanego samodzielnie. Prace będą się toczyły w takim samym modelu jak wcześniej. Napiszesz specyfikację, uruchomisz testy, wykryjesz niepowodzenie, napiszesz kod rozwiązania, wykonasz testy, stwierdzisz, że kończą się sukcesem, a w ostatnim kroku przeprowadzisz refaktoryzację (jeśli uznasz, że kod można poprawić). Staraj się myśleć o tym, jak przetestować jednostkę (metodę) bez zagłębiania się w działanie metod i klas wywoływanych przez tę jednostkę.

Po dodaniu obsługi poszczególnych instrukcji (naprzód, w tył, w lewo i w prawo) pora połączyć całe rozwiązanie w jedną całość. Należy utworzyć metodę, która pozwoli przekazać dowolną sekwencję poleceń w jednym łańcuchu znaków. Każde polecenie to litera: **n** oznacza „naprzód", **w** to „wstecz", **l** to „lewo", a **p** to „prawo".

> Statek przyjmuje łańcuch znaków z poleceniami (literami `nwlp`, oznaczającymi naprzód, wstecz, lewo i prawo).

Specyfikacja

Zacznij od polecenia z argumentem obejmującym tylko literę `n` (naprzód).

```
public void whenReceiveCommandsFThenForward() {
    Location expected = location.copy();
    expected.forward();
    ship.receiveCommands("f");
    assertEquals(ship.getLocation(), expected);
}
```

Ta specyfikacja wygląda niemal tak samo jak specyfikacja `whenMoveForwardThenForward`, jednak tym razem wywoływana jest metoda `ship.receiveCommands("f")`.

Kod rozwiązania

Wcześniej napisano już o tym, jak ważne jest pisanie możliwie najprostszego kodu zgodnego ze specyfikacją.

> **Pisz jak najprostszy kod, który przejdzie testy. Dzięki temu uzyskasz bardziej uporządkowany i przejrzysty projekt oraz unikniesz dodawania zbędnych funkcji**
>
> To podejście oparte jest na założeniu, że im prostszy jest kod rozwiązania, tym łatwiejsza będzie jego konserwacja. Jest to zgodne z regułą KISS. Mówi ona, że większość systemów pracuje najlepiej, jeśli systemy te są proste, a nie skomplikowane. Dlatego prostota powinna być podstawowym celem w trakcie tworzenia projektu. Należy natomiast unikać zbędnych komplikacji.

Jest to dobry moment na zastosowanie wspomnianej reguły. Możliwe, że chcesz napisać fragment kodu podobny do poniższego:

```
public void receiveCommands(String commands) {
    if (commands.charAt(0) == 'f') {
        moveForward();
    }
}
```

Ten przykładowy kod sprawdza, czy pierwsza litera to `f`. Jeśli tak jest, kod wywołuje metodę `moveForward`. Istnieje też wiele innych możliwych wersji tego kodu. Jeżeli jednak stosujesz się do reguły zachęcającej do zachowania prostoty, powinieneś napisać lepsze rozwiązanie:

```
public void receiveCommands(String command) {
    moveForward();
}
```

Jest to najprostszy i najkrótszy kod, który pozwala przejść test. Później możesz uzyskać coś bardziej podobnego do pierwszej wersji kodu. Możliwe, że zastosujesz pętlę lub wymyślisz inne rozwiązanie, gdy zadanie stanie się bardziej skomplikowane. Jednak na razie należy się skoncentrować na jednej specyfikacji i na tworzeniu prostego kodu. Postaraj się zachować jasność umysłu, skupiając się tylko na wykonywanym zadaniu.

W celu zachowania zwięzłości pozostałe specyfikacje i rozwiązania (dla liter `w`, `l` i `p`) pominięto. Potrzebny kod powinieneś napisać samodzielnie. Następnie przejdź do ostatniej specyfikacji dla omawianego wymagania.

Specyfikacja

Teraz gdy kod potrafi już przetwarzać pojedyncze polecenia (niezależnie od tego, jaka jest podana instrukcja), można dodać mechanizm przesyłania łańcuchów poleceń. Oto specyfikacja tego zadania:

```
public void whenReceiveCommandsThenAllAreExecuted() {
    Location expected = location.copy();
    expected.turnRight();
    expected.forward();
    expected.turnLeft();
    expected.backward();
    ship.receiveCommands("pnlw");
    assertEquals(ship.getLocation(), expected);
}
```

Ta specyfikacja jest nieco dłuższa od poprzednich, ale nie jest skomplikowana. Kod przekazuje polecenia `pnlw` (czyli w prawo, naprzód, w lewo, wstecz) i oczekuje, że lokalizacja odpowiednio się zmieni. Tak jak wcześniej, kod nie sprawdza wyniku końcowego (czyli nie określa, czy zmieniły się współrzędne), a jedynie bada, czy używane są prawidłowe wywołania metod pomocniczych.

Kod rozwiązania

Oto efekt końcowy:

```
public void receiveCommands(String commands) {
    for (char command : commands.toCharArray()) {
        switch(command) {
            case 'n':
                moveForward();
```

```
                    break;
                case 'w':
                    moveBackward();
                    break;
                case 'l':
                    turnLeft();
                    break;
                case 'p':
                    turnRight();
                    break;
            }
        }
    }
```

Jeśli próbowałeś samodzielnie pisać specyfikacje i kod rozwiązania oraz starałeś się zachować przy tym prostotę, w drodze do ostatecznego rozwiązania prawdopodobnie kilkakrotnie musiałeś przeprowadzić refaktoryzację. Prostota jest niezwykle istotna, a refaktoryzacja często jest korzystną koniecznością. W trakcie refaktoryzacji pamiętaj, że kod zawsze musi być zgodny ze wszystkimi specyfikacjami.

Przeprowadzaj refaktoryzację tylko wtedy, gdy wszystkie testy kończą się powodzeniem

Korzyść z tego jest taka, że refaktoryzację można wtedy przeprowadzić bezpiecznie.

Jeśli cały kod rozwiązania, który może zostać zmodyfikowany, jest pokryty testami, a wszystkie testy kończą się powodzeniem, refaktoryzacja jest stosunkowo bezpieczna. W większości sytuacji refaktoryzacja nie wymaga dodawania nowych testów — wystarczą drobne zmiany w istniejących testach. Gdy przeprowadzasz refaktoryzację, wszystkie testy powinny się kończyć powodzeniem zarówno przed zmodyfikowaniem kodu, jak i po jego zmianie.

Kompletny kod źródłowy dotyczący tego wymagania znajdziesz w odgałęzieniu `req04-commands` w repozytorium `tdd-java-ch04-ship`. Kod ten jest dostępny na stronie *https://bitbucket.org/vfarcic/tdd-java-ch04-ship/branch/req04-commands*.

Wymaganie nr 5

Ziemia, podobnie jak inne planety, jest kulą. Gdy Ziemię przedstawia się na mapie, po dotarciu do jednego krańca należy przeskoczyć do drugiego. Na przykład jeśli kierujesz się na wschód i docierasz do najdalszego punktu na Pacyfiku, powinieneś przeskoczyć do zachodniej części mapy i kontynuować przemieszczanie się w kierunku Ameryki. Ponadto by uprościć poruszanie się, można zdefiniować mapę jako siatkę. Szerokość i wysokość na siatce należy wtedy przedstawić za pomocą osi X i Y. Tak więc siatka będzie miała maksymalną szerokość (X) i wysokość (Y).

Dodaj obsługę przeskakiwania od jednej krawędzi siatki do innej.

Specyfikacja

Pierwszą rzeczą, jaką trzeba zrobić, jest przekazanie do konstruktora klasy `Ship` obiektu typu `Planet` z maksymalnymi współrzędnymi osi X i Y. Na szczęście `Planet` to jedna z już gotowych (i przetestowanych) klas pomocniczych. Teraz wystarczy utworzyć obiekt tej klasy i przekazać go do konstruktora klasy `Ship`.

```
public void whenInstantiatedThenPlanetIsStored() {
    Point max = new Point(50, 50);
    Planet planet = new Planet(max);
    ship = new Ship(location, planet);
    assertEquals(ship.getPlanet(), planet);
}
```

W tym kodzie zdefiniowano wymiary planety na 50 × 50 jednostek i przekazano je do obiektu klasy `Planet`. Następnie ten obiekt przekazano do konstruktora klasy `Ship`. Może zauważyłeś, że konstruktor będzie wymagał teraz dodatkowego argumentu. W obecnej postaci kodu konstruktor przyjmuje tylko obiekt typu `Location`. Aby napisać kod dla nowej specyfikacji, trzeba sprawić, by konstruktor pobierał także obiekt typu `Planet`.

Jak napiszesz rozwiązanie dla nowej specyfikacji, nie naruszając przy tym żadnych istniejących specyfikacji?

Kod rozwiązania

Zastosuj podejście „od szczegółu do ogółu". Zgodnie z asercją dostępny powinien być getter dla obiektu `planet`.

```
private Planet planet;
public Planet getPlanet() {
    return planet;
}
```

Konstruktor powinien przyjmować obiekt typu `Planet` jako drugi argument i przypisywać go do wcześniej dodanej zmiennej `planet`. Pierwszym pomysłem może być dodanie nowego argumentu do istniejącego konstruktora, to jednak prowadzi do naruszenia wielu istniejących specyfikacji, używających konstruktora jednoargumentowego. Pozostaje więc tylko jedna możliwość — napisać drugi konstruktor.

```
public Ship(Location location) {
    this.location = location;
}
public Ship(Location location, Planet planet) {
    this.location = location;
    this.planet = planet;
}
```

Uruchom wszystkie specyfikacje i upewnij się, że kończą się powodzeniem.

Refaktoryzacja

Specyfikacje wymusiły napisanie drugiego konstruktora, ponieważ modyfikacja pierwszego spowodowałaby naruszenie istniejących testów. Jednak teraz, gdy stan rozwiązania to „zielone", można przeprowadzić refaktoryzację i pozbyć się konstruktora jednoargumentowego. W klasie specyfikacji znajduje się już uruchamiana przed każdym testem metoda z adnotacją `@beforeTest`. Dlatego możesz przenieść prawie cały kod nowej specyfikacji (oprócz samej asercji) do tej metody.

```
public class ShipSpec {
    ...
    private Planet planet;

    @BeforeMethod
    public void beforeTest() {
        Point max = new Point(50, 50);
        location = new Location(new Point(21, 13), Direction.NORTH);
        planet = new Planet(max);
        // ship = new Ship(location);
        ship = new Ship(location, planet);
    }

    public void whenInstantiatedThenPlanetIsStored() {
        // Point max = new Point(50, 50);
        // Planet planet = new Planet(max);
        // ship = new Ship(location, planet);
        assertEquals(ship.getPlanet(), planet);
    }
}
```

Ta zmiana pozwoliła wyeliminować przypadki użycia jednoargumentowego konstruktora klasy `Ship`. Jeśli uruchomisz teraz wszystkie specyfikacje, przekonasz się, że zmiana zadziałała.

Teraz gdy jednoargumentowy konstruktor nie jest już używany, można usunąć go także z klasy z kodem rozwiązania.

```
public class Ship {
    ...
    // public Ship(Location location) {
    // this.location = location;
    // }
    public Ship(Location location, Planet planet) {
        this.location = location;
        this.planet = planet;
    }
...
}
```

Dzięki zastosowaniu tego podejścia wszystkie specyfikacje cały czas pozostały w stanie „zielone". Refaktoryzacja nie spowodowała zmiany istniejących funkcji, nic nie zostało uszkodzone, a cała procedura została szybko ukończona.

Teraz pora przejść do obsługi przeskakiwania między krawędziami mapy.

Specyfikacja

Podobnie jak w innych przykładach, tak i tu klasy pomocnicze już udostępniają wszystkie potrzebne funkcje. Do tej pory używana była metoda `Location.forward` bez argumentów. Aby dodać obsługę przeskakiwania między krawędziami, będziesz potrzebował przeciążonej wersji tej metody, `Location.forward(Point max)`, która po dotarciu statku do krańca siatki powoduje przeniesienie go do odpowiedniej innej krawędzi. W poprzedniej specyfikacji zapewniono, że obiekt typu `Planet` jest przekazywany do konstruktora klasy `Ship` i że przyjmuje argument `Point max`. Zadanie polega na upewnieniu się, że wartość `max` będzie uwzględniana w trakcie poruszania się naprzód. Specyfikacja może wyglądać tak:

```
/* Nazwa metody została skrócona z powodu ograniczenia długości wiersza.
Ten test ma sprawdzać zachowanie statku w sytuacji, gdy
pojawi się polecenie nakazujące wyjście poza prawą granicę siatki.
*/
public void overpassEastBoundary() {
    location.setDirection(Direction.EAST);
    location.getPoint().setX(planet.getMax().getX());
    ship.receiveCommands("f");
    assertEquals(location.getX(), 1);
}
```

Kod rozwiązania

Do tej pory zapewne zdążyłeś się już przyzwyczaić do koncentrowania się na konkretnych jednostkach i nabrałeś zaufania co do tego, że kod napisany wcześniej działa zgodnie z oczekiwaniami. Teraz też powinieneś pracować w standardowy sposób. Wystarczy się upewnić, że w wywołaniu `location.forward` używana jest maksymalna wartość współrzędnych.

```
public boolean moveForward() {
    // return location.forward();
    return location.forward(planet.getMax());
}
```

Tę samą specyfikację i ten sam kod rozwiązania należy zastosować dla przemieszczania się wstecz. Aby zachować zwięzłość, kod dotyczący cofania się pominięto w tej książce, znajdziesz go jednak w kodzie źródłowym.

Kompletny kod źródłowy związany z tym wymaganiem znajduje się w odgałęzieniu `req05-wrap` z repozytorium `tdd-java-ch04-ship`. Kod dostępny jest na stronie *https://bitbucket.org/vfarcic/tdd-java-ch04-ship/branch/req05-wrap*.

Wymaganie nr 6

To już prawie koniec — wystarczy uwzględnić ostatnie wymaganie.

Choć większa część powierzchni Ziemi (około 70%) jest pokryta wodą, znajdują się na niej kontynenty i wyspy, które należy traktować jak przeszkody dla zdalnie sterowanego statku. Potrzebny jest sposób na wykrywanie, czy następny ruch nie spowoduje zderzenia z jedną z przeszkód. Po wykryciu lądu ruch należy anulować, a statek powinien pozostać na obecnej pozycji i zgłosić napotkanie przeszkody.

> Dodaj obsługę wykrywania lądu. Wykrywanie powinno się odbywać przed każdym ruchem na nową pozycję. Jeśli polecenie wymaga wpłynięcia na ląd, statek ma anulować ruch, pozostać na obecnej pozycji i zgłosić napotkanie przeszkody.

Kod specyfikacji i rozwiązania dla tego wymagania jest bardzo podobny do wcześniej prezentowanych fragmentów, dlatego możesz napisać go samodzielnie.

Oto kilka wskazówek, które mogą się okazać przydatne:

- Klasa `Planet` ma konstruktor przyjmujący listę przeszkód. Każda przeszkoda to obiekt klasy `Point`.
- Metody `Location.forward` i `Location.backward` mają przeciążone wersje, przyjmujące listę przeszkód. Te wersje zwracają wartość `true`, jeśli ruch jest możliwy, i wartość `false`, jeżeli statku nie da się przesunąć. Wykorzystaj zwróconą wartość logiczną do wygenerowania raportu o stanie potrzebnego w metodzie `Ship.receiveCommands`.
- Metoda `receiveCommands` powinna zwracać łańcuch znaków ze stanem każdego polecenia. `O` może reprezentować brak problemów, a `X` może oznaczać brak możliwości ruchu. Sekwencja OOXO oznacza więc: OK, OK, niepowodzenie, OK.

Kompletny kod źródłowy dla tego wymagania znajdziesz w odgałęzieniu `req06-obstacles` z repozytorium `tdd-java-ch04-ship`. Kod jest dostępny na stronie *https://bitbucket.org/vfarcic/tdd-java-ch04-ship/branch/req06-obstacles*.

Podsumowanie

W tym rozdziale wykorzystano platformę TestNG jako platformę testową. Nie pojawiły się tu istotne różnice w porównaniu z platformą JUnit, ponieważ nie zastosowano żadnych zaawansowanych mechanizmów z TestNG (na przykład dostawców danych, metod fabrycznych i tak dalej). W przypadku TDD wątpliwe jest to, czy takie funkcje w ogóle są potrzebne.

Odwiedź witrynę *http://testng.org/*, zapoznaj się z jej zawartością i sam zdecyduj, która platforma jest najlepiej dostosowana do Twoich potrzeb.

Głównym celem tego rozdziału było pokazanie, w jaki sposób zachować koncentrację na opracowywanej jednostce. Od początku dostępne były liczne klasy pomocnicze, których wewnętrzne mechanizmy starano się ignorować. W wielu sytuacjach nie powstały specyfikacje sprawdzające poprawność wyniku końcowego. Zamiast tego sprawdzano, czy wywoływane są odpowiednie metody z klas pomocniczych. W *praktyce* będziesz pracował nad projektami razem z innymi członkami zespołu. Dlatego ważne jest, by nauczyć się koncentrować na swoich zadaniach i ufać, że inni poprawnie wykonają swoją pracę. To samo dotyczy korzystania z niezależnych bibliotek. Testowanie wszystkich wewnętrznych procesów, wykonywanych w trakcie wywoływania takich bibliotek, byłoby zbyt kosztowne. Istnieją inne typy testów, w których można spróbować uwzględnić zewnętrzne komponenty. Gdy korzystasz z testów jednostkowych, powinieneś się skupić wyłącznie na rozwijanej jednostce.

Teraz gdy już wiesz, jak skutecznie posługiwać się testami jednostkowymi w TDD, pora przejść do innych zalet TDD. Zobaczysz, jakie są lepsze sposoby na projektowanie aplikacji.

Skorowidz

P

Q

R

S

Ś

T

U

W

X

Y

Z

Ż

TDD Programowanie w Javie sterowane testami

Programowanie sterowane testami (ang. *test-driven development* — TDD) nie jest nową metodyką. Jej praktyczne zastosowanie pozwala na rozwiązanie wielu problemów związanych z procesami rozwijania i wdrażania oprogramowania. Mimo ogromnych zalet programowanie sterowane testami nie jest zbyt popularne wśród programistów. Wynika to z tego, że techniki TDD nie są łatwe do opanowania. Choć teoretyczne podstawy wydają się logiczne i zrozumiałe, nabranie wprawy w stosowaniu TDD wymaga długiej praktyki.

Książka, którą trzymasz w rękach, została napisana przez programistów dla programistów. Jej celem jest przekazanie podstaw TDD i omówienie najważniejszych praktyk związanych z tą metodyką, a przede wszystkim — nauczenie Czytelnika praktycznego stosowania TDD w pracy. Autorzy nie ukrywają, że nabranie biegłości w takim programowaniu wymaga sporo wysiłku, jednak korzyści płynące z metodyki TDD są znaczne: skrócenie czasu wprowadzania produktów na rynek, łatwiejsza refaktoryzacja, a także wyższa jakość tworzonych projektów. Z tą książką dogłębnie zrozumiesz metodykę TDD i uzyskasz wystarczającą pewność siebie, by z powodzeniem stosować to podejście w trakcie programowania aplikacji w Javie.

Programowanie sterowane testami to metodyka dla prawdziwych profesjonalistów!

Dzięki tej książce:

- nauczysz się podstaw metodyki TDD
- poznasz potrzebne narzędzia, platformy i środowiska wraz ze szczegółami ich konfiguracji
- wykonasz praktyczne ćwiczenia i stopniowo wdrożysz się w TDD
- poznasz proces „czerwone, zielone, refaktoryzacja”
- dowiesz się, jak pisać testy jednostkowe wykonywane w izolacji od reszty kodu

Viktor Farcic — architekt oprogramowania. Tworzył w wielu językach, również w tak klasycznych jak Pascal, Basic i ASP. Programuje w C, C++, Perlu, Pythonie, ASP.Net, Visual Basicu, C#, choć obecnie najchętniej posługuje się Scalą i JavaScriptem, a także Javą. Jest niekwestionowanym znawcą takich metodyk jak programowanie sterowane testami, programowanie sterowane zachowaniami oraz techniki ciągłej integracji, ciągłego dostarczania i ciągłego wdrażania.

Alex Garcia — początkowo programował w języku C++, ale później wybrał Javę. Interesują go też języki: Groovy, Scala i JavaScript. Pracował jako administrator systemów, a także jako programista i konsultant. Garcia jest wielkim zwolennikiem metodyk zwinnych. Pasjonuje się nowymi językami, paradygmatami i platformami.

ISBN 978-83-283-2341-4