METODY TESTOWANIA OPROGRAMOWANIA

LABORATORIUM 3

Izolacja testu jednostkowego, frameworki izolujące

Jakość testu - Test Builder

wersja 1.1

przygotował:
dr inż. Radosław Adamus

.

Efekty:

Po ukończeniu laboratorium będziesz:
1. Potrafi wykorzystać framework izolujący do przeprowadzenia testu stanu i zachowania.
2..Potrafił wykonać test wymagający izolacji w kontekście metod statycznych oraz logiki upływu czasu.
3. Potrafił wykorzystać wzorzec Test Builder do przygotowania danych wejściowych dla przypadku testowego.

Wymagania wstępne:

1. Posiadanie konta na platformie Github.

Narzędzia:

1. Eclipse IDE
2. Git

Reguły wykonywania ćwiczeń laboratoryjnych:

1. Zmiany należy zatwierdzać często. Zatwierdzenie zbiorcze zmian na koniec laboratorium równoważne jest z jego niezaliczeniem.
2. Zmiany zatwierdzone w repozytorium kontroli wersji muszą posiadać znaczące komentarze.
3. Git powinien być tak skonfigurowany, aby zatwierdzane zmiany były identyfikowane danymi studenta (adres email oraz nazwisko).
4. W przypadku nieukończenia zadania w trakcie zajęć rezultat pośredni powinien być, po zatwierdzeniu w repozytorium lokalnym wypchnięte do macierzystego repozytorium na GitHub.

Opis laboratorium:

1. Wykorzystanie frameworka izolującego w testach jednostkowych:

1.1 BookKeeper

Sklonuj (operacja fork) repozytorium https://github.com/mto-lab/lab3_1 na swoje konto GitHub (a następnie na lokalny komputer). Zaimportuj projekt do IDE.
Odszukaj klasę pl.com.bottega.ecommerce.sales.domain.invoicing.BookKeeper. Zadaniem bedzie przetestowanie tej metody. Aby tego dokonać należy:

a. Określić przypadki testowe
b. Przygotować dane wejściowe
c. Przygotować dublerów dla zależności

Przypadki testowe zdefiniujemy w taki sposób aby można było wykonać test stanu oraz test zachowania. Dla testu stanu zdefiniujemy następujący przypadek testowy:

Test case 1	żądanie wydania faktury z jedną pozycją powinno zwrócić fakturę z jedną pozycją
Given	żądanie faktury z pojedynczą pozycją
When	wywołanie metody issuance
Then	faktura z jedną pozycją

Test zachowania reprezentował będzie następujący przypadek testowy:

Test case 2	żądanie wydania faktury z dwiema pozycjami powinno wywołać metodę calculateTax dwa razy
Given	żądanie faktury z dwiema pozycjami
When	wywołanie metody issuance
Then	metoda obiektu TaxPolicy wywołana dwa razy z parametrami odpowiadającymi pozycjom faktury

Zaimplementuj powyższe przypadki testowe. Do definicji dublerów wykorzystaj framework izolujący Mockito. Opracuj dwa dodatkowe przypadki testowe, które będą testowały tak stan jak i zachowanie testowanego kodu.
Zmiany zatwierdzaj w gałęzi *issuance* (pamiętaj o jej zsynchronizowaniu z repozytorium na GitHub'ie).

1.2 AddProductCommandHandler

Znajdź klasę:

`pl.com.bottega.ecommerce.sales.application.api.handler.AddProductCommandHandler`

Opracuj przypadki testowe, które sprawdzą poprawność działania metody. Wykorzystaj możliwości związane z testowaniem stanu oraz zachowania.

2. Testowanie istniejącego kodu

Nie zawsze pisanie testów związane jest z opracowaniem nowego kodu. Często zdarza sie, że trzeba "pokryć" testami istniejący kod[1]. Kod ten moze mieć różną, nie zawsze najlepszą, strukturę. Funkcjonalnie kod działa, pojawia się jednak potrzeba dopisania do niego testów[2].
Rozwijając nowy kod możemy pisać go w taki sposób, aby ułatwić testowanie. W przypadku istniejącego kodu takiej możliwości nie mamy, co może mieć negatywny wpływ na proces testowania.
Sklonuj (operacja fork) repozytorium https://github.com/mto-lab/lab3_2 na swoje konto GitHub (a następnie na lokalny komputer). Zaimportuj projekt do IDE.
Napisz przypadki testowe dla scenariusza testowania metody loadNews klasy `edu.iis.mto.staticmock.NewsLoader`. Odizoluj test od klas `edu.iis.mto.staticmock.ConfigurationLoader` oraz `edu.iis.mto.staticmock.NewsReaderFactory`. Przeprowadź test stanu sprawdzający poprawność podziału wiadomości na publiczne oraz tylko dla sybskrybentów. Przeprowadź test zachowania sprawdzający poprawność wywołana wybranej zależności.
Do zasymulowania zależności wykorzystującej metody statyczne wykorzystaj framework PowerMock. Przykład użycia znajduje się w https://github.com/mto-lab/testisolation.
Zmiany zatwierdzaj w gałęzi o nazwie *legacy* (pamiętaj o jej zsynchronizowaniu z repozytorium na GitHub'ie).

3. Testowanie kodu wykorzystującego logikę upływu czasu - izolacja testu w kontekście zegara systemowego

Jednym z problemów w procesie testowania jest radzenie sobie z kodem, który do swojego działania wykorzystuje czas. Z założenia testy jednostkowe powinny wykonywać się szybko, tak aby zachęcać do jak najczęstszego ich wykonywania. Jeżeli testowany kod związany jest z upływem czasu wykonanie pewnych testów mogłoby zająć, w zależności od logiki przetwarzania, minuty, godziny czy nawet dni. Z punktu widzenia testów jednostkowych czas (zegar systemowy) jest zewnętrzną zależnością od którego powinniśmy móc się odizolować. Jednak tego typu izolacja wymaga odpowiedniej implementacji.
Sklonuj (operacja fork) repozytorium https://github.com/mto-lab/lab3_3 na swoje konto GitHub (a następnie na lokalny komputer). Zaimportuj projekt do IDE.
Korzystając z idei "Fake system clock"[1] zmodyfikuj kod w klasie edu.iis.mto.time.Order, w taki sposób aby można było przetestować poprawność działania logiki utraty ważności zamówienia po upływie okresu ważności (wywołanie metody confirm() powinno zakończyć się wyrzuceniem wyjątku typu *OrderExpiredException*).

---

[1] Taki kod określany jest mianem kodu spadkowego (ang. legacy code)
[2] "Po co?" - zapyta ktoś - "Przecież działa." Powodów może być wiele: chcemy zrozumieć jego działanie, chcemy przygotować się do refaktoryzacji, chcemy mieć wiekszą pewność, że na prawdę działa i będzie działał w nieoczekiwanych sytuacjach, itp.

Zmiany zatwierdzaj w gałęzi o nazwie *faketime* (pamiętaj o jej zsynchronizowaniu z repozytorium na GitHub'ie).

4. Jakość kodu testu

Wbrew pozorom jakość kodu testującego jest tak samo istotna jak jakość kodu produkcyjnego. Jeżeli już zdecydowaliśmy się na pisanie testów automatycznych to powinniśmy o nie dbać tak samo jak o kod produkcyjny [2].
Jednym z popularniejszych wzorców projektowych wykorzystywany w testowaniu jest Builder[3]. Wzorzec ten pozwala na odseparowanie konstrukcji złożonego obiektu od jego reprezentacji. Dzięki temu ten sam proces konstrukcji pozwala na tworzenie różnych reprezentacji. Wzorzec ten jest szczególnie użyteczny gdy konstrukcja obiektu wymaga podania wielu parametrów. Podejście wykorzystujące wzorzec Buildera pozwala klientowi utworzyć obiekt reprezentujący Builder za pomocą bezparametrowego konstruktora a następnie wykorzystując zestaw metod buildera określić wartości wybranych parametrów. Pozostałe parametry mogą przyjmować wartości domyślne. Ostatecznie klient wywołuje metodę build(), która w rezultacie zwraca obiekt docelowy. Oprócz tego wzorzec ten doskonale nadaje się do zbudowania kodu inicjalizującego dane dla potrzeb testu z wykorzystaniem stylu deklaratywnego.
Zmodyfikuj testy z zadania 1 tak aby inicjalizacja danych dla potrzeb testowania wykorzystywała wzorzec Test Builder. Rezultaty refaktoryzacji zatwierdzaj w gałęzi *builder*.

Przypisy

1. http://www.javapractices.com/topic/TopicAction.do?Id=234
2. Robert C. Martin: Czysty kod. Podręcznik dobrego programisty, rozdział 9 (Testy jednostkowe).
3. http://www.javacodegeeks.com/2013/06/builder-pattern-good-for-code-great-for-tests.html
4. http://www.martinfowler.com/bliki/TestDouble.html
5. http://www.javablog.eu/java/powermock-pomoc-w-testach-jednostkowych/
6. http://stackoverflow.com/questions/38181/when-should-i-mock