Tomek Kaczanowski

złe TESTY

dobre TESTY

Złe testy, dobre testy

Tomek Kaczanowski

Złe testy, dobre testy

Tomek Kaczanowski

Po więcej informacji zapraszam na http://practicalunittesting.com

Opublikowane przez Tomasz Kaczanowski kaczanowscy.pl

Okładkę zaprojektowała Agata Wajer-Gadecka, http://poleznaku.pl

ISBN: 978-83-938471-8-1

Wersja: print_20151026_2155

Dedykacja

Wam wszystkim, którzy pracujecie nad projektami open-source.
Niech Moc będzie z Wami!

Spis treści

Podziękowania

Ta książka nigdy nie ujrzałaby światła dziennego gdyby nie pomoc i zachęta ze strony wielu osób.

Bartosz Ocytko, Martin Skurla i Bartek Zdanowski byli uprzejmi podzielić się ze mną przykładami fatalnie napisanych testów (co nie oznacza, że byli autorami tychże!). Moi koledzy z zespołów, w których pracowałem, również dostarczyli wielu przykładów niedoskonałego kodu (jednak nie zawsze czynili to świadomie…).

Jakub Nabrdalik i Tomasz Borek dostarczyli trzy kompletne sekcje, które stanowią część tej książki.

Krzysztof Kozioł i Marcin Michalak przeczytali książkę we wczesnej fazie jej tworzenia i podzielili się ze mną wieloma cennymi uwagami.

Specjalne podziękowania składam na ręce Petri Kainulainena. Petri wykonał olbrzymią pracę recenzując kilkukrotnie kolejne wersje książki.
Nie tylko pomógł mi odkryć wiele słabych punktów i zaproponował szereg ulepszeń, ale i wskazywał mocne strony mojej pracy. Zachęcał mnie w ten sposób do dalszych starań i nie pozwalał osiąść na laurach. Dziękuję!

Peter Kofler - tak, sam Code Cop! - zgodził się napisać słowo wstępne do wersji angielskiej tej książki.

Carl Humphries namęczył się tłumacząc wersję angielską książki z mojego angielskiego na nieco bardziej angielski angielski. Następnie niejaki Tomek Kaczanowski przetłumaczył tę książkę z powrotem na język polski, dzięki czemu, Drogi Czytelniku, możesz dziś czytać o mockach w języku Reja, Kochanowskiego i Kazika.

Współudział w tworzeniu to nie współodpowiedzialność. Inni pomagali, inspirowali, poprawiali, ale za wszystkie błędy i niedoskonałości odpowiadam ja, i tylko ja!

Wstęp

Ah, więc chciałbyś pisać wspaniałe testy? Ja również. :) Tak się składa, że akurat w tej kwestii mam pewne doświadczenie i chętnie się nim z Tobą podzielę.

Proszę byś czytając tę książkę pozostał czujny. Porównuj to co czytasz z własnymi doświadczeniami, odnieś to do typu kodu z jakim pracujesz i wybierz to, co wyda Ci się właściwe i wartościowe. Nikt nie powiedział, że masz się zgadzać ze wszystkim co tu napisano. Nawet czułbym się nieswojo gdyby tak było.

Pomysł

Ta książka opiera się na prostym pomyśle: prezentuję krótkie kawałki kodu testowego i omawiam sposoby ich poprawienia.

Wszystkie przykłady przedstawione w ksiażce są prawdziwe, tj. pochodzą z prawdziwych aplikacji. Nic nie zmyśliłem. Są prawdziwe i kropka. Z tysięcy testów, jakie miałem okazje widzieć, wybrałem te, które ilustrują pewne typowe błędy lub niedoskonałości. Zazwyczaj natrafiałem na nie robiąc przeglądy kodu (a czasem trafiałem na nie przeglądając mój własny kod!) i na tyle zalazły mi za skórę, że postanowiłem je tu opisać.

Niektóre z prezentowanych fragmentów kodu zostały z lekka *zobfuskowane*. Zabieg ten stosowałem przede wszystkim by uniemożliwić identyfikacje projektów, z których je pożyczyłem.
Wiele kawałków kodu zostało znacznie skróconych i uproszczonych w taki sposób, że przetrwała tylko ich *esencja*, tylko ten pojedynczy aspekt, na którym w danym przykładzie chciałem skupić uwagę. W ten sposób przykłady stały się czytelniejsze i łatwe do zrozumienia. Z drugiej strony, patrząc na tak niewielkie fragmenty kodu trudniej jest uświadomić sobie wagę opisywanych problemów. W końcu zrozumienie kilku linijek nawet słabego kodu nie jest wyzwaniem. Jednak zapewniam, że opisywane problemy **są istotne** w prawdziwych, życiowych scenariuszach, gdzie ilość kodu jest o wiele większa, jest on o wiele bardziej skomplikowany, a podobne konstrukcje występują w wielu testach.

SUT i asercje

Od czasu do czasu będę używał terminu *SUT* (ang. *System Under Test*), czyli to **coś** co właśnie testujemy. SUT może być bardzo różnej wielkości. W przypadku testów jednostkowych jest to zazwyczaj pojedyncza klasa. W przypadku testów integracyjnych może być to klasa, ale równie dobrze warstwa lub moduł. W przypadku testów end-to-end jest to zazwyczaj cały system.

W kodzie prezentowanym w tej książce używam asercji dostarczonych przez projekt AssertJ[1] (`assertThat(...)`) zamiast tych dostarczanych przez frameworki testowe takie jak JUnit lub TestNG (np. `assertEquals(...)`). Po napisaniu wielu tysięcy testów każdego rodzaju jestem przekonany, że takie właśnie asercje zwiększają czytelność kodu i pozwalają mi lepiej wyrazić intencję.

Ikony

W tekście pojawiają się następujące ikony;

Pomocna wskazówka.

Dodatkowa informacja.

Uważaj, niebezpieczeństwo jest blisko!

Wszystkie trzy ikony zostały zaprojektowane przez Webdesigner Depot http://www.webdesignerdepot.com/.

Fonty

Do prezentacji kodu wykorzystuję font Oxygen Mono stworzony przez Vernona Adamsa http://code.newtypography.co.uk/.

Czego oczekuję od Ciebie

By w pełni skorzystać z tej książki byłoby wyśmienicie gdybyś:

- miał pewne doświadczenie w pisaniu testów, przede wszystkim testów jednostkowych,
- znał frameworki testowe; najlepiej JUnit albo TestNG,
- chciał poprawić swoje umiejętności pisania testów,

[1] http://joel-costigliola.github.io/assertj/

- rozumiał, że przejrzysty kod jest wielką wartością.

Jeżeli chciałbyś zwiększyć swoją wiedzę w powyższych tematach polecam lekturę książek poświęconych testom, które znajdziesz na stronie http://practicalunittesting.com.

Dobre rady

Ta książka zawiera wiele porad, np.

Nigdy nie używaj metody `System.currentTimeMillis()` ale użyj dodatkowego interfejsu, który później możesz bez problemu podmienić w kodzie testowym.

Tego typu rady mają to do siebie, że są prawdziwe w 99% przypadków. I dlatego właśnie zdecydowałem się je tu zamieścić. Jednak napotkasz w życiu takie sytuacje, w których **nie powinieneś się do nich stosować**. Więc miej je w pamięci ale nie podążaj za nimi na oślep.

Dlaczego ta książka jest darmowa?

Jest darmowa, bo tak postanowiłem. :) To przede wszystkim zasługa tych z Was, którzy kupili moje poprzednie książki z serii *Practical Unit Testing*. Bardzo dziękuję!

W przeciągu całej mojej "kariery" w IT wiele razy udało mi się rozwiązać jakiś problem korzystając z wpisów na blogach czy też forach internetowych. Wielokrotnie też korzystałem z kodu open-source. Nie wyobrażam sobie programowania bez korzystania z wiedzy i doświadczeń innych programistów. Rozdając tą książkę za darmo chciałbym spłacić mój dług w stosunku do społeczności deweloperów. Dziękuję Wam wszystkim!

A poza tym wszystkim, to po prostu miło jest dawać prezenty. Więc bardzo się cieszę, że mam taką możliwość! :)

Strona WWW

Zapraszam na stronę książki: http://practicalunittesting.com.

O autorze

Tomek Kaczanowski na co dzień pracuje jako programista i lider zespołu. Interesuje go wysokiej jakości kod, testy i automatyzacja (a najchętniej wszystkie trzy naraz). Odkąd zorientował się, że nie potrafi poprawnie napisać nawet linijki kodu, rzucił się na testy upatrując w nich nadziei i wybawienia. Stuprocentowe wybawienie nie nastąpiło, ale przynajmniej jest w stanie znaleźć sporo błędów nim trafią na produkcję, a to już coś! Jego uzależnienie od pisania testów przejawia się w postaci silnej alergii na widok pustego katalogu `src/test/java`.
Tomek wierzy, że poprawiając kod poprawia świat (przynajmniej odrobinę). Ku jego rozczarowaniu, świat wydaje się nie zwracać na to żadnej uwagi.

Oprócz umiejętności programowania Tomek posiada też pewne tzw. umiejętności miękkie, co w połączeniu z wysokim mniemaniem o sobie pozwala mu występować na konferencjach, pisać artykuły (ba, nawet książki!), i zgrywać mentora wobec młodszych kolegów po fachu. Słowem, pouczać innych na różne sposoby. Oczywiście wszystko z pięknych, altruistycznych pobudek! Tomek przekonany jest bowiem, że zły kod wraca do autora niczym zdradziecki bumerang, by w najmniej spodziewanym momencie huknąć go znienacka w głowę. Uważa więc za swój obowiązek ostrzegać innych przed tym okropnym niebezpieczeństwem.

Pomimo tych wszystkich dziwactw, Tomek jest w sumie dość normalną osobą: mężem, ojcem trójki dzieci i właścicielem kota.

Rozdział 1. Po cóż w ogóle się trudzić?

Nie jest moim celem wyjaśniać całej teorii testów ani rozpisywać się nad zaletami płynącymi z ich posiadania. Dla potrzeb tej książki wystarczy, że przypomnimy sobie tylko trzy podstawowe powody, dla których warto je pisać. Mam nadzieję, że to uproszczenie zostanie mi wybaczone. A oto i trzy obiecane powody:

- Mając kompletny zestaw testów możemy ze sporą dozą pewności powiedzieć, że nasza aplikacja działa.
- Dzięki testom wprowadzanie zmian jest łatwiejsze. Poinformują nas, jeżeli popsuliśmy coś, co dotychczas działało.
- Testy są prawdopodobnie jedyną zawsze aktualną formą dokumentacji jaką posiadamy.

Warto pamiętać o tych trzech powodach pisania testów gdy w dalszych częściach książki zajmiemy się analizą różnych **źle napisanych** testów. Warto zadawać pytania: *"Czy to w ogóle coś testuje?"*, *"A co z utrzymaniem tego testu w sytuacji zmian w kodzie produkcyjnym?"*, *"Czy łatwo zrozumieć o co chodzi w tym teście?"* i tym podobne.

Bardzo martwią mnie przypadki osób, które postanowiły włączyć pisanie testów do swoich praktyk deweloperskich… i w końcowym rezultacie mają tylko więcej słabej jakości kodu do utrzymania! Jak to jest możliwe? Pewnie tak to się kończy gdy dysponuje się tylko powierzchowną wiedzą na temat pisania testów. Ciężko w ten sposób napisać kod testowy, który będzie wartościowym dodatkiem a nie dodatkowym obciążeniem.

Cokolwiek robisz, rób to dobrze. Jeżeli warto coś robić, to warto to zrobić dobrze. Pamiętaj o tych zasadach pisząc testy!

Im więcej kodu piszesz, tym więcej kodu masz do utrzymania. Jest to prawdą również w odniesieniu do testów!

Rozdział 2. Łamiąc wszelkie zasady

> Brejkam wszystkie rule
>
> — Świetliki

Naszą przygodę rozpoczniemy od analizy kilku naprawdę beznadziejnych testów. Niektóre z nich zostały napisane ponad 10 lat temu, czyli w czasach, kiedy wielu z nas w ogóle nie miało nawet pojęcia o tym, że testy należy pisać. Żywiąc więc prawdziwy szacunek dla wysiłków podejmowanych przez (nieznanych mi) autorów tychże testów, dziś możemy już z całym przekonaniem powiedzieć, że napisany przez nich kod jest po prostu straszny.

2.1. Bez asercji

Zaprezentowane poniżej kilka linijek kodu skopiowałem z pewnego bardzo starego i okropnego (naprawdę okropnego!) testu. Wiele można by napisać o jego ohydzie, ale najbardziej kłuł w oczy fakt, że nie było w nim żadnych asercji. Żadnych. Zero. Null. Zamiast tego test wypisywał sporo różnorodnych informacji na `System.out`. Deweloper mógł je przeanalizować by stwierdzić czy testowana funkcjonalność działa poprawnie czy nie.

```
IResult result = format.execute();
System.out.println(result.size());
Iterator iter = result.iterator();
while (iter.hasNext()) {
    IResult r = (IResult) iter.next();
    System.out.println(r.getMessage());
}
```

Z oczywistych względów nie jest to właściwe podejście. Poprawnie napisany test nigdy nie powinien nas zmuszać do wykonania jakiejkolwiek manualnej weryfikacji. Żadnego śledzenia logów, żadnego odpytywania bazy danych. Nic z tych rzeczy. Wszystko powinno dziać się automatycznie. Dobrze wychowany test po prostu pada kiedy testowana przez niego funkcjonalność nie działa jak trzeba.

Poniżej przedstawiam poprawioną wersję:

```
IResult result = format.execute();
assertThat(result.size()).isEqualTo(3); ❶
Iterator iter = result.iterator();
while (iter.hasNext()) {
    IResult r = (IResult) iter.next();
    assertThat(r.getMessage()).contains("error"); ❷
}
```

❶❷ Rzecz jasna akurat w tym przypadku nie mam pojęcia jakiego wyniku należy się spodziewać, więc wartości w assercjach wyssałem z palca.

Ta wersja testu nie wymaga od dewelopera wykonywania jakichkolwiek dodatkowych czynności. To framework testujący powiadomi nas o wyniku testu.

Jest tylko jeden powód pisania testów bez asercji (i jest to powód wynikający z niskich pobudek). Mianowicie taki test, choć nic nie weryfikuje, to jednak podwyższa stopień pokrycia kodu testami. Paskudztwo!

2.2. Generatory testów

Idea jest w zasadzie genialna. Skoro testy jednostkowe są tak proste, to może by je tak wygenerować? No przecież taki test jednostkowy to co on niby robi? Ano tworzy obiekt, potem wywołuje jego metody, a na końcu sprawdza rezultat. Da się takie coś napisać automatycznie? A pewnie, że się da! Dzięki temu nie tylko zaoszczędzimy mnóstwo czasu ale jeszcze dostaniemy 100% pokrycia kodu. Na co więc czekać?!

Cóż, może brzmi to zachęcająco, ale niestety nie działa. Wystarczy zadać sobie pytanie po co tak właściwie piszemy testy by zrozumieć, że automatyczne ich generowanie nie trafia w sedno naszych bolączek. Taki test nie wykryje żadnego ciekawego błędu ani nie pomoże ulepszyć designu. Co więcej stosowanie generatorów testów promuje podejście *"najpierw kod potem testy"* i łamie zasadę *"testuj zachowanie a nie metody"* (patrz sekcja 4.6.2). Jego wartość dokumentacyjna również jest nikła. Jak widzisz naprawdę nie ma sensu zwalać zadania pisania testów na automat.

A teraz przykład czym kończy się taka próba. W poniższym kodzie widać, że automat zdołał wygenerować testy dla getterów i setterów, co, szczerze powiedziawszy, jest stratą czasu.

```
public void testSetGetTimestamp() throws Exception {
    // JUnitDoclet begin method setTimestamp getTimestamp
    java.util.Calendar[] tests = {new GregorianCalendar(), null};

    for (int i = 0; i < tests.length; i++) {
        adapter.setTimestamp(tests[i]);
        assertEquals(tests[i], adapter.getTimestamp());
    }
    // JUnitDoclet end method setTimestamp getTimestamp
}

public void testSetGetParam() throws Exception {
    // JUnitDoclet begin method setParam getParam
    String[] tests = {"a", "aaa", "---", "23121313", "", null};
```

```
    for (int i = 0; i < tests.length; i++) {
        adapter.setParam(tests[i]);
        assertEquals(tests[i], adapter.getParam());
    }
    // JUnitDoclet end method setParam getParam
}
```

Niby jakież to błędy spodziewasz się odkryć pisząc, o przepraszam!, autogenerując takie testy?

A teraz, powtórzmy wszyscy razem: *"Nie będę autogenerował testow ani nawet ich szkieletu. Nie będę autogenerował testów ani nawet ich szkieletów. Nie będę generował testów ani nawet ich szkieletu."* (Ta mantra jest również bardzo pomocna dla osób, które mają problem z zaśnięciem.)

Nie używaj autogeneratorów kodu testowego. Automatyczne tworzenie szkieletu testów (na co pozwalają niektóre IDE) też jest złe.

2.3. Albo cały czas, albo szkoda zachodu

Z pisaniem testów jest już tak, że albo o nie dbamy, albo nie. Albo piszemy je codziennie, albo się samooszukujemy wmawiając sobie, że lada moment przydzie taki cudowny czas kiedy naprawdę weźmiemy się za te testy. Co ciekawe, to przekonanie w nas nie ginie, mimo że kolejne tygodnie i miesiące pokazuje nam, że ów mityczny czas nigdy nie nadchodzi.

Przyjrzyjmy się teraz kawałkowi kodu pochodzącemu z klasy testowej `SystemAdminSmokeTest`. Jak nazwa klasy wskazuje jest to **smoke test**[1]. Skoro tak, to pewnie jest dość istotny, bo pozwala szybko stwierdzić czy pewne część systemu (prawdopodobnie moduł administracyjny) działa. Niestety w momencie gdy dołączyłem do projektu, do którego należał ten test, wyglądał on w sposób przedstawiony poniżej. Zwróć proszę uwagę na wykomentowane linie!

```
class SystemAdminSmokeTest extends GroovyTestCase {

void testSmoke() {
// do not remove below code
// def ds = new org.h2.jdbcx.JdbcDataSource(
//     URL: 'jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;MODE=Oracle',
//     user: 'sa', password: '')
//
//     def jpaProperties = new Properties()
//     jpaProperties.setProperty(
//          'hibernate.cache.use_second_level_cache', 'false')
//     jpaProperties.setProperty(
```

[1] https://en.wikipedia.org/wiki/Smoke_testing#Software_development

```
//          'hibernate.cache.use_query_cache', 'false')
//
//     def emf = new LocalContainerEntityManagerFactoryBean(
//        dataSource: ds, persistenceUnitName: 'my-domain',
//        jpaVendorAdapter: new HibernateJpaVendorAdapter(
//            database: Database.H2,  showSql: true,
//            generateDdl: true), jpaProperties: jpaProperties)

   ...
}
```

Cóż, ten test z pewnością nie poinformuje nikogo o stanie systemu… A przecież pójdę o zakład, że kiedyś ten test działał. Ale potem najwyraźniej nastąpiły zmiany i test został wykomentowany. Jestem przekonany, że osoba, która to zrobiła, była przekonana, że to *"tylko na te kilka dni"*. Niestety, te kilka dni trwało latami.

Usuwaj testy, które nie są już potrzebne. Po co je trzymać?

No dobrze, ale...

Gdy przeprowadzamy redesign systemu zdarza się, że naprawdę chcemy wyłączyć na pewien czas wybrane testy. Czasami po prostu ma to sens.
Zawsze wiąże się z tym niebezpieczeństwo, że testy te pozostaną już wyłączone.

Zazwyczaj radzę sobie z tym problemem dodając komentarz do ignorowanego testu. W komentarzu umieszczam numer zadania z issue trackera. Gdy kończe pracę nad zadaniem zawsze przeszukuję kod, czy nie zostały tam jeszcze jakieś komentarze oznaczone tym właśnie numerem. A nawet gdybym zapomniał to zrobić to pewnie zrobi to mój kolega podczas przeglądu kodu (robisz przeglądy kodu, prawda?). Prawdopodobieństwo, że obydwoje przeoczymy wykomentowany test, jest znikome.

Kolejny przykład jest jeszcze bardziej przygnębiający. Oto w katalogu `src/test/java` pewnego projektu odnalazłem taki oto test:

```
@Test
public class ExampleTest {

    public void testExample() {
        assertTrue(true);
    }
}
```

Pewnie domyślasz się już dalszego ciągu tej historii. Tak, ten test był tam bardzo samotny.

2.4. Wnioski

Poznaliśmy do tej pory kilka bardzo marnych testów, ale i z nich możemy wyciągnąć pewną naukę:

- Warto naprawiać rzeczy od razu. Teoria rozbitych okien[2] mówi, że bałagan rodzi jeszcze większy bałagan.
- Z pisaniem i utrzymywaniem testów związany jest pewien wysiłek. I to wysiłek całego zespołu. Co więcej, potrzebne jest wsparcie (a przynajmniej brak obstrukcji) ze strony wszystkich osób zaangażowanych w projekt.
- *Dobrymi chęciami piekło brukowano*. By osiągnąć upragniony cel jakim jest dobrze przetestowany system potrzebujesz wiedzy, czasu i determinacji.
- Weryfikacja wyników testu musi następować automatycznie.
- Testy trzeba pisać, a nie generować je.
- Pisanie testów to zadanie na każdy dzień. I pamiętaj, nikt nie zrobi tego za Ciebie.
- Skoro nie piszesz testów, to nigdy nie nauczysz się ich pisać.

A teraz uzbrojeni w te podstawowe wiadomości możemy przejść ku nieco bardziej interesującym przykładom.

[2] Patrz https://pl.wikipedia.org/wiki/Teoria_rozbitych_okien

Rozdział 3. Siła testów

Pierwszym powodem, dla którego zazwyczaj zaczynamy pisać test, jest chęć *"sprawdzenia czy to działa"*. To nie wydaje się zbyt trudne, prawda? Po prostu wymyślasz przypadek testowy, zapisujesz go w postaci kodu przy pomocy JUnit, TestNG czy Spocka i po krzyku. Niestety, nie jest to aż tak trywialne.

W tym rozdziale skoncentrujemy się na tym właśnie aspekcie testów. Zbadamy czy naprawdę testują cokolwiek istotnego. I jak się przekonamy, czasami zdarza się im zawieść nawet w tym kluczowym przecież aspekcie. Niektóre testy z pewnością *coś* testują, ale być może nie całkiem to, co byłoby godne przetestowania. Spotkamy też takie testy, które nie testują zupełnie niczego!

3.1. Znajdź dobre wartości do testów

> małe liczby - niepewność
> wielkie liczby - pomyślność
>
> — Sennik: liczby

Sytuacja, w której test przechodzi, mimo że funkcjonalność, którą chcieliśmy przetestować, nie działa, jest więcej niż niepokojąca. Może się zdarzyć, że sami pakujemy się w tego typu kłopoty niefrasobliwie dobierając przypadki testowe. Przyjrzymy się teraz dwóm testom, w których zachodzi takie właśnie ryzyko.

3.1.1. 20, 50 i 50?

Przyjrzyjmy się następującej metodzie fabryki obiektów:

```
public PriceCalculator create() {
    BigDecimal minMargin = settings.getMinMargin();
    BigDecimal maxMargin = settings.getMaxMargin();
    BigDecimal premiumShare = settings.getPremiumShare();
    return new PriceCalculator(minMargin, maxMargin, premiumShare);
}
```

`settings` jest współpracownikiem, nic więc dziwnego, że w teście używamy stuba zamiast oryginalnego obiektu.

```
public class PriceCalculatorFactoryTest {
    SettingsService settings = mock(SettingsService.class);

    @Test
```

```
    public void shouldCreatePriceCalculator() {
        //given
        given(settings.getMinMargin()).willReturn(new BigDecimal(20));
        given(settings.getMaxMargin()).willReturn(new BigDecimal(50));
        given(settings.getPremiumShare()).willReturn(new BigDecimal(50));

        //when
        PriceCalculator calculator
            = new PriceCalculatorFactory(settings).create();

        //then
        assertThat(calculator)
            .isEqualTo(new PriceCalculator(new BigDecimal(20),
            new BigDecimal(50), new BigDecimal(50)));
    }
}
```

Wartości jakich użyto w tym teście (`minMargin` = 20, `maxMargin` = 50, `premiumShare` = 50) powinny wzbudzić nasze podejrzenia. Być może biznesowo mają one sens, a jednak stanowią słaby przypadek testowy.
Wyobraźmy sobie, że testowana metoda działa niepoprawnie i przypisuje wartości nie do tych pól co powinna (co może się zdarzyć gdy ktoś tworzył ją metodą "copy&paste"). Jeżeli parameter `maxMargin` użyto w miejscu `premiumShare` (i na odwrót) wówczas test przejdzie, bo oba pola otrzymają tę samą wartość 50.

Aby zminimalizować ryzyko powinniśmy użyć różnych wartości dla każdego z parametrów. Kolejny fragment kodu przedstawia test po takiej właśnie zmianie (dodatkowo pozwoliłem sobie opisać liczby w postaci pól `static final`):

```
public class PriceCalculatorFactoryTest {
    SettingsService settings = mock(SettingsService.class);

    private static final BigDecimal MIN_MARGIN = new BigDecimal(20);
    private static final BigDecimal MAX_MARGIN = new BigDecimal(30);
    private static final BigDecimal PREMIUM_SHARE = new BigDecimal(40);

    @Test
    public void shouldCreatePriceCalculator() {
        //given
        given(settings.getMinMargin()).willReturn(MIN_MARGIN);
        given(settings.getMaxMargin()).willReturn(MAX_MARGIN);
        given(settings.getPremiumShare()).willReturn(PREMIUM_SHARE);

        //when
        PriceCalculator calculator
            = new PriceCalculatorFactory(settings).create();

        //then
```

```
        assertThat(calculator).isEqualTo(
            new PriceCalculator(MIN_MARGIN, MAX_MARGIN, PREMIUM_SHARE));
    }
}
```

Po tej zmianie test wykryje przypadek niewłaściwego przypisywania parametrów.

W metodach testowych unikaj używania tych samych wartości (liczb, tekstów) dla różnych zmiennych. Niezłym pomysłem jest używanie kolejnych potęg liczby 2 (1, 2, 4, 8, … itd.). Ich zaletą jest to, że nie sumują się do siebie nawzajem.

3.1.2. Unikaj zer

Wybierając przypadki testowe powinniśmy bardzo starannie dobierać dane wejściowe. Poniższy przykład ilustruje dlaczego jest to tak istotne.

```
public class PaymentServiceTest {

    PaymentAdapter paymentAdapter = mock(PaymentAdapter.class);
    PaymentService paymentService = new PaymentService(paymentAdapter);

    @Test
    public void shouldReturnRevenueForClient() {
        //given
        Client client = new Client();
        given(paymentAdapter
            .getRevenue(client, PaymentService.REPORT_COUNT))
            .willReturn(0d);

        //when
        double actual = paymentService.getRevenue(client);

        //then
        assertThat(actual).isEqualTo(0d);
    }
}
```

Na pierwszy rzut oka nie wydaje się by ten test miał jakąkolwiek wadę. Jest bardzo czytelny, prosty, skupiony na jednej funkcjonalności.

…ale ktoś dociekliwy (jak ja) mógłby zakwestionować użycie zera. Nie żebym miał coś do zer, są w porządku, ale… Chodzi o to, że niektóre wartości - w szczególności `0` i `null` - są często zwracane przez metody wygenerowane automatycznie przez IDE. Więc istnieje pewne szansa, że taki test przejdzie nawet jeżeli funkcjonalność tak naprawdę nie została jeszcze zaimplementowana.

Wydaje się, że bezpieczniej byłoby użyć dowolnej inne liczby. Na przykład tak:

```
@Test
public void shouldReturnRevenueForClient() {
    //given
    Client client = new Client();
    given(paymentAdapter
            .getRevenue(client, PaymentService.REPORT_COUNT))
            .willReturn(1.23);

    //when
    double actual = paymentService.getRevenue(client);

    //then
    assertThat(actual).isEqualTo(1.23);
}
```

To wcale nie oznacza, że nie należy pisać testów weryfikujących zachowanie metod dla wartości `null` i `0`! Wprost przeciwnie, takie przypadki testowe prowadzą często do wykrycia ciekawych błędów. Chodzi o to, żeby nie ograniczyć się do pojedynczego testu, w którym użyjemy tylko `0` albo `null`.

3.2. Selfie

W pewnej aplikacji powstała konieczność zaimplementowania wielu typów płatności. Poszczególne typy płatności były akceptowane w różnych krajach. Ich lista przechowywana była w enumie `PaymentMethod`.

Przedstawiony poniżej test sprawdza czy metoda `getMethodsForCountry()` enuma `PaymentMethod` zwraca poprawne metody płatności dla Polski. W sekcji *"given"* tworzy on listę oczekiwanych metod płatności, w *"when"* wywołuje testowaną metodę, a w sekcji *"then"* dokonuje porównania oczekiwanego wyniku z faktycznie uzyskanym.

```
@Test
public void shouldGetMethodsForPoland() {
    //given
    List<PaymentMethod> all = Lists.newArrayList(PaymentMethod.values());
    List<PaymentMethod> methodsAvailableInPoland = Lists.newArrayList();
    for (PaymentMethod method : all) {
        if (method.isEligibleForCountry("PL")) {
            methodsAvailableInPoland.add(method);
        }
    }

    //when
    List<PaymentMethod> methodsForCountry = PaymentMethod
```

```
        .getMethodsForCountry("PL", all);

    //then
    assertThat(methodsForCountry).isEqualTo(methodsAvailableInPoland);
}
```

Na pierwszy rzut oka wszystko gra. A jednak można zauważyć coś niepokojącego w sekcji *"given"*. Używa ona dwóch metod należących do testowanego enuma `PaymentMethod`: `values()` i `isEligibleForCountry()`). Korzystając z nich przygotowuje oczekiwany rezultat.
A teraz spójrzmy na kod metody `getMethodsForCountry()`. Niestety, podejrzenia okazują się uzasadnione.

```
public enum PaymentMethod {

    public static List<PaymentMethod> getMethodsForCountry(
            Country country, List<PaymentMethod> availableMethods) {
        List<PaymentMethod> methodsForCountry = Lists.newArrayList();
        for (PaymentMethod method : availableMethods) {
            if (method.isEligibleForCountry(country.getCode())) {
                methodsForCountry.add(method);
            }
        }
        return methodsForCountry;
    }
}
```

Jak widać używaliśmy identycznego kodu do stworzenia obu wyników - oczekiwanego i faktycznego. No, w ten sposób raczej trudno spodziewać się, że kiedykolwiek będą się one różniły!
Rozwiązaniem byłaby zmiana sposobu w jaki przygotowujemy oczekiwany rezultat. Na przykład na taki:

```
@Test
public void shouldGetMethodsForPoland() {
    //given
    List<PaymentMethod> all = Lists.newArrayList(PaymentMethod.values());
    List<PaymentMethod> methodsAvailableInPoland = Arrays.asList(
        new PaymentMethod[] {
            PaymentMethod.MASTERCARD,
            PaymentMethod.VISA,
            ...
            // and all other methods available in Poland
        });

    //when
    List<PaymentMethod> methodsForCountry = PaymentMethod
        .getMethodsForCountry("PL", all);
```

```
    //then
    assertThat(methodsForCountry).isEqualTo(methodsAvailableInPoland);
}
```

No, teraz lepiej! Test nie używa już logiki należącej do enuma `PaymentMethod` do przygotowania oczekiwanego rezultatu.

3.3. Happy Path

Wydaje mi się, że pisanie testów typu *"happy path"* to najpopularniejszy antywzorzec jaki zdarza się stosować tworząc testy.

Ale co to właściwie znaczy? Any *happy path* to taki najprostszy, najbardziej oczywisty przypadek testowy. To jak testować, że kalkulator potrafi dodać dwa do dwóch, albo, że jeżeli dodasz użytkownika *Jan Kowalski* do bazy danych, to potem możesz go w niej znaleźć.

Ważne by pamiętać o jednej sprawie. Takie testy **nie są złe**! Wprost przeciwnie, one **są niezbędne**. Dlaczego? Bo pokrywają scenariusze, które są najbardziej typowe dla pisanej aplikacji. Oprogramowanie, które nie spełnia nawet tak podstawowych wymogów, jest zupełnie bezużyteczne. Tak więc problemem nie jest pisanie takich testów. Problemem jest gdy są to jedyne testy jakie istnieją. Wówczas kłopoty dosięgną nas bardzo prędko. Zaraz okaże się, że kalkulator nie działa gdy jedna z liczb jest ujemna, albo że baza danych nie jest gotowa przyjąć danych osoby o imieniu dłuższym niż 20 znaków. Klienci z pewnością szybko nas o takich kwiatkach powiadomią.

3.3.1. FizzBuzz

> Napisz program, który wypisze liczby od 1 do 100. Ale dla wielokrotności trójki niech wypisze "Fizz" zamiast liczby. A dla wielokrotności piątk niech wypisze "Buzz". Dla liczb będącyh wielokrotnością zarówno trójki i piątki niech wypisze "FizzBuzz".
>
> — FizzBuzz RosettaCode

Przyjrzyjmy się temu testowi klasy `FizzBuzz`, która iplementuje podany powyżej algorytm [1].

```
public class FizzBuzzTest {
    @Test
    public void testMultipleOfThreeAndFivePrintsFizzBuzz() {
        assertEquals("FizzBuzz", FizzBuzz.getResult(15));
    }
```

[1] Poniższy kod znalazłem na forum dyskusyjnym CodeReview, http://codereview.stackexchange.com/questions/9749

```
    @Test
    public void testMultipleOfThreeOnlyPrintsFizz() {
        assertEquals("Fizz", FizzBuzz.getResult(93));
    }

    @Test
    public void testMultipleOfFiveOnlyPrintsBuzz() {
        assertEquals("Buzz", FizzBuzz.getResult(10));
    }

    @Test
    public void testInputOfEightPrintsTheNumber() {
        assertEquals("8", FizzBuzz.getResult(8));
    }
}
```

Wygląda całkiem nieźle. Każde wymaganie jest pokryte przez osobną metodę testową. Każde jest weryfikowane przy pomocy jednego przypadku testowego.

A jednak upierałbym się, że wciąż jest to test typu *happy path*. Niepokoi mnie skromna ilość przypadków testowych. Moim zdaniem jest ich zbyt mało by móc z przekonaniem powiedzieć, że klasa `FizzBuzz` działa poprawnie dla wszystkich liczb z zakresu od 1 do 100. Mógłbym nawet dostarczyć (złośliwą) implementację, która bez problemu przeszła by powyższy test, a która w rzeczywistości nie spełniałaby wymagań gry.

Nie twierdzę, że powinniśmy przetestować poprawność działania algorytmu dla każdej liczby (choć nie byłoby to specjalnie trudne biorąc pod uwagę, że jest ich tylko 100). Wydaje mi się, że bylibyśmy bezpieczniejsi, gdyby każde z wymagań zweryfikować przy pomocy kilku wartości. Okazuje się, że aby wprowadzić taką zmianę do kodu testowego nie musimy nawet pisać nowych metod. Wystarczy tylko istniejącym dostarczyć większej ilości danych. Kolejny listing demonstruje jak można to zrobić korzystając z JUnit.

Używasz JUnit? Skorzystaj z projektu JUnitParams[2] - biblioteki rozszerzającej możliwości JUnita w zakresie pisania testów parametryzowanych.

```
@RunWith(JUnitParamsRunner.class)
public class FizzBuzzJUnitTest {

    @Test
    @Parameters(value = {"15", "30", "75"})
    public void testMultipleOfThreeAndFivePrintsFizzBuzz(
```

[2]https://github.com/Pragmatists/junitparams

```
            int multiple0f3And5) {
        assertEquals("FizzBuzz", FizzBuzz.getResult(multiple0f3And5));
    }

    @Test
    @Parameters(value = {"9", "36", "81"})
    public void testMultiple0fThree0nlyPrintsFizz(int multiple0f3) {
        assertEquals("Fizz", FizzBuzz.getResult(multiple0f3));
    }

    @Test
    @Parameters(value = {"10", "55", "100"})
    public void testMultiple0fFive0nlyPrintsBuzz(int multiple0f5) {
        assertEquals("Buzz", FizzBuzz.getResult(multiple0f5));
    }

    @Test
    @Parameters(value = {"2", "16", "23", "47", "52", "56", "67", "68", "98"})
    public void testInput0fEightPrintsTheNumber(int expectedNumber) {
        assertEquals("" + expectedNumber,
                FizzBuzz.getResult(expectedNumber));
    }
}
```

Tak napisany test z pewnością jest silniejszy od poprzednika. Każde wymaganie sprawdzane jest teraz nie przez jedno, ale przez kilka przypadków testowych.

Zagraj w BizzBuzz z przyjaciółmi. Instrukcję znajdziesz na http://en.wikipedia.org/wiki/Bizz_buzz :)

3.3.2. Ciągle za mało testów

Działo się to dawno temu, w czasach gdy trzeba było pisać sporo własnego kodu by obsłużyć typowe zadania, które dziś wykonują popularne frameworki. Otóż w tych oto starożytnych czasach pracowałem nad aplikacją webową. Aplikacja ta, jak to aplikacje tego typu mają w zwyczaju, prezentowała pewne dane w postaci listy. Moim zadaniem było zaimplementowanie stronicowania. Chodziło o taki oto element strony, którego klikanie skutkowałoby wyświetlaniem kolejnych stron z wynikami:

```
<< < 1 2 3 ... 99 > >>
```

Dane wejściowe obejmowały numer aktualnej strony, ilość elementów do wyświetlenia na każdej ze stron, oraz całkowitą ilość elementów. Na ich podstawie projektowany komponent miał podać:

- które numery stron należy wypisać (np. 1, 2, 3 i 99),
- jaki jest offset poszczególnych stron (by można było wygenerować odpowiednie linki),
- które strony nie powinny być linkami (w szczególności strona bieżąca).

Zadanie nie jest zbyt skomplikowane, ale sporo w nim sytuacji brzegowych do rozpatrzenia i o błąd nietrudno[3]. Zdecydowałem więc, że warto napisać testy jednostkowe. Powstało ich całkiem sporo. Ten przedstawiony poniżej związany jest z logiką przycisku "next" (czyli znaku strzałki >, naciśnięcie której powinno przenosić użytkownika na kolejną stronę).

```
@Test
public class PagerTest {

    private static final int PER_PAGE = 10;

    public void shouldGiveOffsetZeroWhenOnZeroPage() {
        Pager pager = new Pager(PER_PAGE);

        assertThat(pager.getOffset()).isEqualTo(0);
    }

    public void shouldIncreaseOffsetWhenGoingToPageOne() {
        Pager pager = new Pager(PER_PAGE);

        pager.goToNextPage();

        assertThat(pager.getOffset()).isEqualTo(PER_PAGE);
    }
}
```

Obie metody testowe sprawdzają czy metoda zwraca właściwy `offset`. Ta wartość w kodzie produkcyjnym jest wykorzystywana do skonstruowania odpowiedniego zapytania do bazy danych by wydobyć z niej wyniki należące do kolejnej strony. Powyższe testy są całkiem rozsądne: zakładając, że na każdej stronie ma być wyświetlonych 10 elementów (`PER_PAGE = 10`), na stronie o numerze 0 offest powinien wynosić 0, a na stronie o numerze 1 powinien on wynosić 10. Wszystko gra.

Tak więc dumny i blady, że wszystko na pewno będzie pięknie działać (bo przecież wszystkie testy przechodzą) uruchomiłem moja aplikację. I tu niespodzianka. Ku mojemu bezgranicznemu zdziwieniu przechodzenie po kolejnych stronach nie działało. Dało się przejść ze strony o numerze 0 na stronę o numerze 1, ale już kolejne kliknięcia na strzałkę > nie powodowały żadnych zmian. To jakaś bzdura - przecież wiedziałem, że to działa! No, nawet to przetestowałem!

[3]Patrz http://en.wikipedia.org/wiki/Off-by-one_error.

Pierwszą rzeczą jaką należy zrobić w takiej sytuacji to uciszyć ten głupi głos w głowie, który krzyczy *"to działa! to działa! przecież to działa!"*. Dopóki tego nie zrobisz, nie znajdziesz błędu. Więc ucisz go jak najszybciej i zajmij się znalezieniem przyczyny problemu.

Jakoś udało mi się otrząsnąć z początkowego szoku i zacząłem analizować testy. Zadałem sobie pytanie: *"Co ja tam właściwie przetestowałem?"*. I okazało się, że mój test sprawdza czy można poprawnie przejść ze strony o numerze 0 na stronę o numerze 1. Wydawało mi się logicznym, że skoro taki krok działa poprawnie, to poprawnie zadziałają również i kolejne: przejście z 1 na 2, z 2 na 3 i tak dalej. Rzecz wydawała się oczywista i to na tyle, że zadowoliłem się tym jednym testem. Najwyraźniej jednak nie miałem racji.

A teraz spójrzmy na kod produkcyjny odpowiedzialny za zwiększanie offsetu przy przechodzeniu na kolejną stronę:

```
public void goToNextPage() {
    this.offset = +perPage;
}
```

Ten niezwykle skomplikowany kawałek kodu przechodzi wszystkie testy jakie napisałem, a jednak jest niepoprawny! Tak przy okazji - czy zauważyłeś już błąd?

Napiszmy zatem test aby ostatecznie przekonać się, że problem naprawdę istnieje. Na przykład w ten sposób:

```
public void shouldIncreaseOffsetTwiceWhenGoingToPageTwo() {
    Pager pager = new Pager(PER_PAGE);

    pager.goToNextPage();
    pager.goToNextPage();

    assertThat(pager.getOffset()).isEqualTo(2 * PER_PAGE);
}
```

Test oczywiście nie przejdzie. Pora poprawić kod produkcyjny.

Główną lekcją, którą wyniosłem z tego przypadku jest to, że trzeba naprawdę dobrze zastanowić się co warto przetestować. Jeden prosty przypadek testowy to za mało. Potrzebna jest wieksza ilość kombinacji by móc ze spokojem powiedzieć, że "to działa".

Przypadki testowe powinny znacząco różnić się między sobą. To zwiększa szanse odnalezienia błędów. Dodawanie kolejnych podobnych przypadków testowych nic nie da. Co więcej pachnie to raczej naruszeniem zasady DRY[4]!

[4]Patrz https://pl.wikipedia.org/wiki/DRY

Zero, jeden i wiele

Istnieje prosta reguła, która pomaga dobrać przypadki testowe. Należy przetestować jak testowana funkcjonalność zachowuje się dla zera (dla braku), dla jednej wartości i dla wielu wartości. Dużo łatwiej wyjaśnić to na przykładzie niż zdefiniować, więc przejdźmy do przykładu:

Zgodnie z proponowaną regułą testując zapytanie SQL, powinniśmy rozpatrzyć następujące przypadki testowe:

a. w bazie danych brak obiektu spełniającego kryteria wyszukiwania,

b. w bazie danych jest dokładnie jeden obiekt spełniający kryteria wyszukiwania,

c. w bazie danych jest wiele obiektów spełniających kryteria wyszukiwania.

W przypadku opisywanego wcześniej komponentu do stronicowania przetestowałem przypadek dla zera i dla jednego. Pominąłem sprawdzenie dla wielu i zapłaciłem za to. Powinienem był dopisać test, który weryfikował poprawność przejścia np. ze strony piątej na szóstą.

3.4. Gdzie ten wyjątek?

Teraz zajmiemy się testem, który znalazłem w pewnym znalezionym w sieci poradniku. To naprawdę niedobry test i mógłbym mu sporo wytknąć. Skoncentrujemy się tylko na jednym jego aspekcie, mianowicie na sposobie w jaki weryfikuje on występowanie wyjątków.

Mam nadzieję, że Mama nauczyła cię, że nie można ufać wszystkiemu co się czyta! Stosuj się proszę do jej porad i starannie wybieraj poradniki, z których zamierzasz się uczyć!

Rzecz dotyczy własnej implementacji Javowego interfejsu `List`. Dwa poniżej zaprezentowane testy są jedynymi jakie znalazłem w owym poradniku. Ich zadaniem jest sprawdzić czy klasa `MyList` działa zgodnie z oczekiwaniami. Przyjrzyjmy się im.

```
@Test(expected=IndexOutOfBoundsException.class)
public void testMyList() {
    MyList<Integer> list = new MyList<Integer>();
    list.add(1);
    list.add(2);
    list.add(3);
```

```
    list.add(3);
    list.add(4);
    assertTrue(4 == list.get(4));
    assertTrue(2 == list.get(1));
    assertTrue(3 == list.get(2));

    list.get(6);
}

@Test(expected=IndexOutOfBoundsException.class)
public void testNegative() {
    MyList<Integer> list = new MyList<Integer>();
    list.add(1);
    list.add(2);
    list.add(3);
    list.add(3);
    list.add(4);
    list.get(-1);
}
```

Jak widać obie metody używają anotacji `@Test(expected=IndexOutOfBoundsException.class)`. Znaczy to, że oba przejdą **tylko jeżeli wyjątek tej klasy zostanie wyrzucony** w trakcie wykonywania metody testowej. Problem w tym, że używając anotacji `@Test` nie jesteśmy w stanie określić gdzie, tj. w której linii kodu, spodziewamy się wyjątku. Nieważne gdzie nastąpi katastrofa zakończona wyjątkiem: każde miejsce jest dobre by zaspokoić wymaganie zapisane przy pomocy anotacji `@Test(expected=IndexOutOfBoundsException.class)`. Nawet poniższa - niekompletna i zupełnie niedziałająca implementacja listy - przejdzie oba testy bez problemu.

```
public class MyList<T> {

    public MyList() {
        throw new IndexOutOfBoundsException();
    }

    public void add(T i) {
    }

    public T get(T i) {
        return null;
    }
}
```

Hm, to wręcz żałosne… Wypadało by poprawić sposób w jaki weryfikujemy wyjątki w naszych testach. Możemu zastosować dwa podejścia. Pierwsze polega na rozbicu metod testowych na kilka mniejszych (zgodnie z zasadą SRP, którą omówimy w sekcji 4.6). Drugie podejście polega na dokładniejszym określeniu miejsca, w którym spodziewamy się wyjątku.

3.4.1. Podział

Idąc pierwszą z zaproponowanych dróg skończymy z większą ilością małych metod testowych. Każda z nich będzie weryfikować inne wymaganie dotyczące klasy `MyList`. Przykładowo będziemy mieli metody typu; `shouldKeepValuesInOrder()`, `shouldAcceptDuplicateValues()` i im podobne. Kiedy już będziemy wiedzieli, że te funkcjonalności działają, możemy napisać test, który sprawdzi co się stanie gdy spróbujemy pobrać z listy nieistniejący element.

```
@Test(expected=IndexOutOfBoundsException.class)
public void shouldThrowExceptionWhenTryingToGetElementOutsideTheList() {
    MyList<Integer> list = new MyList<Integer>();
    list.add(0);
    list.add(1);
    list.add(2);
    list.get(3);
}
```

Jak widać ten test nie różni się znacząco od oryginalnego. Różnica polega na tym, że oprócz niego mamy jeszcze wiele innych metod testowych (np. wspominane już `shouldKeepValuesInOrder` i `shouldAcceptDuplicateValues()`). Dzięki temu wiemy, że konstruktor działa poprawnie i że podstawowa funkcjonalność związana z dodawaniem elementów do listy i ich pobieraniem również działa. W takim przypadku mamy większą pewność, że oczekiwany wyjątek klasy `IndexOutOfBoundsException` poleci w ostatniej linii naszej metody testowej.

3.4.2. Lokalizacja

Drugie podejście opiera się na lepszym określeniu miejsca, w którym spodziewamy się wyjątku. By było to możliwe, musimy użyć innego sposoby weryfikacji wyjątku niż użycie atrybutu `expected` anotacji `@Test`, gdyż jak już wiemy, ta jest usatysfakcjonowana wystąpieniem wyjątku w dowolnej linii metody.

Moglibyśmy użyć zwykłej konstrukcji `try-catch` i oczekiwać, że znajdziemy się w klauzuli `catch` (a w innym przypadku położyć test). Nie jest to jednak eleganckie rozwiązanie.

Podejście zaprezentowane poniżej używa:

- biblioteki catch-exception [5], która udostępnia metody `catchException()` oraz `caughtException()`, czyli dokładnie to czego nam trzeba,

[5] https://github.com/Codearte/catch-exception

- DSLa dostarczonego przez projekt AssertJ dzięki któremu z łatwością zbadamy złapany wyjątek.

```
@Test
public void shouldThrowExceptionWhenTryingToGetElementOutsideTheList() {
    MyList<Integer> list = new MyList<Integer>();
    list.add(0);
    list.add(1);
    list.add(2);

    catchException(list).get(3);

    assertThat(caughtException())
            .isExactlyInstanceOf(IndexOutOfBoundsException.class);
}
```

No, teraz sprawdzamy już dokładnie, czy oczekiwany wyjątek powstaje dokładnie tam gdzie tego oczekujemy!

Biblioteka catch-exception jest bardzo przydatna do testowania wyjątków i warto ją poznać. Jeżeli używasz Javy 8 nie będziesz jej już potrzebował - wystarczą lambdy.

3.5. Zmień się, albo...

> Jedyną stałością w produkcji oprogramowania jest nieustanna zmiana.
>
> — Mądrość programistyczna

Podczas jednego code review pewien krótki test przyciągnął moją uwagę. Zamieszczam go poniżej. Jak widać weryfikuje on działanie klasy `Transaction`, która przetrzymuje informacje o użytkowniku zaangażowanym w tę transakcję oraz o kwocie transakcji. Obie metody testowe weryfikują przypadki negatywne: system powinien odrzucić transakcje z kwotą równą zero lub mniejszą od zera.

```
public class TransactionTest {

    @Test
    public void shouldRecognizeTransactionsWithZeroValueAsInvalid() {
        //given
        Transaction tx = new Transaction(BigDecimal.ZERO,
            new InternalUser());

        //when
```

```
        boolean actual = tx.validate();

        //then
        assertThat(actual).isFalse();
    }

    @Test
    public void shouldRecognizeTransactionWithNegativeValueAsInvalid() {
        //given
        Transaction tx = new Transaction(BigDecimal.ONE.negate(),
            new InternalUser());

        //when
        boolean actual = tx.validate();

        //then
        assertThat(actual).isFalse();
    }
}
```

Po bliższej analizie okazało się, że ten test **kiedyś** był całkiem rozsądny. Mianowicie jakiś czas temu metoda `validate()` wyglądała następująco:

```
public boolean validate() {
    return amount.compareTo(BigDecimal.ZERO) > 0;
}
```

Jednak w momencie code review ta sama metoda wyglądała już inaczej:

```
public boolean validate() {
    if (!user.isExternal()) {
        return false;
    }
    return amount.compareTo(BigDecimal.ZERO) > 0;
}
```

Zauważmy, że w obu metodach testowych używamy klasy `InternalUser`, która jak można przypuszczać spełnia warunek `!user.isExternal()`. W związku z czym, to właśnie ta pierwsza linijka metody `validate()` sprawia, że oba testy przechodzą. Wartość kwoty transakcji - w pierwszym przypadku 0, w drugim -1 - nie ma tu nic do rzeczy. W ogóle nie jest weryfikowana.

Jak doszło do takiej sytuacji? To wydaje się oczywiste: najwyraźniej zmianie kodu nie towarzyszyła zmiana testów.

Poprawna zmiana powinna wyglądać następująco. Obie metody testowe powinny używać instancji klasy `ExternalUser` zamiast dotychczasowego `InternalUser` (lub jeszcze lepiej, użyć stuba zwracającego odpowiednie wartości). Dodatkowo potrzebny jest kolejny test, który

testowałby zachowanie tej metody przy poprawnej kwocie i użytkowniku nie spełniającym wymagania `isExternal()`.

Kiedy zmienia się kod trzeba dobrze zastanowić nad niezbędnymi zmianami w testach. Sam fakt, że testy przechodzą, to za mało by uznać, że wszystko jest w porządku!

A tak przy okazji, gdyby się nad tym głębiej zastanowić… Czy przypadkiem tego typu problem w ogóle by nie powstał gdyby **najpierw napisać test** a dopiero potem dokonać zmian w kodzie produkcyjnym? Wydaje się, że wówczas trudno byłoby sprowokować tego typu przypadek.

Najpierw pisz testy!

3.6. Asercje: bezlitosne i nieubłagane

Nie każdy test kończący się sekwencją `assertXYZ` jest wystarczająco dobry. Chodzi bowiem o to, aby asercje naprawdę weryfikowały testowany scenariusz. Jak się zaraz przekonamy nie zawsze tak jest. Spójrzmy na następujący fragment kodu:

```
@Test
public void shouldRemoveEmailsByState() {
    //given
    Email pending = createAndSaveEmail("pending","content pending",
            "abc@def.com", Email.PENDING);
    Email failed = createAndSaveEmail("failed","content failed",
            "abc@def.com", Email.FAILED);
    Email sent = createAndSaveEmail("sent","content sent",
            "abc@def.com", Email.SENT);

    //when
    emailDAO.removeByState(Email.FAILED);

    //then
    assertThat(emailDAO.findAll()).doesNotContain(failed);
}
```

Ten test w sekcji *"given"* tworzy kilka obiektów klasy `Email` i zapisuje je w bazie danych. Następnie uruchamia metodę `removeByState()` testowanej klasy `EmailDAO`. A potem sprawdza, czy… no właśnie, co on właściwie sprawdza?

Końcowa asercja sprawdza, że email `failed` nie znajduje się już w bazie danych. Niestety, to nie wystarcza by powiedzieć, że metoda `removeByState()` działa jak powinna. Ten test przejdzie nawet jeżeli testowana metoda działa zupełnie niepoprawnie i usuwa wszystkie emaile z bazy danych.

Istnieje kilka sposobów by zabezpieczyć się przed takimi niepokojącymi scenariuszami poprawiając końcową asercję. Można zapisać ją na przykład tak:

```
assertThat(emailDAO.findAll())
    .isNotEmpty()
    .doesNotContain(failed);
```

Albo, co wydaje się jeszcze lepsze, w ten sposób:

```
assertThat(emailDAO.findAll())
    .contains(pending, sent)
    .doesNotContain(failed);
```

Ostatnie rozwiązanie sprawdza dokładnie czy usunięte zostały tylko emaile o statusie `FAILED`.

Metody `contains()` i `doesNotContain()` pochodzą z frameworku AssertJ (który polecam Ci używać zamiast asercji dostarczanych przez JUnit lub TestNG).

3.7. Czy Mockito działa poprawnie?

Tą sekcję napisał Tomasz Borek (https://lafkblogs.wordpress.com/). Tłumaczenie z angielskiego: Tomek Kaczanowski.

Tak więc mamy formularz. I prędzej czy później wypełnimy go danymi. I z pewnych powodów chcielibyśmy mieć kontrolę nad tym, czy dane w formularzu można modyfikować, czy też nie. Przedstawiony poniżej test teoretycznie testuje właśnie to wymaganie.

```
@Test
public void testFormUpdate() {
    // given
    Form f = Mockito.mock(Form.class);
    Mockito.when(f.isUpdateAllowed()).thenReturn(true);

    // when
    boolean result = f.isUpdateAllowed();

    // then
```

```
    assertTrue(result);
}
```

Hm… coś chyba jest nie tak z tym jakże prostym i przejrzystym testem. Cóż, problem polega na tym, że on wcale nie testuje klasy `Form`, tylko framework Mockito! Jeżeli `Mockito.mock()` i +Mockito.when()+ działają jak powinny, to test będzie zielony. Nawet gdy kompletnie zmienimy logikę metody `isUpdateAllowed()` test wciąż będzie przechodził.

Co pozostaje nam zrobić? Oczywiście powinniśmy uzależnić wynik testu od zachowania testowanej klasy. Zastąpienie testowanego obiektu mockiem nie ma sensu. Zamiast tego powinniśmy użyć prawdziwego obiektu, jak poniżej:

```
@Test
public void testFormUpdate() {
    // given
    Form f = new Form();
    f.setUpdateAllowed(true);

    // when - then
    assertTrue(f.isUpdateAllowed());
}
```

W bardziej skomplikowanych przypadkach możemy zostać zmuszeni do zastępowania obiektów mockami. Wystarczyłoby, jeśli metoda `isUpdateAllowed()` do odpowiedzi wymagałaby zewnętrznego komponentu (walidatora bądź mechanizmu bezpieczeństwa). Wciąż jednak mockami zastępowalibyśmy właśnie te zewnętrzne komponenty, nie zaś testowaną klasę!

3.8. WWW czyli Weryfikacja Wyjątków Wymiata

Frameworki testowe są zwodniczo proste. Niby banalne w użyciu, aż tu nagle - **trach!** Czasami zachowują się nieco inaczej niż oczekujemy.

Przyjrzyjmy się temu kawałkowi kodu produkcyjnego:

```
public void registerDomain(Domain domain) {
    try {
        dnsService.addDomainIfMissing(domain.getAddress());
    } catch (RuntimeException ex) {
        domainService.saveDomain(domain, domain.isRegisteredInDns(),
            domain.getDnsFailures() + 1);
        throw ex;
    }
    domainService.saveDomain(domain, domain.isRegisteredInDns(),
```

```
            domain.getDnsFailures());
}
```

Programista zamierzał sprawdzić czy metoda `saveDomain()` klasy `DomainService` zostanie wywołana z odpowiednią liczbą błędów DNS. Szczególnie interesujące byłoby stwierdzić czy w przypadku wyłapania wyjątku liczba ta wzrośnie. W tym celu powstał następujący test:

```
@Test(expected = RuntimeException.class)
public void shouldSaveFailureInformationWhenExceptionOccurWhenAddingDomain() {
    //given
    doThrow(new RuntimeException()).when(dnsService)
        .addDomainIfMissing(DOMAIN_ADDRESS);

    //when
    domainRegistrator.registerDomain(domain);

    //then
    verify(domainService)
        .saveDomain(domain, false, DNS_FAILURES + 1);
}
```

Ten test stara się zweryfikować dwie rzeczy. Po pierwsze, upewnia się, że został wyrzucony odpowiedni wyjątek. Za to odpowiada atrybut `expected` anotacji `@Test` nad metodą testową. Po drugie, ostatnia linia testu sprawdza czy zaszło oczekiwane wywołanie metody `saveDomain()` z odpowiednimi parametrami.

Tak przy okazji. Lepiej jeżeli testy są zgodne z zasadą SRP. Czasami jednak, np. gdy koszt uruchomienia testu jest wysoki, co zdarza się w testach integracyjnych i end-to-end, wówczas warto weryfikować więcej niż jedną rzecz w metodzie testowej.

Wydaje się, że ten test robi dokładnie to na czym nam zależało.
Niestety, tak naprawdę to działa on tylko w połowie. Problem tkwi w drugiej weryfikacji. Rzecz w tym, że ta linia kodu:

```
domainRegistrator.registerDomain(domain);
```

wyrzuca wyjątek, który kończy wykonanie testu! Kolejna linia - `verify(domainService)`... nigdy się zatem nie wykona.

Na szczęście łatwo to naprawić. Wystarczy złapać wyjątek w teście, dzięki czemu ostatnia linia też się wykona. Można poradzić sobie przy pomocy konstrukcji `try/catch` albo posłużyć się biblioteką catch-exception:

```
@Test
public void shouldSaveFailureInformationWhenExceptionOccurWhenAddingDomain() {
```

```
    //given
    doThrow(new RuntimeException()).when(dnsService)
        .addDomainIfMissing(DOMAIN_ADDRESS);

    //when
    catchException(domainRegistrator).registerDomain(domain);

    //then
    verify(domainService)
        .saveDomain(domain, false, DNS_FAILURES + 1);
    assertThat(caughtException()).isInstanceOf(RuntimeException.class);
}
```

W poprawionej wersji testu pozbyliśmy się atrybutu `expected` z anotacji `@Test`. Zamiast tego łapiemy wyjątek metodą `catchException()` i weryfikujemy go przy pomocy metody `caughtException()`.
Po tej zmianie ostatnia asercja też zostanie wywołana, bo wykonanie kodu testowego nie jest już przerywane przez rzucenie wyjątku.

Frameworki testowe są proste, ale nie aż tak proste. Warto czasem zajrzeć do ich dokumentacji.

A na koniec tradycyjna już uwaga na temat sposobu pisania testów. Tak, również i ten test nie powstałby w swojej pierwotnej postaci, gdyby programista napisał go zgodnie z regułami TDD (w szczególności gdyby zobaczył, że test nie przechodzi z powodu złej ilości błędów DNS przekazywanych do metody `saveDomain()`). Po raz kolejny okazuje się, że pisząc testy po napisaniu kodu sporo ryzykujemy.

Najpierw uruchom test i zobacz, że nie przechodzi. Dopiero potem napisz kod, który sprawi, że test przejdzie.

3.9. Mockito: any() czy isA()

Ten przykład pochodzi z bloga Bartka Zdanowskiego (http://touk.pl/blog/author/bzd/). Z angielskiego przetłumaczył Tomek Kaczanowski.

Spójrzmy na kawałek kodu produkcyjnego:

```
public class AddOrganizationAction implements Action { ... }
public class AddPersonToOrganizationAction implements Action { ... }

public interface DispatchAsync {
    void execute(Action action, AsyncCallback callback);
}
```

Pisząc test możemy mieć ochotę użyć matchera `any()`, na przykład w taki sposób;

```
verify(async).execute(any(AddOrganizationAction.class),
    any(AsyncCallback.class));
```

Wszystko wydaje się być w porzadku. A jednak coś tu nie gra. Spróbujmy zmienić pierwszy parametr wywołania metody z `AddOrganizationAction.class` na `AddPersonToOrganizationAction.class`:

```
verify(async).execute(any(AddPersonToOrganizationAction.class),
    any(AsyncCallback.class));
```

Coś jest bardzo nie tak, bo test jak przechodził, tak nadal przechodzi! Jak to możliwe?

By rozwiązać tę zagadkę zerknijmy do dokumentacji klasy `Matcher`. Znajdziemy tam co następuje:

> Ta metoda **nie dokonuje żadnego sprawdzenia typów**; jest tylko po to, żebyś nie musiał rzutować[6].
>
> — Mockito JavaDocs

Wniosek jest taki, że ilekroć używasz konstrukcji `any(SomeClass.class)` w nadziei zweryfikowania czy to właśnie obiekt tej klasy został przekazany jako parametr wywołania pewnej metody, tylekroć trafiasz jak kulą w płot!

Przeczytaj dokumentację narzędzi, których używasz. I pomóż autorom ulepszyć je, jeżeli tylko znajdziesz jakieś niedoskonałości.

W opisanym przypadku wystarczy zastąpić matcher `any()` innym matcherem: `isA()`. W przeciwieństwie do `any()` ten matcher sprawdza przekazany typ.

```
verify(async).execute(isA(AddOrganizationAction.class),
    any(AsyncCallback.class));
```

[6] W oryginale: *This method **don't do any type checks**; it is only there to avoid casting in your code.*

Przejrzyj swoje testy w poszukiwaniu użycia matchera `any()`. Zastanów się czy `isA()` nie byłby bardziej odpowiedni.

3.10. Bądź ogólny!

Dawno dawno temu, w pewnym zapomnianym już projekcie, był sobie servlet, który przyjmował dwa parametry: `packet` i `type`. Jeżeli oba parametry było obecne w requeście, wówczas servlet zwracał się z prośba do swojego współpracownika - `packetDataProcessor` - by ten zajął się obsługą konkretnego pakietu. Jeżeli brakowało któregoś z parametrów, wówczas `packetDataProcessor` nie miał nic do roboty. Proste, prawda? Spójrzmy zatem na test.

3.10.1. Co się nie stało

Zacznijmy od przyjrzenia się testowi typu "happy path" (patrz sekcja 3.3). Sprawdza on, czy servlet zachowuje się poprawnie jeżeli w requeście otrzyma 2 poprawne parametry.

```
@Test
public void shouldProcessPacket() throws IOException, ServletException {
    //given
    given(request.getParameter(PacketApi.PACKET_PARAMETER))
        .willReturn(PACKET);
    given(request.getParameter(PacketApi.TYPE_PARAMETER))
        .willReturn(TYPE);

    //when
    servlet.doGet(request, response);

    //then
    verify(packetDataProcessor).process(PACKET, TYPE);
}
```

Jak dla mnie bomba. Być może lekko zmodyfikowałbym nazwę test tak, by pełniej opisywała testowany scenariusz, ale poza tym wydaje mi się idealny.

Spójrzmy jednak na kolejny test:

```
@Test
public void shouldNotProcessIfPacketParameterIsMissing()
                throws IOException, ServletException {
    //given
    given(request.getParameter(PacketApi.TYPE_PARAMETER))
        .willReturn(TYPE);
```

```
    //when
    servlet.doGet(request, response);

    //then
    verify(packetDataProcessor, never()).process(PACKET, TYPE);
}
```

Tutaj mamy do czynienia z testem negatywnym, który sprawdza co się stanie gdy zabraknie jednego parametru[7]. Oczekujemy, że w takim przypadku pakiet nie zostanie przeprocesowany.

…ale zaraz, jedna rzecz wydaje się dziwna. Dlaczego ostatnia linia testu weryfikuje czy metoda `process()` zostanie wywołana z parametrem `PACKET`? To zastanawiające, bo przecież taka wartość nie występuje nigdzie w teście. Aha! Nie ma jej w teście, ale pewnie wciąż tkwi w głowie programisty, który ledwo co skończył pisać poprzedni test (patrz poprzedni listing).

Wydaje się, że sito weryfikacyjne jest w tym przypadku zbyt szczelne. Tak naprawdę to chcielibyśmy napisać następujący test: *"Jeżeli brakuje parametru* `PACKET_PARAMETER` *to* `packetDataProcessor` *nie powinien w ogóle nic robić"*. Teraz wystarczy przetłumaczyć to na język Javy/Mockito/JUnita. A oto i rezultat:

```
@Test
public void shouldProcessIfPacketParameterIsMissing()
                throws IOException, ServletException {
    //given
    given(request.getParameter(PacketApi.TYPE_PARAMETER))
        .willReturn(TYPE);

    //when
    servlet.doGet(request, response);

    //then
    verifyZeroInteractions(packetDataProcessor);
}
```

Metoda `verifyZeroInteractions()` z biblioteki Mockito nadaje się właśnie do takich zastosowań. Chcemy sprawdzić, że nie było żadnych odwołań do metod obiektu `packetDataProcessor`, i dokładnie to udało nam się zapisać w kodzie testu.

[7]Fakt braku parametru `PACKET_PARAMETER` nie jest explicite określony w teście, ale Mockity domyślnie zwróci `null` gdy servlet poprosi obiekt `request` o ten parametr. Pewnie to wiedziałeś, ale pomyślałem, że nie zaszkodzi przypomnieć. :)

Pisać czy pozostawić domysłom?

Jak wiesz, Mockito zwraca pewne domyślne wartości (takie jak `null` czy pustą kolekcję). Nie musisz instruować zmockowanych obiektów - gdy wywołasz ich metody dostaniesz te właśnie domyślne wartości. Oznacza to tyle, że poniższe fragmenty kodu są równoważne.

W tym przypadku dokładnie opisujemy, że próba pobrania `PACKET_PARAMETER` zwróci nam `null`.

```
HttpServletRequest request = mock(HttpServletRequest.class);
given(request.getParameter(PacketApi.PACKET_PARAMETER))
    .willReturn(null); ❶
given(request.getParameter(PacketApi.TYPE_PARAMETER))
    .willReturn(TYPE);
```

❶ tu instruujemy Mockito, że w odpowiedzi ma zwrócić `null`

Druga wersja, w której zadowalamy się domyślnymi wartościami.

```
HttpServletRequest request = mock(HttpServletRequest.class);
given(request.getParameter(PacketApi.TYPE_PARAMETER))
    .willReturn(TYPE);
```

Pytanie, którą z tych wersji powinniśmy preferować? Wydaje mi się, że to raczej kwestia gustu. Mój podpowiada mi, że lepiej by test opowiadał "całą historię". Dlatego też raczej dodaję te (nadmiarowe) linie kodu, które specyfikują zachowania identyczne z domyślnymi zachowaniami Mockito. Sądzę, że ułatwia to zrozumienie scenariusza testowego.

3.10.2. Wyjątek, ale jaki wyjątek?

Przeanalizujmy teraz inny przykład pochodzący z tej samej klasy testowej. Tym razem sprawdzamy jak zachowuje się servlet po wystąpieniu błędu w procesie obsługi żądania przez `packetDataProcessor`.

```
@Test
public void shouldReturnStatus500IfThereWasAnErrorDuringProcessing()
                throws IOException, ServletException {
    //given
    given(request.getParameter(PacketApi.PACKET_PARAMETER))
        .willReturn(PACKET);
    given(request.getParameter(PacketApi.TYPE_PARAMETER))
        .willReturn(TYPE);
```

```
    doThrow(NullPointerException.class)
        .when(packetDataProcessor).process(PACKET,TYPE);

    //when
    servlet.doGet(request, response);

    //then
    verify(response).setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
}
```

W tym teście nakazujemy obiektowi `packetDataProcessor` symulować błąd poprzez rzucenie wyjątku klasy `NullPointerException`. Hmm… tak właściwie, to czemu z mnogości dostępnych wyjątków wybraliśmy akurat tę klasę? Trudno mi to powiedzieć. Spojrzałem w implementację klasy `PacketDataProcessor` i dowiedziałem się, że rzuca ona różnego typu wyjątki dotyczące parsowania. Test, który koncentruje się akurat na wyjątku klasy `NullPointerException`, nie pozwoli nam dowiedzieć się, jak zachowuje się servlet gdy rzucane są te właśnie wyjątki związane z błędami parsowania. Słowem, nie mówi nam tego, co chcielibyśmy wiedzieć.

Podobnie jak w poprzednim przykładzie, tutaj również zbyt szczegółowo wyspecyfikowaliśmy nasze oczekiwania. Nie powinniśmy się koncentrować na wyjątku określonej klasy, tylko sprawdzić czy servlet ustawia status `SC_INTERNAL_SERVER_ERROR` w odpowiedzi na **dowolny** wyjątek rzucony podczas procesowania danych.

```
@Test
public void shouldReturnStatus500IfThereWasAnErrorDuringProcessing()
                throws IOException, ServletException {
    //given
    given(request.getParameter(PacketApi.PACKET_PARAMETER))
        .willReturn(PACKET);
    given(request.getParameter(PacketApi.TYPE_PARAMETER))
        .willReturn(TYPE);
    doThrow(Exception.class)
        .when(packetDataProcessor).process(PACKET,TYPE);

    //when
    servlet.doGet(request, response);

    //then
    verify(response).setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
}
```

Zastanów się dobrze co naprawdę chciałbyś sprawdzić. Nie koncentruj się na jednym przypadku, bo to osłabi siłę testu.

3.11. Pisz te testy, które trzeba

W przeciwieństwie do wszystkich innych przykładów z książki tym razem wyjątkowo nie zaprezentuje żadnego kodu. Ale opowiem historię z pewnego projektu.

Był to całkiem porządny projekt. Jakość kodu oceniłbym wysoko. Architektura była rozsądna, ludzie sprawni i zaangażowanie. Wszystko wydawało się być na swoim miejscu. A jednak wraz ze wzrostem złożoności tworzonego systemu, zaczęliśmy doświadczać różnych porażek. Wykresy prezentowane na dashboardzie klienta pokazywały zupełne bzdury, gdy użytkownik zmienił strefę czasową. Zdarzało się, że w czasie dużych zmian w kodzie w tajemniczych okolicznościach "gubiliśmy" jakąś funkcjonalność. Użytkownicy zgłaszali problemy z funkcją eksportu danych do CSV: przy zbyt dużej ilości danych zawieszała się. Po zarejestrowaniu się do systemu użytkownicy (czasami) nie otrzymywali stosownych powiadomień. I tak dalej.

To wszystko bardzo nas frustrowało. W końcu mieliśmy testy jednostkowe! Mieliśmy ich całe mnóstwo! Nasz kod był obiektowy aż do bólu - każda mała klasa była testowalna i pokryta testami tak, że pokrycie kodu sięgało sufitu.

Morał z tej historii jest taki, że powinieneś dobrze zastanowić się jaki kształt "piramidy testów" będzie właściwy dla aplikacji którą tworzysz. Jeżeli włożysz zbyt wiele wysiłku w testy jednostkowe, ale niewystarczająco dużo w testy typu end-to-end, wówczas prawdopodobnie natkniesz się na problemy takie jak opisałem powyżej. Jeżeli zrobisz na odwrót, napotkasz inne trudności - np. okaże się, że pewne przypadki brzegowe działają nieprawidłowo, i że masz mnóstwo roboty z utrzymaniem ciężkich i wolnych testów end-to-end.

"Szacun" dla wszystkich, którym udaje się dobrać odpowiednie proporcje testów jednostkowych, integracyjnych i end-to-end!

W tej książce poświęcamy wiele uwagi temu, by testy pisać **we właściwy sposób**. Rozważamy kwestie nazewnictwa, oddzielenie kodu testowego od implementacji, przejrzystość kodu i tym podobne. Każda z tych rzeczy z osobna może wydawać się nieistotna, ale gdy je zlekceważysz, szybko znajdziesz się w dżungli trudnych do utrzymania testów. Mam nadzieje, że zdajesz sobie z tego sprawę!

To wszystko prawda, i to są naprawde ważne rzeczy. Jednak to nie wystarczy. Nawet najlepsze, najpiękniejsze testy nie wystarczą, jeżeli (po zsumowaniu) nie pokrywają całości funkcjonalności systemu.

Skup się na pisaniu tych testów, które wzmacniają poczucie, że system działa.

Przerwa reklamowa :)

Hej, jak leci? Czy jesteś zadowolony z dotychczasowej lektury? Masz poczucie, że czegoś się nauczyłeś? Mam nadzieję, że tak!

Być może zainteresują Cię inne moje książki poświęcone testom. Zapraszam na http://practicalunittesting.com po szczegóły. A może interesuje Cię kanban? Jeżeli tak, to polecam http://123kanban.pl

Rozdział 4. Utrzymanie

> […] jeżeli testy nie były tworzone z myślą o tym, żeby łatwo było je modyfikować, to zmienianie ich po zmianach systemu może zająć dużo czasu.
>
> — Gojko Adzic *Specification by Example*

Testy pomagają nam utrzymywać nasz software. Gdy wprowadzamy zmiany wówczas informują nas o tym, co przestało działać. I to jest **absolutnie bezcenne**!

Jest jednak druga strona medal. Pytanie, jak bardzo "kruche" są nasze testy? Ile z nich będziemy musieli zmienić, jeżeli zmienimy tylko jedną linijkę kodu produkcyjnego? Jak dużo czasu i wysiłku nam to zabierze?

W tej sekcji będziemy przyglądać się testom biorąc pod uwagę ich odporność na zmiany w kodzie produkcyjnym. W przeciwieństwie do testów z poprzedniego rozdziału, przedstawione tu testy zazwyczaj testując coś całkiem porządnie. Zajmiemy się tym jak je poprawić, by nie wymagały od nas tak wiele pracy przy zmianach w kodzie.

4.1. Mockujemy wszystko!

> Gdy twoim jedynym narzędziem jest młotek, wszystko zaczyna ci przypominać gwoździe.
>
> — Abraham Maslow

Obserwuje takie zachowanie zwłaszcza u programistów, którzy dopiero co odkryli rozkosze mockowania. :) Stosują mocki wszędzie - nawet tam, gdzie nie powinni. Porozmawiajmy o tym posiłkując się przykładami kodu.

4.1.1. Mockowanie pojemnika na dane

Poniższy kod stara się zweryfikować czy pewne dane zostały dodane do obiektu `modelAndView`[1]. Tak, nie przesłyszałeś się: weryfikuje czy coś **zostało dodane**.

```
@Test
public void shouldAddTimeZoneToModelAndView() {
    //given
    Context context = mock(Context.class);
    ModelAndView modelAndView = mock(ModelAndView.class);
    given(context.getTimezone()).willReturn("timezone X");

    //when
```

[1] Jest to obiekt klasy `ModelAndView` pochodzący z biblioteki Spring MVC.

```
    new UserDataInterceptor(context)
        .postHandle(null, null, null, modelAndView);

    //then
    verify(modelAndView).addObject("timezone", "timezone X");
}
```

Co tak naprawdę nas obchodzi: czy fakt, że `modelAndView` zawiera pewne dane, czy też może to, że zostały wywołane na nim określone metody? Właściwie można by powiedzieć, że nie ma o co kruszyć kopii. W końcu skoro została wywołana metoda `addObject()` to możemy zakładać, że dane zostały dodane, czyli że tam są. Więc właściwie można powiedzieć, że ten test radzi sobie ze sprawdzeniem czy obiekt `modelAndView` zawiera wpis o kluczu `timezone`.

Reguły dobrego pisania testów mówią jednak, że zawsze powinniśmy raczej weryfikować wynik działania, a nie sposób w jaki został on uzyskany. Dzięki takiemu podejściu mamy możliwość dokonywania zmian w kodzie produkcyjnym (w szczególności zmian algorytmu generującego wynik), który nie zmusza nas do dokonywania zmian w teście.

Podany poniżej przykład zilustruje tę sytuacje. Nie jest on co prawda zgodny z API klasy `ModelAndView` API, ale wystarczający dla naszych potrzeb.
Zastanówmy się zatem, czy istotne jest sprawdzić czy zmienna `timezone` została ustawiona w taki sposób:

```
ModelAndView mav = new ModelAndView();
mav.addObject("timezone", "timezone X");
```

czy w taki?

```
ModelAndView mav = new ModelAndView("timezone", "timezone X");
```

Czy to ma jakieś znaczenie w jaki sposób wartość o kluczu `timezone` dostała się do obiektu `modelAndView`? Nie wydaje mi się. I nie wydaje mi się rozsądnym, żeby test zajmował się takimi niuansami. A jednak test, którym rozpocząłem tą opowieść, zajmuje się dokładnie tym: sprawdzeniem w jaki sposób SUT wykonuje pewne zadania.

Poniżej zamieszczam zmodyfikowaną wersję testu. Traktuje on testowany obiekt bardziej "czarnoskrzynkowo" i interesuje się wyłącznie zwracanymi wartościami.

```
@Test
public void shouldAddTimeZoneToModelAndView() {
    //given
    Context context = mock(Context.class);
    ModelAndView modelAndView = new ModelAndView();
    given(context.getTimezone()).willReturn("timezone X");

    //when
    new UserDataInterceptor(context)
```

```
        .postHandle(null, null, null, modelAndView);

    //then
    assertThat(modelAndView).contains("timezone", "timezone X"); ❶
}
```

❶ Taka metoda naprawdę nie istnieje, ale chodziło mi tu o wyrażenie biznesowej intencji testu, a nie o zgodność z API.

Jak widać tym razem użyłem prawdziwych obiektów, a nie mocków. I ma to sens ponieważ klasa `ModelAndView` nie jest serwisem (nie udostępnia funkcjonalności) ale kontenerem na dane[2], i jako taka nie powinna zostać zmockowana. Howgh!

Podekscytowanie, że oto potrafię zmockować wszystko, trwa dłuższą chwilę. Nie daj mu sobą kierować, i nie używaj mocków póki nie masz pewności, że naprawdę musisz zweryfikować jak obiekty porozumiewają się ze sobą. W wielu przypadkach nie musisz, a nawet nie powinieneś.

4.1.2. PrintWriter

Przed nami kolejny przykład nadużycia mocków. On rownież pochodzi z aplikacji oparte o Spring MVC. Pewien kontroler - nasz SUT - zapytany o dane miał dwa zadania do wykonania:

- pobrać dane od swojego współpracownika - obiektu typu `ReportService`,
- wysłać te dane w formacie CSV do użytkownika.

```
public class ReportController {

    @Autowired
    private final ReportService reportService;

    public void generateReport(HttpServletRequest request,
            HttpServletResponse response) throws IOException {
        Filter filter = parseRequest(request);
        List<ReportData> reportData = reportService.getReportData(filter);
        PrintWriter writer = response.getWriter();
        writeHeaders(writer);
        for (ReportData data : reportData) {
            writer.append(String.valueOf(data.getMin()));
            writer.append(",");
            writer.append(String.valueOf(data.getMax()));
            writer.append(",");
            writer.append(String.valueOf(data.getAvg()));
```

[2]Tak naprawdę klasa `ModelAndView` to nieco rozszerzona mapa służąca jedynie przekazaniu danych z kontrolera do widoków.

```
            writer.append("\n");
        }
    }

    private void writeHeaders(PrintWriter writer) {
        writer.append("min,max,avg\n");
    }

    private Filter parseRequest(HttpServletRequest request) { ... }
}
```

Nas szczególnie interesuje fragment użycia klasy `PrintWriter` do tworzenia odpowiedzi dla użytkownika. By przetestować wynik (czyli zawartość przekazywanej użytkownikowi odpowiedzi) musimy jakoś przechwycić dane pisane do strumienia. Oto próba napisania takiego testu:

```
public class ControllerTest {

    HttpServletRequest req = mock(HttpServletRequest.class);
    HttpServletResponse resp = mock(HttpServletResponse.class);
    ReportService reportService = mock(ReportService.class);
    PrintWriter writer = mock(PrintWriter.class);

    ReportController ctrl = new ReportController(reportService);

    @Test
    public void shouldWriteReportData() throws IOException {
        // given
        ReportData dataPl = new ReportData(1, 2, 1.5);
        ReportData dataFr = new ReportData(3, 4, 0.12345);

        given(resp.getWriter()).willReturn(writer);
        given(reportService.getReportData(any(Filter.class)))
                .willReturn(Arrays.asList(dataPl, dataFr));

        // when
        ctrl.generateReport(req,  resp);

        // then
        InOrder inOrder = Mockito.inOrder(writer);
        inOrder.verify(writer).append("min,max,avg\n");
        inOrder.verify(writer).append(String.valueOf(1));
        inOrder.verify(writer).append(",");
        inOrder.verify(writer).append(String.valueOf(2));
        inOrder.verify(writer).append(",");
        inOrder.verify(writer).append(String.valueOf(1.5));
        inOrder.verify(writer).append("\n");
        inOrder.verify(writer).append(String.valueOf(3));
        inOrder.verify(writer).append(",");
```

```
        inOrder.verify(writer).append(String.valueOf(4));
        inOrder.verify(writer).append(",");
        inOrder.verify(writer).append(String.valueOf(0.12345));
        inOrder.verify(writer).append("\n");
    }
}
```

Hmm… No trzeba przyznać, że część *"then"* wygląda nieciekawie. Weryfikacja polega na sprawdzeniu każdej kolejnej metody wywołanej na obiekcie `writer`. Powoduje to, że test jest bardzo nieczytelny, ale co gorsza, że jest niezwykle kruchy. Każda zmiana w sposobie pisania do strumienia spowoduje konieczność przeróbki testu. Niedobrze.

Wydaje się, że w tym przypadku zastąpienie mockiem obiektu klasy `PrintWriter` nie jest dobrym rozwiązaniem. Spróbujmy zatem postąpić inaczej.

```
public class ControllerWithStringWriterTest {

    HttpServletRequest req = mock(HttpServletRequest.class);
    HttpServletResponse resp = mock(HttpServletResponse.class);
    ReportService reportService = mock(ReportService.class);

    ReportController ctrl = new ReportController(reportService);

    @Test
    public void shouldWriteReportData() throws IOException {
        // given
        ReportData dataPl = new ReportData(1, 2, 1.5);
        ReportData dataFr = new ReportData(3, 4, 0.12345);

        Writer stringWriter = new StringWriter();
        PrintWriter writer = new PrintWriter(stringWriter);
        given(resp.getWriter()).willReturn(writer);
        given(reportService.getReportData(any(Filter.class)))
                .willReturn(Arrays.asList(dataPl, dataFr));

        // when
        ctrl.generateReport(req,  resp);

        // then
        assertThat(stringWriter.toString())
                .isEqualTo("min,max,avg\n1,2,1.5\n3,4,0.12345\n");
    }
}
```

Weryfikacja jest teraz o niebo czytelniejsza! A to dzięki zmianie, jaką było zastąpienia mocka klasy `PrintWriter` prawdziwą implementacją tejże. Chytrość polega na skonstruowaniu obiektu klasy `PrintWriter` z użyciem innego `writera`, z którego łatwo można odczytać wpisane dane. Dzięki temu bez trudu zweryfikujemy odpowiedź jaką kontroler przesyła klientowi.

Konceptualnie różnica polega na przejściu od weryfikacji *"jak to zostało zrobione"* do weryfikacji wyniku działania. Dzięki tej zmianie możemy do woli modyfikować implementację metody `generatReport()`.

W powyższym przykładzie użyłem biblioteki Mockito do stworzenia mocków klas `HttpServletRequest` i `HttpServletResponse`. Równie dobre byłoby użycie specjalnie do tego celu przeznaczonych klas z projektu Spring: `MockHttpServletRequest` i `MockHttpServletResponse`.

4.2. Kontroluj otoczenie

Powinieneś zawsze mieć świadomość co do otoczenia, w jakim uruchamiane są testy. W przeciwnym wypadku możesz spędzić sporo czasu zastanawiając się dlaczego testy zachowują się dziwnie. Może minąć sporo czasu, nim zorientujesz się, że pewne założenia odnośnie środowiska były nazbyt optymistyczne. Tak więc jeżeli używasz w testach systemu plików, bazy danych albo zewnętrznych serwisów, to lepiej miej całkowitą pewność co do stanu, w jakim znajdują się one przed rozpoczęciem testów.

W tej sekcji zajmiemy się dwoma przykładami ilustrującymi problemy wynikłe z ignorowania powyższych uwag.

4.2.1. Zależność między testami

Jednym z wymagań wobec testów jednostkowych jest by były od siebie niezależne. I słusznie, bo takie testy dużo łatwiej utrzymywać i uruchamiać (w dowolnej kolejności). Dla testów integracyjnych i end-to-end czasami jednak sensowne jest uzależnić je od siebie (i przykładowo zawsze uruchamiać je w pewnej kolejności). I nie jest to aż takie złe, szczególnie jeżeli zadecydujesz o tym świadomie i wyrazisz swoją decyzję w kodzie (np. używając atrybutu `dependsOnMethod` anotacji `@Test` z frameworka TestNG). Dużo gorzej jest jeżeli zależność między testami została wprowadzona nieświadomie.

Niech twoje testy będą niezależne. Jeżeli muszą już od siebie zależeć to bądź przynajniej tak miły i wyraź to jasno w kodzie!

Spójrzmy na poniższy przykład. Weryfikuje on czy kod konfigurujący logowanie przy pomocy Log4J[3] działa poprawnie. Oczekujemy, że w przypadku obecności zmiennej systemowej o nazwie `logConfig` powinien zostać użyty plik przez nią wskazywany. Jeżeli jednak ta zmienna nie istnieje, wówczas powinna zostać użyta konfiguracja domyślna.

[3]http://logging.apache.org/log4j/

```
LoggingPropertyConfigurator configurator
    = mock(LoggingPropertyConfigurator.class);
BaseServletContextListener baseServletContextListener
    = new BaseServletContextListener(configurator);

@Test
public void shouldLoadDefaultProperties() {
    baseServletContextListener.contextInitialized(null);
    verify(configurator).configure(any(Properties.class));
}

@Test(expected = LoggingInitialisationException.class)
public void shouldThrowLoggingException() {
    System.setProperty("logConfig", "nonExistingFile");
    baseServletContextListener.contextInitialized(null);
}
```

Ten test był zielony miesiącami. A potem, nagle i niespodziewanie zamienił sie na czerwony, po tym jak do repozytorium kodu trafił zupełnie niezwiązany commit. Ah, czemuż to? Przecież nie dotykaliśmy konfiguracji Log4J od niepamiętnych czasów…

Po dłuższej chwili dociekań doszliśmy do wniosku, że test zaczął się wysypywać, ponieważ zmieniła się kolejność wykonywania testów. Wszystko działało poprawnie dopóki najpierw wykonywała się metoda `shouldLoadDefaultProperties()` a dopiero potem metoda `shouldThrowLoggingException()`. Gdy tylko kolejność uległa zmianie, wówczas zaczęły się problemy. Wynikały one z tego, że w momencie wykonywania metody testowej `shouldLoadDefaultProperties` zmienna systemowa `logConfig` była już ustawiona. To oczywiście wpływało na zachowanie testowanego obiektu skutkiem czego test nie przechodził.

Można oczywiście zapytać dlaczego kolejność wykonywania testów uległa zmianie. Można, ale nie jest to istotne. Frameworki testowe domyślnie nie gwarantują kolejności wykonywania, więc nie można zakładać, że test X zawsze wykona się przed testem Y.

Zachowaj ostrożność modyfikując z poziomu testów współdzielone zasoby (ustawienia systemowe, system plików, bazy danych itp.). Może to wpłynąć na wykonanie innych testów. Upewnij się, że wiesz jaki jest stan początkowy środowiska, w którym odpalasz testy!

No dobrze, zatem zastanówmy się jak można poprawić ten test. Wydaje mi się, że są tu dwie możliwości.

Po pierwsze, możesz narzucić kolejność wykonywania testów (TestNG pozwoli ci na to, JUnit nie).

```
@Test
```

```
public void shouldLoadDefaultProperties() { ... }

@Test(expected = LoggingInitialisationException.class,
    dependesOnMethod = "shouldLoadDefaultProperties")
public void shouldThrowLoggingException() { ... }
```

Ten sposób zadziała tak długo, jak długo nie pojawią się inne metody testowe również zależne od wartości zmiennej systemowej `logConfig`. W takim przypadku będzie trzeba również określić kolejność ich wykonywania w stosunku do istniejących już metod. To niestety prędzej czy później doprowadzi do trudnej do utrzymania sieci zależności między testami. I z tego powodu nie polecam tego rozwiązania.

Innym sposobem, który wydaje mi się bardziej godny polecenia, jest czyszczenie wartości zmiennej systemowej `logConfig` przed wykonaniem metody testowej `shouldLoadDefaultProperties()`. Jeżeli metod zależnych od tej zmiennej systemowej jest więcej, wówczas warto przenieść ten kod do metody `setUp()`. Przykładowo:

```
@BeforeMethod
public void cleanSystemProperties() {
    System.setProperty("logConfig", null);
}

// reszta kodu bez zmian
```

Lepiej czyścić wszelkie ustawienia **przed** testami, a nie po nich. W ten sposób gwarantujesz sobie, że testy zawsze wykonają się w określonym środowisku.

4.2.2. Założenia odnośnie bazy danych

Spójrzmy na poniższy test. To test integracyjny, w którym `userService` i dao rozmawiają z bazą danych.

```
@Test
public void shouldAddUser() {
    User user = new User();
    userService.save(user);
    assertEquals(dao.getNbOfUsers(), 1);
}
```

Niby wszystko w porządku, a mnie jednak niepokoją dwie rzeczy:

1. test tak naprawdę nie weryfikuje, czy użytkownik został dodany do bazy,
2. test opiera sie na pewnych założeniach odnośnie stanu bazy danych.

Odnośnie punktu pierwszego, to z pewnością lepiej byłoby upewnić się, czy użytkownik z bazy danych jest identyczny jak obiekt `user`. Jeżeli `userService` albo `dao` dostarczają metod pozwalających wyciągnąć dane użytkownika z bazy (a powinny), to będzie to bardzo proste. Jeżeli nie, to będzie trzeba się trochę nagimnastykować (brr… ciarki przechodzą mnie na myśl pisania zapytań SQL czy kodu JDBC tylko na potrzeby testu).

Upewnij się, że metoda testowa robi to, co obiecuje jej nazwa.

Jeżeli o punkt drugi chodzi, to rozwiązaniem jest zastąpienie wartości bezwzględnych wartościami względnymi:

```
@Test
public void shouldAddUser() {
    int nb = dao.getNbOfUsers();
    User user = new User();
    userService.save(user);
    assertEquals(dao.getNbOfUsers(), nb + 1);
}
```

Wydaje się, że teraz ta metoda zaczyna faktycznie weryfikować to, co zapowiada jej nazwa (`shouldAddUser()`). I dobrze.

Tak się składa, że jakiś czas temu mój zespół miał sporo roboty właśnie z powodu problemów opisanych w tej sekcji. Otóż nasze testy zakładały, że w momencie ich uruchomienia w bazie danych nie ma ani jednego użytkownika. W trakcie prac nad projektem okazało się jednak, że potrzebny nam jest taki "udawany" użytkownik. Taki co to jest w bazie, ale nie wyświetla się na żadnych raportach i ogólnie zachowuje się jakby go nie było. Stworzyliśmy więc patcha na bazę danych, który dodawał go i… ah… Gdy uruchomiliśmy testy integracyjne ten patch oczywiście został nałożony na bazę danych i mnóstwo testów padło. A wszystko dlatego, że zakładaliśmy w testach, że tabelka z użytkownikami jest pusta.

Nie rób założeń co do zawartości bazy danych. Używaj raczej wartości względnych niż absolutnych.

Istnieje wiele sposobów na zapełnienie bazy danych danymi na potrzeby testów. Można wykorzystać do tego zapytania przez JDBC, użyć narzędzi typu DBUnit czy Flyway, warstwy serwisów w połączeniu z Hibernatem (jak to pokazano powyżej), i pewnie jeszcze wielu innych sposobów. Niezależnie od sposobu, w jaki będziesz

ustawiał bazę danych w stanie początkowym, i tak możesz wpędzić się w kłopoty, takie jak przedyskutowaliśmy powyżej.

4.3. Czas zawsze gra przeciwko nam

> Z naprawdę wielkich, posiadamy tylko jednego wroga - czas.
>
> — Joseph Conrad

Nigdy nie używaj `System.currentTimeMillis()` albo `new Date()` w kodzie produkcyjnym. Stwórz dodatkową warstwę abstrakcji - np. interfejs `TimeProvider` - którego domyślna implementacja zwróci aktualny czas. To pozwoli wygodnie testować funkcjonalności uzależnione od czasu.

Zobaczmy jak wielkie, straszne, przerażające i potworne zło powstaje gdy nie stosujemy się do powyższego zalecenia.

Oczywisty przypadek przedstawiam poniżej. Prawdę powiedziawszy nigdy nie spotkałem w kodzie czegoś tak odrażającego, ale zacznijmy od niego by zilustrować sedno problemu.

```
time = System.currentTimeMillis();

if (time.isAfter(5, PM)) { ❶
    ... do some afternoon activity
} else {
    ... do something else
}
```

❶ W API nie ma metody `isAfter()` ale chodzi tylko o zilustrowanie problemu.

W tym przypadku wszystko jest jasne. Nie kontrolując wartości zmiennej `time` nie jesteśmy w stanie przetestować wszystkich możliwych ścieżek. A nie mamy nad nią kontroli, bo przypisujemy jej wartość używając metody `System.currentTimeMillis()`, którą ciężko zmockować w testach przy pomocy cywilizowanych narzędzi.

Po tym wstępie chciałbym przedstawić inny przykład - co nieco bardziej interesujący, a przy tym prawdziwy. Przyjrzyjmy się takiemu oto fragmentowi kodu produkcyjnego (jest to część klasy `Util`):

```
public String getUrl(User user, String timestamp) {
    String name = user.getFullName();

    String url = baseUrl
        + "name="+URLEncoder.encode(name, "UTF-8")
        + "&timestamp="+timestamp;
    return url;
}
```

```
public String getUrl(User user) {
    Date date = new Date();
    Long time = (date.getTime() / 1000); //convert ms to seconds
    String timestamp = time.toString();
    return getUrl(user, timestamp);
}
```

Programista chciał przetestować drugą metodę `getUrl(User user)`. Wydaje się to banalne: wszak metoda ta zwraca `String`, który wystarczałoby porównać z oczekiwanym wynikiem. Niestety, ze względu na użycie metody `new Date()` do przygotowania jednego z elementów wynikowego obiektu typu `String`, programista nie wiedział do końca czego spodziewać się jako wyniku. Musiał być kreatywny… i udało mu się to w 100%! Oto test jaki napotkałem w kodzie:

```
@Test
public void shouldUseTimestampMethod() {
    //given
    Util util = new Util();
    Util spyUtil = Mockito.spy(util);

    //when
    spyUtil.getUrl(user);

    //then
    verify(spyUtil).getUrl(eq(user), anyString());
}
```

Wręcz trudno uwierzyć, że to nie jakiś żart. Ten kod jest zły, jest bardzo zły, i to z kilku powodów:

- kod testu testuje implementacje a nie zachowanie (patrz sekcja 4.6.2),
- testuje interakcje (`verify()`) zamiast zwracanych wyników (`assertThat()`),
- w użyciu jest "partial mocking" (użycie metody `Mockito.spy()` na prawdziwym obiekcie), co jest sygnałem ostrzegawczym i powinno budzić podejrzenia[4]).

A to wszystko tylko dlatego, że kod produkcyjny nie radzi sobie poprawnie z czasem!

Ciężko napisać test dla danego fragmentu kodu? Prawdopodobnie ten kod jest po prostu niedobry. Może lepiej najpierw go naprawić?

Okazuje się, że kiedy już przerobimy kod produkcyjny wprowadzając dodatkową warstwę abstrakcji, test staje się banalny, czytelny i oczywisty:

[4]Proszę przeczytaj odpowiednie fragmenty dokumentacji Mockito.

```
@Test
public void shouldAddTimestampToGeneratedUrl() {
    //given
    Util util = new ...
    TimeProvider timeProvider = mock(TimeProvider.class);
    when(timeProvider.getTime()).thenReturn("12345");
    util.set(timeProvider);

    //when
    String url = util.getUrl(user);

    //then
    assertThat(url).contains("timestamp=12345");
}
```

Zauważmy, że teraz metoda testowa nie weryfikuje żadnego zachowania (używa stubów a nie mocków).

Jeżeli testowana metoda zwraca jakieś wartości to powinno się ich użyć do weryfikacji poprawności jej działania. Mocków używaj tylko wtedy gdy naprawdę nie masz innego wyjścia.

I jeszcze jedno: taki kod testu **nigdy by nie powstał** gdyby deweloper najpierw napisał test!

Czy większy młotek jest lepszy?

W tej sekcji wspomniałem (między wierszami) że większość narzędzi do mockowania nie pozwoli mockować metod statycznych. I to nie dlatego, że nie da się tego zrobić. Owszem, da się. Powodem jest to, że autorzy tych narzędzi są przekonani, że takie działanie nie jest wskazane. Przykładowo Mockito pozwala mockować tylko to co łatwo mockowalne (np. wstrzyknięte przy pomocy mechanizmu Dependency Injection) i w ten sposób wymusza na nas pewien typ designu (który większość świata uważa za elegancki i poprawny). Odmawiając współpracy z pewnymi fragmentami kodu, narzędzia takie jak Mockito wysyłają nam sygnał, że coś jest z naszym kodem nie tak.

Oczywiście są narzędzia, które umożliwiają zmockowanie wszystkiego. Jednak ze względu kwestie opisane powyżej nie polecam ich stosowania.

P.S. Mogę sobie wyobrazić pewne przypadki, w których takie bardziej potężne narzędzia mogą okazać się bardzo pomocne. Nigdy jednak nie napotkałem na taką potrzebę pracując z nowym kodem!

4.4. Strata czasu

> Testuj wszystko co może nie zadziałać!
>
> — Extreme Programming Gurus

Są pewne fragmenty kodu, których nie warto pokrywać testami jednostkowymi. Serio, są takie. Najlepszymi przykładami są gettery, settery i delegatory. W większości przypadków i tak pokrywają je testy integracyjne i end-to-end, a poza tym szanse na to, że popełniliśmy błąd pisząc je, są minimalne.

Spójrzmy na poniższy przykładowy test, który (jak wszystko w tej książce), pochodzi z prawdziwego projektu.

```
@Test
public void shouldReturnImportantValue() {
    //given
    given(settings.getImportantValue()).willReturn(IMPORTANT_VALUE);

    //when
    BigDecimal importantValue = settingsFacade.getImportantValue();

    //then
    assertThat(importantValue).isEqualByComparingTo(IMPORTANT_VALUE);
}
```

Ten test dotyczy dwóch klas: klasy `SettingsFacade` i jej współpracownika, obiektu klasy `Settings`. Z testu wynika, że `SettingsFacade` jest delegatorem, tj. przekierowuje zapytania do swojego współpracownika. I faktycznie tak jest, o czym może przekonać nas rzut oka na testowaną metodę `getImportantValue()`[5]

```
public BigDecimal getImportantValue() {
    return settings.getImportantValue();
}
```

Patrząc na tę metodę zastanawiam się jakież to potworne i przerażające błędy mogą się w niej kryć? I nieważne jak długo się nad tym zastanawiam, zawsze dochodzę do wniosku, że **żadne**. …więc po co ten test?

Czasami słyszę, że prawdziwym powodem pisania tego typu testów jest przekonanie o tym, że gdy kod będzie ewoluował, wówczas ten test uchroni nas przed problemami pozwalając nam stwierdzić czy dotychczasowa funkcjonalność nadal działa. Zgadzam się z pierwszą częścią tego twierdzenia: kod ewoluuje, i jest możliwe że ta metoda też zmieni swój kształt. Niestety, pisanie tego typu testów nic nam nie pomoże.

[5]Metoda delegująca to taka, która nie robi nic więcej jak tylko przekazuje pracę komuś innemu.

Tego typu defensywne pisanie testów po pierwsze łamie zasadę YAGNI[6]. Oczywiście nie jestem w stanie podać żadnych statystyk, ale powiedziałbym, że zdecydowana większość (99%?) tak prostych metod nie ulegnie zmianom. Co oznacza, że te 99 razy na 100 tracimy czas pisząc testy. Co więcej, każdy szanujący się programista rozumie, że modyfikując logikę metody powinien dokonać odpowiednich zmian w testach. Po trzecie wreszcie twierdzę, że nawet gdyby testy jednostkowe przeoczyły zmiany w tej metodzie, to z pewnością wyłapią to testy wyższego rzędu. A po czwarte jestem przekonany, że napisanie takiego testu nie jest nas w stanie przed niczym zabezpieczyć. Rzecz sprowadza się do tego, że nie potrafimy przewidzieć przyszłej ewolucji kodu. Zastanówmy się co będzie, jeżeli omawiany tu fragment ulegnie następującej zmianie:

```
public BigDecimal getImportantValue() {
    return (settings.getImportantValue() != null)
        ? settings.getImportantValue() : DEFAULT_VALUE;
}
```

Wydaje się, że zaprezentowany kilka akapitów wyżej test przejdzie, czyż nie? Otóż to! Istniejący test jest bezradny wobec niektórych zmian.

Pisz testy jednostkowe dla fragmentów kodu, które wykazują przynajmniej minimalny stopień komplikacji. Nie marnuj czasu na testowanie tego, co złamać się nie może.

99% kodu jaki piszesz nie jest aż tak proste! Więc nie ma żadnej wymówki: testy jednostkowe mają być i basta!

4.5. Testy, które wiedzą za dużo

W tej sekcji zajmiemy się bardzo rozpowszechnioną plagą trapiącą setki testów. Jest nią mianowicie obfitość szczegółów wyspecyfikowanych w kodzie testowym, które tak naprawde

[6]See http://en.wikipedia.org/wiki/You_ain%27t_gonna_need_it

nie są istotne dla testowanego scenariusza. Taki przerost formy nad treścią nie tylko obniża czytelność testów, ale również czyni je trudniejszymi w utrzymaniu zwiększając liczbę rzeczy, których zmiana wymusi zmiany w kodzie testu.

4.5.1. Mockowanie to nie małpowanie

Zamieszczony poniżej kod, który pochodzi z listy dyskusyjnej Mockito, łamie jedną z zasad dobrego mockowania: *"mockuj tylko własne typy"*. A przy okazji bardzo wiernie odwzorowuje szczegóły kodu produkcyjnego (piszę te słowa zgadując, że tak właśnie jest - ale idę o zakład, że ma to miejsce!). I nie jest to właściwe podejście. Kod testowy jest tak ściśle związany z implementacją, że nie można wprowadzić nawet najdrobniejszej zmiany do kodu produkcyjnego by go nie popsuć.

A co właściwie robi ten test, co weryfikuje? Wydaje się, że wszystko i nic jednocześnie. Weryfikuje wszystkie interakcje zachodzące pomiędzy SUT a współpracownikami. A nawet więcej: na miejsce SUT wprowadza stuba, a potem sprawdza czy SUT został prawidłowo zastubowany! To doprawdy bezcelowe, no chyba, że ktoś chce sprawdzić czy Mockito działa poprawnie (chyba nie warto tego robić: zrobiło to już tysiące programistów korzystających z Mockito na co dzień).

```
@Mock private DataSource dataSource;

@Mock private Mock connection;

@Mock private Mock statement;

@Mock private ResultSet resultSet;

@Test
public void test() throws Exception {
    MockitoAnnotations.initMocks(this);
    systemUnderTest = new OracleDAOImpl();
    systemUnderTest.setDBConnectionManager(connectionManager);
    Set<NACustomerDTO> set = new HashSet<NACustomerDTO>();
    when(connectionManager.getDataSource()).thenReturn(dataSource);
    when(dataSource.getConnection()).thenReturn(connection);
    when(connection.createStatement()).thenReturn(statement);
    when(statement.executeQuery(anyString())).thenReturn(resultSet);
    when(resultSet.next()).thenReturn(false);
    when(resultSet.getLong(1)).thenReturn(1L);
    when(resultSet.getString(2)).thenReturn("7178");

    doNothing().when(resultSet).close();

    stub(systemUnderTest.getNACustomers()).toReturn(set); ❶
    final Set<NACustomerDTO> result = systemUnderTest.getNACustomers();
```

```
    verify(connectionManager).getDataSource();
    verify(dataSource).getConnection();
    verify(connection).createStatement();
    verify(statement).executeQuery(anyString());
    verify(resultSet).next();
    verify(resultSet).getLong(1);
    verify(resultSet).getString(2);

    assertNotNull(result); ❷

    verify(connectionManager).getDataSource().getConnection();
}
```

❶ Tu zastępujemy SUT stubem.
❷ A tu weryfikujemy czy poprawnie zastąpiliśmy SUT stubem.

Pisanie testu nie polega na odwzorowywaniu kodu produkcyjnego linia po linii!

Łatwo się znęcać nad tym testem, ale lepiej zastanówmy się co można by tu zrobić lepiej. Pozwolę sobie na kilka słów komentarze odnośnie testowania warstwy DAO.

Można przyjąć, że klas DAO nie warto testować testami jednostkowymi. Dlaczego? Ponieważ zazwyczaj nie posiadają żadnej logiki (DAO z definicji powinno być cienką warstwą odzielającą nasz kod od kod dostarczonego przez silnik bazy danych). Zamiast testów jednostkowych, dużo skuteczniej jest testować DAO testami integracyjnymi. Takie testy pozwolą sprawdzić, że nasz kod faktycznie dobrze współpracuje z bazą danych. Jeżeli nasze DAO nie ma żadnych elementów specyficznych dla danego dostawcy rozwiązania bazodanowego, wówczas można użyć łatwej w użyciu bazy danych - np. H2[7]. Testowanie może polegać na sprawdzeniu czy encje zapisane i odczytane z bazy przez DAO są takie, jak oczekiwaliśmy. Testując warto skupić się na co bardziej skomplikowanych obszarach (o ile warstwa DAO w ogóle takowe posiada - np. na parametryzowanych zapytaniach). Pisanie testów dla funkcjonalności CRUD (Create-Read-Update-Delete), które załatwia za nas framework ORM, wydaje się być słabo uzasadnione (jeżeli są tam jakieś błędy, co wydaje się wątpliwe, to z pewnością wykryją je testy typu end-to-end).

Polecam przeszukanie zasobów http://stackoverflow.com pod kątem *"dao testing"* – z pewnością znajdziesz tam masę użytecznych informacji.

[7] http://www.h2database.com

4.5.2. Przyczyna wszelkiego zła

Autorem tej sekcji jest Jakub Nabrdalik (http://blog.solidcraft.eu/). Na polski przetłumaczył Tomek Kaczanowski.

Zbyt szczegółowe testy są często źródłem wszelkiego zła. Popełnione w nich błędy skutkują opłakanymi w skutkach i długotrwałymi konsekwencjami. Nim się tego nauczyłem przez blisko dwa lata tworzyłem zbyt szczegółowe testy pewnego frameworku, aż w końcu okazało się, że kompletnie uniemożliwiają one jakikolwiek refaktoring. Jak to możliwe? Przyjrzyjmy się przykładowi w języku Groovy[8] i frameworku Spock[9].

> **Spock**
>
> Jeżeli nie miałeś okazji używać Spocka to warto byś wiedział, że `n*mock.whatever()` oznacza, że oczekujemy dokładnie `n` wywołań metody `whatever()` mockowanego obiektu. Ani mniej, ani więcej. Podkreślenie _ oznacza "wszystko" albo "cokolwiek". A znak >> nakazuje frameworkowi testowemu zwrócić wartość wyrażenia po prawej stronie gdy zostanie wywołana określona metoda. Kod Spocka jest na tyle czytelny, że po tym wstępie z całą resztą konstrukcji z pewnością już sobie poradzisz.

Rozpocznijmy od testu.

```
def "should create outlet insert command with valid params with new account"() {
    given:
        def defaultParams = OutletFactory.validOutletParams
        defaultParams.remove('mobileMoneyAccountNumber')
        defaultParams.remove('accountType')
        defaultParams.put('merchant.id', merchant.id)
        controller.params.putAll(defaultParams)

    when:
        controller.save()

    then:
        1 * securityServiceMock.getCurrentlyLoggedUser() >> user
        1 * commandNotificationServiceMock.notifyAccepters(_)
        0 * _._
        Outlet.count() == 0
        OutletInsertCommand.count() == 1
        def savedCommand = OutletInsertCommand.get(1)
        savedCommand.mobileMoneyAccountNumber == '1000000000000'
```

[8] http://groovy.codehaus.org
[9] http://spockframework.org

```
        savedCommand.accountType == CyclosAccountType.NOT_AGENT
        controller.flash.message != null
        response.redirectedUrl == '/outlet/list'
}
```

Zauważyłeś, że część *"then"* testu jest naprawdę duża. Jeden z programistów podczas code review zauważył, że ten test jest tak bogaty w szczegóły, że zdaje się być raczej przypadkiem "reverse engineering" obiektu SUT niż prawdziwym testem. W zasadzie w kodzie testowym powtarzamy każdą linię (a już na pewno każdy blok) kodu klasy, którą testujemy. Różnica leży tylko w składni. Jeżeli przetestujemy w ten sposób cały system – którego budowa pochłonęła wiele osobo-miesięcy lub nawet osob-lat – to nigdy nie będziemy w stanie zmienić nawet jednej linijki kodu nie łamiąc przy okazji jakiegoś testu. A ponieważ w tym teście bardzo skrupulatnie weryfikujemy wszelką współpracę obiektów, to można spodziewać się, że przy każdym prostym refaktoringu będziemy psuć setki testów. Na własne oczy widziałem takie sytuacje.

Przyjrzyjmy się teraz z bliska fragmentom testu by przekonać się co jest z nim nie tak. Oto pierwsza linia części *"then"*:

```
1 * securityServiceMock.getCurrentlyLoggedUser() >> user
```

Ta linia sprawdza czy została wywołana pewna metoda serwisu odpowiedzialnego za bezpieczeństwo i czy zwróciła ona obiekt `user`. Zweryfikowaniu podlega też liczba wywołań: oczekiwane jest dokładnie jedno. Ni mniej, ni więcej.

Ale to wcale nie jest to, co chcemy sprawdzić. Być może tego wymaga implementacja kontrolera (testowanej klasy), ale to znaczy tylko tyle, że powinno się to znaleźć w części *"given"*. I nie powinniśmy sprawdzać czy wywołanie tej metody nastąpiło dokładnie raz. Matko jedyna, przecież to jest stub! Użytkownik albo jest zalogowany, albo nie. Nie ma sensu robić z niego użytkownika *"zalogowanego, ale tylko jednokrotnie"*.

Ważne jest by oddzielić warunki wstępne od wyników.

To teraz zajmijmy się kolejną linią testu.

```
1 * commandNotificationServiceMock.notifyAccepters(_)
```

Ta linia sprawdza czy pewna metoda serwera notyfikacji jest wywoływana jednokrotnie. I być może to ma sens: logika biznesowa może tego wymagać. Ale skoro tak, to czemu nazwa testu nie zawiera nic na ten temat? Ah, już wiem, bo wtedy byłaby zbyt długa. Cóż, to też powinno nam coś powiedzieć. Zdaje się, że potrzebny jest osobny test: coś w stylu *"should notify about newly created outlet insert command"*.

Napotykamy tu inny problem ze zbyt szczegółowym testami: zazwyczaj grupują kilka biznesowych wymagań pod wspólną nazwą. I później nie jest łatwo dokopać się do nich. Zobaczymy więcej przykładów tego problemu w kolejnych analizowanych liniach kodu.

A oto trzecia linia.

```
0 * _._
```

Praktycznie każdy framework do mocków pozwala sprawdzić, że *"nie zaszło nic więcej, poza tym co już wspominałem"*. Spock nie stanowi tu wyjątku. Na wypadek gdybyś jeszcze tego sam nie wykombinował, to powiem, że ta właśnie linia oznacza: *"Nie będziesz miał w swoim kodzie żadnych więcej interakcji z mockami, stubami ani niczym innym. Amen!"*. Moja sugestia jest następująca: nigdy, przenigdy nie używaj tego typu asercji. Sprawia ona, że test jest wrażliwy na wszelkie zmiany. Każda zmiana w kodzie dotycząca współpracy testowanej klasy z jej współpracownikami będzie łamała taki test. A test powinien padać tylko jeżeli zmieniła się logika biznesowa albo jeżeli złamany został kontrakt.

Dostarczana przez Mockito metod `verifyNoMoreInteractions()` także powinna być używana z rozwagą. Jak zaznaczono w dokumentacji Mockito: *"Używać tylko gdy w razie konieczności. Nadużywanie prowadzi do tworzenia testów zbyt szczegółowych i trudniejszych w utrzymaniu."*. Co w skrócie oznacza, że powinniśmy używać tej metody tylko jeżeli naprawdę zależy nam na stwierdzeniu, że nie nastąpiły już kolejne interakcje z danym mockiem.

A potem kolejna linia.

```
Outlet.count() == 0
```

Ten fragment sprawdza czy nie mamy żadnych outletów w bazie. Czy wiesz czemu to robi? Nie wiesz, bo nazwa metody testowej nic na ten temat nie mówi. Ja wiem, bo akurat znam logikę biznesową tej domeny. Ten test nie przedstawia jasno logiki biznesowej. To typowy przykład ukrywania wymagań biznesowych pod niewłaściwą nazwą, co sprawia, że tracimy szansę przedstawienia wymagań biznesowych w postaci zbioru testów (czy też specyfikacji).

W końcu następuje część, która wydaje się odnosić do nazwy testu.

```
OutletInsertCommand.count() == 1
def savedCommand = OutletInsertCommand.get(1)
savedCommand.mobileMoneyAccountNumber == '1000000000000'
savedCommand.accountType == CyclosAccountType.NOT_AGENT
```

W tej części wyrażamy nasze oczekiwania co do stworzonego obiektu. Oczekujemy, że będzie on "nowy", co w tym przypadku oznacza określony numer konta i typ. Ten blok kodu aż prosi się o to by go przenieść do osobnej metody opatrzonej odpowiednio czytelną nazwą.

A potem widzimy taką linię kodu:

```
controller.flash.message != null
```

```
response.redirectedUrl == '/outlet/list'
```

Oczekujemy, że pojawi się jakiś komunikat (`flash message`). I że nastąpi przekierowanie. A czemu testujemy te dwie rzeczy? Z pewnoscią nie dlatego, że tak mówi nazwa metody testowej. Prawda jest taka, że ani zawartość komunikatu, ani adres przekierowania nie powinny nas obchodzić.
A co, jeżeli te dwie rzeczy są określone w wymaganiach?

Hm, czy naprawdę są tam określone? Jeżeli wymagania są tak szczegółowe, że określają gdzie przekierowujemy użytkownika po wykonaniu określonej czynności, to lepiej zrobimy weryfikując to przy pomocy testu funkcjonalnego (webowego). Test taki jak powyżej nie powie nam czy użytkownik po przekierowaniu zobaczy listę outletów. Przejście na weryfikowany URL `/outlet/list` równie dobrze może zakończyć się błędem HTTP 404.

I wydaje mi się bardzo wątpliwe żeby wymagania specyfikowały, że po zakończeniu akcji *"nie należy wyświetlać komunikatu"*.

Te dwie linie tak naprawdę nie weryfikują niczego: dodają tylko kolejne szczegóły do testu, co jeszcze bardziej utrudni jego późniejszy refaktor.

I tak oto dochodzimy do konkluzji: **Nie pisz zbyt szczegółowych testów!** Sprawdzaj tylko to co należy i to tylko jedną rzecz na raz! Niech testy wyrażają wyłącznie wymagania biznesowe. Jeżeli testy nie przejdą to niech przyczyną ich niepowodzenia będą wyłącznie kwestie biznesowe!

Nigdy, przenigdy nie weryfikuj algorytmu metody krok po kroku! Weryfikuj wynik algorytmu! Powinieneś mieć swobodę zmiany implementacji metody, tak długo jak wynik – to czego naprawdę oczekujesz – nie ulegnie zmianie.

Wyobraź sobie problem związany z sortowaniem. Skupiałby się na sprawdzeniu poszczególnych kroków algorytmu, czy raczej sprawdzałbyś, że wynikowa kolekcja jest posortowana? Niby dlaczego miałbyś sprawdzać algorytm? Powinieneś mieć swobodę zmieniania go. Test nie powinien w tym przeszkadzać.

Zastanówmy się jak powinien wyglądać powyższy kod. Zauważyliśmy już, że mamy w nim do czynienia z trzema wymaganiami biznesowymi, które warto zweryfikować.

1. Kiedy tworzymy outlet, oczekujemy że obiekt klasy `OutletInsertCommand` znajdzie się w bazie danych, ale w systemie nie pojawi się jeszcze `Outlet` (zmiana musi zostać jeszcze zaakceptowana przez kierownika). Sprawdzenie odbywa się w tych dwóch liniach:

   ```
   Outlet.count() == 0
   ```

```
OutletInsertCommand.count() == 1
```

2. Kierownik (zwany tutaj "acceptorem") powinien zostać poinformowany o nowej komendzie. I to również jest sprawdzane:

```
1 * commandNotificationServiceMock.notifyAcceptors(_)
```

3. Komenda powinna mieć "nowe konto", którego stan wyrażać się ma w następujący sposób:

```
def savedCommand = OutletInsertCommand.get(1)
savedCommand.mobileMoneyAccountNumber == '1000000000000'
savedCommand.accountType == CyclosAccountType.NOT_AGENT
```

Cóż więc powinniśmy zrobić? Czy wystarczy wykasować wszystkie pozostałe linie testu pozostawiając tylko te wymienione powyżej. Nie, nie całkiem.

Gdybyśmy pisali ten test przed kodem produkcyjnym, to pewnie każde z tych wymagań zawarlibyśmy w formie osobnego testu rozpoczynając od nazwy metody testowej. I byłoby to dużo lepsze rozwiązanie. Gdyby zdarzyło się, że nasz kod nie spełnia któregoś z wymagań biznesowych, to od razu wiedzielibyśmy którego. Innymi słowy, lepiej gdy test posiada tylko jedną biznesową przyczynę ewentualnej porażki.

Tak więc spróbujmy teraz napisać ten test raz jeszcze. Tym razem poprawnie.

```
def setup() { ❶
    userIsLoggedIn()
}

private void userIsLoggedIn() {
    securityServiceMock.getCurrentlyLoggedUser() >> user
}
```

❶ Spock automatycznie rozpoznaje metodę `setUp()` i uruchamia ją przed każdą metodą testową.

W ten sposób załatwiliśmy niezbyt istotne warunki wstępne. Nie ma potrzeby umieszczać ich w metodach testowych i powiększać tam zamieszania.

Wiedząc już, że mamy do czynienia z trzema wymaganiami biznesowymi stwórzmy metodę, która przygotuje grunt pod wszystkie testy. Nadamy jej znaczącą nazwę i będziemy ją wywoływać z każdej metody testowej, jako że odnosi się ona bezpośrednio do danej części testowanego scenariusza – to będzie tak, jakbyśmy pisali test w stylu BDD.

```
private void setValidOutletInsertCommandParameters(def controller) {
    def validParams = OutletFactory.validOutletParams
    validParams.remove('mobileMoneyAccountNumber')
    validParams.remove('accountType')
```

```
    validParams.put('merchant.id', merchant.id)
    controller.params.putAll(validParams)
}
```

Teraz możemy już stworzyć testy dla każdego z wymagań biznesowych:

```
def "created outlet insert command should have new account"() {
    given:
        setValidOutletInsertCommandParameters(controller)

    when:
        controller.save()

    then:
        outletInsertCommandHasNewAccount()
}

private boolean outletInsertCommandHasNewAccount() {
    def savedCommand = OutletInsertCommand.get(1)
    savedCommand.mobileMoneyAccountNumber == '1000000000000' &&
        savedCommand.accountType == CyclosAccountType.NOT_AGENT
}

def "should not create outlet, when creating outlet insert command"() {
    given:
        setValidOutletInsertCommandParameters(controller)

    when:
        controller.save()

    then:
        Outlet.count() == 0
        OutletInsertCommand.count() == 1
}

def "should notify acceptors when creating outlet insert command"() {
    given:
        setValidOutletInsertCommandParameters(controller)

    when:
        controller.save()

    then:
        1 * commandNotificationServiceMock.notifyAccepters(_)
}
```

Teraz każdy z testów nie przejdzie tylko jeżeli naprawdę musi. Rozdzielenie wymagań między trzy testy zapewnia nam dobry feedback. Dodatkowo, dzięki uniknięciu wpychania w testy niepotrzebnych szczegółów możemy dokonywać w kodzie dowolnych zmian.

4.5.3. Kopiuj i wklej

Tym razem na warsztat weźmiemy test mojego autorstwa. Jego zadaniem jest sprawdzenie czy pewna klasa obsługująca artefakty Mavena potrafi rozpoznać snapshoty.

```
@DataProvider
public Object[][] snapshotArtifacts() {
    return new Object[][]{
        {"a", "b", "2.2-SNAPSHOT", Artifact.JAR },
        {"c", "d", "2.2.4.6-SNAPSHOT", Artifact.JAR},
        {"e", "f", "2-SNAPSHOT", Artifact.JAR}
    };
}

@Test(dataProvider = "snapshotArtifacts")
public void shouldRecognizeSnapshots(
        String groupId, String artifactId,
        String version, Type type) {
    Artifact artifact
        = new Artifact(groupId, artifactId, version, type);
    assertThat(artifact.isSnapshot()).isTrue();
}
```

Wszystko gra, ale… Gdy się tak przyjrzeć uważniej temu, co ten test tak właściwie weryfikuje, to okaże się, że tylko jeden z czterech parametrów przekazywanych przez metodę `snapshotArtifacts()` ma tak naprawdę znaczenie. Po co nam w ogóle `groupId`, `artifactId` i `type`, jeżeli tak naprawdę interesuje nas tylko wersja artefaktu?

Po drobnej zmianie ten sam test może przyjąć następującą postać:

```
@DataProvider
public Object[][] snapshotVersions() {
    return new Object[][]{
        {"2.2-SNAPSHOT"},
        {"2.2.4.6-SNAPSHOT"},
        {"2-SNAPSHOT"}
    };
}

@Test(dataProvider = "snapshotVersions")
public void shouldRecognizeSnapshots(String version) {
    Artifact artifact
        = new Artifact(VALID_GROUP, VALID_ARTIFACT_ID,
        version, VALID_TYPE);
    assertThat(artifact.isSnapshot()).isTrue();
}
```

Zdecydowałem się użyć wielu wartości statycznych (takich jak `VALID_GROUP`). Nadałem im bardzo opisowe nazwy, tak by rzut oka na nie mówił czytelnikowi kodu, że są to wartości poprawne, a przy tym zupełnie nieistotne dla testowanego scenariusza.

Jak wspomniałem jest to test mojego autorstwa. Po tym jak go już naprawiłem zadałem sobie pytanie jak w ogóle mogłem coś takiego napisać. Szybko doszedłem do wniosku, że wszystko zaczęło się od skopiowania metody dostarczającej dane. W poprzednim teście wszystkie cztery parametry miały znaczenie. Skopiowałem je wszystkie przez co niepotrzebnie skomplikowałem test. Lekcja do zapamiętania: człowieku, nie kopiuj bezmyślnie!

Można też nieco inaczej podejść do zagadnienia i użyć wzorca *test data builder* (patrz sekcja 5.3). Efekt będzie podobny do poniższego:

```
Artifact artifact = new ArtifactBuilder()
    .validArtifact()
    .withVersion(version)
    .build();
```

Jest to prawdodpobonie rzecz gustu, ale akurat w tym przypadku bardziej odpowiada mi rozwiązanie ze `STAŁYMI`. Wydaje mi się bardziej czytelne.

Więcej narzekania na kopiuj i wklej

To normalne, że kopiujemy fragmenty kodu testowego by uniknąć pisania w kółko tych samych (lub bardzo podobnych linii kodu). Nie winię za to ani siebie, ani innych. Problem pojawia się, gdy robimy to bezmyślnie.

W przypadku testów metoda kopiuj i wklej prowadzi do:

- przerośniętych części tworzących obiekty - kopiowanie prowadzi do tworzenia obiektów z wypełnionymi polami zupełnie nieistotnymi w danym scenariuszu testowym,
- zbyt wielu asercji weryfikujących kwestie niezwiązane z danym przypadkiem testowym.

W obu przypadkach niepotrzebne elementy zaciemniają test utrudniając jego zrozumienie, a dodatkowo czynią go bardziej kruchym.

4.5.4. Zbyt szczegółowe asercje

Kolejny test godny naszej uwagi! Przedstawiam tylko jego końcówkę, ale uwierz mi proszę: to był naprawde skomplikowany test end-to-end. Testowaniu podlegał cały system, który otrzymywał żądania i odpowiadał w formie pliku CSV.

```
@Test
public void invalidTxShouldBeCanceled() {
    ... some complex test here

    // then
    String fileContent =
        FileUtils.getContentOfFile("response.csv");
    assertTrue(fileContent.contains(
        "CANCEL,123,123cancel,billing_id_123_cancel,SUCCESS,"));
}
```

Przyjrzyjmy się nazwie metody - `invalidTxShouldBeCanceled()` (jak można się domyśleć *tx* to skrót od *transakcji*) - i kończącej test asercji. Czy one w ogóle do siebie pasują? Czy czytając ostatnią linię masz poczucie, że sprawdza ona że *"transakcja została anulowana"*? Cóż, być może tak jest, ale pewnym jest to, że oprócz tego ten test sprawdza również, że wygenerowana odpowiedź jest zgodna z pewnym formatem.

Sugerowałbym nieco inny zapis ostatniej linii z wykorzystaniem własnych asercji (patrz sekcja 5.6).

```
@Test
public void invalidTxShouldBeCanceled() {
    ... some complex test here

    // then
    TxDTOAssert.assertThat("response.csv")
        .hasTransaction("123cancel").withResultCode(SUCCESS);
}
```

Ta wersja odpowiada mi o wiele bardziej. Szczegóły implementacyjne zostały schowane. Wewnątrz asercji ukryta jest logika, która parsuje plik CSV i wyciąga z niego status transakcji. Test stał się dzięki temu bardzo czytelny: *sprawdź, że w pliku odpowiedzi znajduje się transakcja* `123cancel`*, wraz z informacją, że anulowanie się powiodło (status:* `SUCCESS`*)"*.

Oczywiście jeżeli zmieni się format wysyłanej odpowiedzi wówczas będziemy musieli uaktualnić wyłącznie kod wewnątrz klasy `TxDTOAssert`. W innym przypadku musielibyśmy dokonywać zmian w wszystkich (być może bardzo wielu) metodach testowych weryfikujących zawartość pliku CSV.

assertThat()

Petri Kainulainen zwrócił mi uwagę na to, że użycie w teście nazwy klasy asercji (w naszym przypadku `TxDTOAssert`) co nieco zaciemnia test. Oczywiście konieczność podania klasy w kodzie wynika z konfliktu nazw metody `assertThat()`, która to metoda występuje zarówno w klasie `Assertions` frameworku AssertJ jak i w klasie naszej własnej asercji. Jeżeli czujesz że jest to problem, to warto pamiętać o tym, że można przecież w klasie asercji `TxDTOAssert` użyć innej nazwy, np. `assertThatFile()`. Wówczas można pozbyć się prefixu z nazwą klasy, a kod testu będzie wyglądał następująco:

```
@Test
public void invalidTxShouldBeCanceled() {
    ... some complex test here

    // then
    assertThatFile("response.csv")
        .hasTransaction("123cancel").withResultCode(SUCCESS);
}
```

4.5.5. Test pełen szczegółów

Dlaczego testy są tak nieodporne na zmiany? Nierzadko dlatego, że zupełnie niepotrzebnie umieszczamy w nich mnóstwo szczegółów! Spójrzmy na poniższy przykład pochodzący z pewnej listy dyskusyjnej (musiałem go mocno skrócić by zmieścił się w książce):

```
public void createSurvey() throws InterruptedException {
    //CREATE SURVEY
    WebElement allproject
        = driver.findElement(By.xpath("//*[@id='projectnav']/ul/li[2]/a"));
    allproject.click();

    WebElement myfolder
        = driver.findElement(By.linkText("John Doe"));
    myfolder.click();

    WebElement myProject
        = driver.findElement(By.linkText("My project"));
    myProject.click();

    WebElement createsurveylink = driver.findElement(By.xpath(
        "//*[@id='bcontrol']/body/form[1]/table[2]/tbody/tr/td[2]/a[1]/img"));
    createsurveylink.click();
```

```
    WebElement surveyname = driver.findElement(By.xpath(
        "//*[@id='bcontrol']/body/form/table[4]/tbody/tr[2]/td[3]/input"));
    surveyname.sendKeys("Test Survey created on " + new Date());

    WebElement surveynameconfirm = driver.findElement(By.xpath(
        "//*[@id='bcontrol']/body/form/table[1]/tbody/tr[2]/td[3]/a[2]/img"));
    surveynameconfirm.click();
}
```

Podstawowy problem z powyższym kodem polega na tym, że opisuje on w najdrobniejszych szczegółach rozmieszczenie elementów na stronie WWW. Każda, nawet najdrobniejsza modyfikacja struktury strony spowoduje, że ten test przestanie przechodzić. A przecież pewnym jest, że **strona będzie się zmieniać**! Termin przydatności do użycia zastosowanych w teście wyrażenia XPath wydaje mi się bardzo krótki.

Podejrzewam, że jest to tylko jeden z wielu testów napisanych w ten sposób. To oznacza, że informacja o szczegółach strony WWW występuje wielokrotnie w licznych testach. Jeżeli tak jest, to każda zmiana na stronie spowoduje konieczność modyfikacji ich wszystkich!

Spróbujmy jakoś poprawić ten test. Jedną z możliwości jest poprawienie struktury strony, w taki sposób by zlikwidować bardzo kruche wyrażenia XPath. W tym celu można na przykład przypisać identyfikatory istotnym elementom strony i później odwoływać się do nich w teście. To pomoże, ale nie do końca. Testy wciąż będą świadome szczegółów strony i wyczytanie z nich scenariusza testowego będzie wymagać przebicia się przez gąszcz nieistotnych detali implementacyjnych.

Tak naprawdę powinniśmy spróbować uwolnić kod testowy od wszelkich szczegółów na temat strony WWW. Testy end-to-end powinny zajmować się biznesowym scenariuszem i abstrahować od szczegółów!

W świecie testów Selenium (WebDriver) wzorzec projektowy, o którym mówię, nazywa się *Page Objects*[10]. Przerabiając test zgodnie z tym wzorcem, uzyskamy następujący efekt:

```
public void shouldCreateSurvey() {
    Date date = new Date();

    ProjectsPage projectsPage = mainDashboard.goToProjectsPage();

    projectsPage.openProject("My project");

    SurverEditionPage surverEditionPage
        = projectsPage.createSurvey("Test Survery created on " + date);
```

[10]Strona http://code.google.com/p/selenium/wiki/PageObjects dostarczy więcej informacji na ten temat.

```
    // there were no assertions in the original tests
    // (which was rather weird...),
    // but I guess something like this would make sense
    String surveyName = surveyEditionPage.getEditedSurveyName();

    assertThat(surveyName).isEqualTo("Test Survey created on " + date);
}
```

Spójrzmy co udało się uzyskać:

- większość "niskopoziomowej" pracy wykonuję obiekty reprezentujące różne strony (`projectsPage` i `surverEditionPage`) - to w nich zawarte są szczegóły takie jak wyrażenia XPath,
- test jest czytelny, w tym sensie, że bardzo łatwo można zrozumieć przypadek testowy jaki on weryfikuje; nie trzeba w tym celu wiedzieć nic o żadnych wyrażeniach XPath,
- kod testu przetrwa nawet poważne zmiany na stronie, tak długo jak nie zmieni się cała koncepcja wypełniania formularzy w serwisie.

To wcale nie oznacza, że te niskopoziomowe szczegóły magicznie zniknęły. Nie, nie poszły sobie. Tylko że teraz siedzą w obiektach klas typu `xyzPage`. Zysk z tej operacji jest taki, że testy są czytelne a szczegółowe informacje są zgromadzone w jednym miejscu. Przykładowo, wiele scenariuszy testowych będzie odwoływać się do strony z listą projektów reprezentowanej przez klasę `ProjectsPage`. Zmiana w ułożeniu na stronie wymusi na programiście zmianę wyłącznie w klasie ją reprezentującej.

Zawsze dobrze jest oddzielać to co zmienia się rzadko (scenariusz biznesowy), od tego co zmienia się często (ułożenie elementów na stronie WWW).

4.6. Jedna rzecz, za to porządnie

> Rzeczy do wszystkiego są do niczego.
>
> — Mądrość Babci Aliny

Znamy wszyscy zasadę SRP[11], która w skrócie mówi tyle, że każda klasa powinna zajmować się robieniem jednej rzeczy. Zachęcam do tego, by podobnie myśleć o testach. Niech każdy z nich stosuje się do prostej zasady: *"Każda metoda testowa powinna weryfikować dokładnie jeden scenariusz"*.

[11] SRP, see http://en.wikipedia.org/wiki/Single_responsibility_principle

Jaki ma to sens? Powody są dwa:

- łatwo zrozumieć takie metody testowe,
- jeżeli dany test nie przejdzie, wiemy **dokładnie** która z funkcjonalności systemu nie działa (co jest szczególnie cenne gdy wprowadzamy zmiany do istniejącego kodu).

Na poziomie testów jednostkowych łatwo jest stosować się do zasady SRP. W przypadku testów integracyjnych czy end-to-end może być to nie tylko trudniejsze, ale i nie zawsze pożądane. W wielu przypadkach biorąc pod uwagę kosztowny setup testu warto zweryfikować więcej niż jedną rzecz w pojedynczej metodzie testowej.

Wielokrotnie zauważałem w kodzie testowym jak zasada SRP jest łamana i to bez wyraźnego uzasadnienia. Postaram się omówić ten temat posiłkując się przykładami.

4.6.1. Prawidłowe i nieprawidłowe

Spójrzmy na przykład testu klasy pomocniczej, które zadaniem była weryfikacja prefiksów numerów telefonicznych.

```
@DataProvider
public Object[][] data() {
    return new Object[][] { {"48", true}, {"+48", true},
        {"++48", true}, {"+48503", true}, {"+4", false},
        {"++4", false}, {"", false},
        {null, false}, {"  ", false}, };
}

@Test(dataProvider = "data")
public void testQueryVerification(String query, boolean expected) {
    assertEquals(expected, FieldVerifier.isValidQuery(query));
}
```

Na pierwszy rzut oka można uznać, że ten test skupia się na jednej sprawie (weryfikacji zapytania), ale patrząc bardziej wnikliwie można zaobserwować kilka niepokojących sygnałów:

- nazwa metody testowej - `testQueryVerification` - jest raczej ogólna i doprawdy trudno byłoby przerobić ją tak, by rozpoczynała się od *"should"* (patrz sekcja 5.4.4),
- nazwa metody dostarczającej dany[12] - `data` - nie rodzi żadnych skojarzeń z testowanym scenariuszem,

[12]W tym przypadku jest to metoda oznaczoa anotacją `@DataProvider` z TestNG. Jeżeli używasz JUnit to pewnie w tym miejscu zastosowałbyś anotację `@Parameters` z frameworka JUnitParams.

- test wyposażony jest w (przyznaję, szczątkową) logikę: asercja zależy od wartości przekazanej flagi `expected`,
- do weryfikacji używana jest ogólna metoda - `assertEquals`.

By sprawdzić czy test jest zgodny z zasadą SRP można zadać następujące pytanie; *"Gdy ten test nie przejdzie, to czy patrząc na nazwę jego metody będę w stanie określić jaka funkcjonalność mojej aplikacji nie działa?"*.

Zobaczmy co można zrobić by nieco poprawić ten test. Oto inny jego wariant, w którym weryfikacja została podzielona na dwie części:

```
@DataProvider
public Object[][] validQueries() {
    return new Object[][] { {"48"}, {"48123"},
        {"+48"}, {"++48"}, {"+48503"}};
}

@Test(dataProvider = "validQueries")
public void shouldRecognizeValidQueries(String validQuery) {
    assertTrue(FieldVerifier.isValidQuery(validQuery));
}

@DataProvider
public Object[][] invalidQueries() {
    return new Object[][] {
        {"+4"}, {"++4"},
        {""}, {null}, {"  "} };
}

@Test(dataProvider = "invalidQueries")
public void shouldRejectInvalidQueries(String invalidQuery) {
    assertFalse(FieldVerifier.isValidQuery(invalidQuery));
}
```

Test mocno się nam wydłużył, ale chyba zyskał w ten sposób na czytelności. Jest też łatwiejszy do zrozumienia (usunęliśmy logikę związaną z flagę `expected`), i z pewnością jest zgodny z zasadą SRP. A to wszystko dzięki oddzieleniu pozytywnych i negatywnych przypadków testowych.

Bardzo podobają mi się nowe nazwy metod i zmiennych. Każda z nich dobrze oddaje sens danej metody czy zmiennej. Mamy więc metode dostarczającą poprawne dane `validQueries()` i mamy metodę testową, która przyjmuje poprawne zapytanie `validQuery`. Oraz ich przeciwieństwo: metodę `invalidQueries()` i metodę, która przyjmuję niepoprawne zapytanie `invalidQuery`. Również nazwy obu metod testowych dokładnie opisują jaki scenariusz weryfikuje każda z nich.

Zwróć uwagę na nazwy metod. Czy przybliżają one czytelnikowi testowany scenariusz?

Logika w testach to zło! Nawet najprostsza!

4.6.2. Testuj odpowiedzialności a nie metody!

Spójrzmy teraz na kolejny przykład testu, który robi zbyt wiele. Ale najpierw przyjrzyjmy się metodzie, którą testuje. Należy ona do klasy `UserRegisterController` i wygląda następująco:

```
public ModelAndView registerUser(UserData userData, BindingResult result,
                    HttpServletRequest request) {
    if (result.hasErrors()) {
        return showRegisterForm(request, false);
    }

    User savedUser = userService.saveNewUser(userData);
    mailSender.sendRegistrationInfo(savedUser);
    return new ModelAndView("redirect:/signin");
}
```

Hm… czy na pewno zadaniem kontrolera webowego jest zarządzanie logiką biznesową?

Załóżmy, że w przedstawionych dalej testach wszystkie zmienne zostały prawidłowo zainicjalizowane:

```
// mocks
UserData userData = mock(UserData.class);
UserService userService = mock(UserService.class);
BindingResult bindingResult = mock(BindingResult.class);
MailSender mailSender = mock(MailSender.class);
User user = mock(User.class);
HttpServletRequest request = mock(HttpServletRequest.class);

// sut
UserRegisterController userRegisterController =  ... // object created;
```

Spójrzmy teraz na oryginalny test.

Musiałem mocno uprościć oryginalny tekst, tak by móc skupić się na sednie zagadnienia, które rozważamy. Niestety, taki zabieg sprawia jednocześnie, że zmiany które wprowadzimy wydadzą się mniej istotne, niż gdybym przedstawił je posiłkując się o wiele dłuższym fragmentem kodu.

```
@Test
public void shouldReturnRedirectViewAndSendEmail() {
    //given
    given(bindingResult.hasErrors()).willReturn(false);
    given(userService.saveNewUser(eq(userData)))
        .willReturn(user);

    //when
    ModelAndView userRegisterResult = userRegisterController
            .registerUser(userData, bindingResult, request);

    //then
    assertThat(userRegisterResult.getViewName())
        .isEqualTo("redirect:/signin");
    verify(mailSender).sendRegistrationInfo(user);
}
```

Jak widać ten test sprawdza dokładnie co robi testowana metoda. Weryfikacji podlegają dwa oczekiwania:

- że metoda `sendRegistrationInfo()` obiektu `mailSender` zostanie wywołana z określonym parametrem (obiektem `user`),
- że użytkownik (ten który wypełnił formularz) zostanie przekierowany na określoną stronę.

Dodatkowo test sprawdza również czy obiekt `user` zostaje zapisany w bazie za pośrednictwem obiektu `userService`.

Ten test nie jest zły. Prawdę powiedziawszy jest całkiem w porządku. Ja jednak sugerowałbym rozdzielić go na dwie części, tak by każda zajęła się jednym wymaganiem:

```
@Test
public void shouldRedirectToSigninPageWhenNoErrors() {
    //given
    given(bindingResult.hasErrors()).willReturn(false);

    //when
    ModelAndView userRegisterResult = userRegisterController
            .registerUser(userData, bindingResult, request);

    //then
    assertThat(userRegisterResult.getViewName())
```

```
        .isEqualTo("redirect:/signin");
}
```

Pierwszy test sprawdza wyłącznie czy użytkownik zostaje prawidłowo przekierowany.

```
@Test
public void shouldNotifyAboutNewUserRegistration() {
    //given
    given(bindingResult.hasErrors()).willReturn(false);
    given(userService.saveNewUser(eq(userData)))
        .willReturn(user);

    //when
    userRegisterController.registerUser(userData, bindingResult, request);

    //then
    verify(mailSender).sendRegistrationInfo(user);
```

Drugi test weryfikuje czy wywołano metodę `sendRegistrationInfo()` obiektu `mailSender`.

Główna różnica pomiędzy pierwotną wersją a tą zaproponowaną przeze mnie polega na tym, że pierwotna wersja **testowała metodę** `registerUser()` podczas gdy aktualna wersja skupia się na **testowaniu odpowiedzialności** klasy `UserRegisterController`. W aktualnej wersji każda z metod testowych skupia się na weryfikacji jednej odpowiedzialności klasy. Skoro jedną z odpowiedzialności klasy jest przekierowanie użytkownika po prawidłowym wypełnieniu formularza, to mamy test, który to sprawdza. Innym wymaganiem jest by klasa wysłała powiadomienie po udanej rejestracji. I na to również mamy osobny test.

Jestem pewny, że testowana klasa ma szerszą odpowiedzialność (przykładowo, zakładam że zwraca informacje o błedach gdy formularz rejestracji nie został kompletnie wypełniony). I dobrze. Zawsze możemy dodać kolejne testy by sprawdzić te dodatkowe odpowiedzialności. Odpowiedzialności, nie metody!

Przez analogię do kodu produkcyjnego, również i testy powinny być możliwie małe i skupione na jednym zagadnieniu. Powinniśmy również abstrahować od szczegółów implementacyjnych by nie osłabiać żywotności testów. Zmiany które wprowadziliśmy, pomagają nam zrealizować oba te cele. Aktualne testy są skupione na pojedynczych odpowiedzialnościach, przez co zmiany w metodzie `registerSubmit()` mogą wpłynąć tylko na jeden z nich. Wówczas łatwiej będzie stwierdzić, co tak naprawdę nie działa. I tak właśnie powinno być!

Zapomnij o metodach. Testuj odpowiedzialności klasy.

Prosta reguła: *"Jedna metoda robi kilka rzeczy? Potrzebujesz kilku testów!"*.

4.6.3. A teraz kontrprzykład

W poprzedniej sekcji namawiałem do rozbijania metod testowych na mniejsze, tak by każda skupiała się na pojedynczym przypadku testowym. A teraz zademonstruję przykład, w którym będę namawiał do nie rozdzielania metod na mniejsze. Brzmi to schizofrenicznie, ale mam nadzieje, że dalsza lektura wyjaśni sprawę.

Spójrzmy na poniższy test:

```
@Test
public void shouldRecognizeDistrict() {
    //given
    District district = mock(District.class);
    District anotherDistrict = mock(District.class);

    //when
    City city = new City(district, NUMBER_OF_PEOPLE);

    //then
    assertThat(city.isLocatedIn(district)).isTrue();
    assertThat(city.isLocatedIn(anotherDistrict)).isFalse();
}
```

Jak widać weryfikuje on dwa scenariusze - pozytywny i negatywny. Ten test sprawdza czy:

- obiekt klasy `City` potrafi rozpoznać województwo, do którego należy,
- oraz czy potrafi rozpoznać województwo, do którego nie należy.

Czy w związku z tym powinniśmy rozbić ten test na dwa mniejsze? Moim zdaniem nie.

W przeciwieństwie do poprzednio omawianego przykładu tutaj liczba testowanych przypadków jest bardzo mała. To sprawia, że test jest krótki i łatwy do zrozumienia. Inaczej mówiąc, rozmiar metody testowej (biorąc pod uwagę ilość przypadków testowych, a nie liczbę linii kodu) nie przekracza mojego prywatnego limitu bezpieczeństwa. I dlatego też nie czuję powodu by podzielić ten test na dwa.

Dla pełnego obrazu zobaczmy jak wyglądałby owe dwa testy:

```
@Test
public void shouldRecognizeItsDistrict() {
    //given
    District district = mock(District.class);

    //when
    City city = new City(district, NUMBER_OF_PEOPLE);

    //then
    assertThat(city.isLocatedIn(district)).isTrue();
}

@Test
public void shouldRecognizeDifferentDistrict() {
    //given
    District district = mock(District.class);
    District anotherDistrict = mock(District.class);

    //when
    City city = new City(district, NUMBER_OF_PEOPLE);

    //then
    assertThat(city.isLocatedIn(anotherDistrict)).isFalse();
}
```

Nie mam tu czego krytykować. Jedyne co mam do powiedzenia, to że sam bym takiego testu nie napisał, bo wydaje mi się przesadnie rozdrobniony. Proszę, podejmij decyzję, którą z zaprezentowanych wersji uważasz za lepszą.

Reguły są po to, żeby je łamać. ;) Więc łammy je (świadomie) przynajmniej od czasu do czasu.

A czemu mock?

Dociekliwy czytelnik mógłby tu zapytać dlaczego zdecydowałem się zastąpić mockiem obiekt klasy `District` zamiast użyć prawdziwego obiektu. Jest to zasadne pytanie, w końcu klasa `District` wygląda na bardzo prostą (prawdopodobnie jest to POJO). A skoro tak, to chyba nie zasługuje na to, by ją mockować?

Już wyjaśniam. Po pierwsze mockowanie mockowaniu nierówne. Tak naprawdę, to powołany do życia konstrukcją `mock(District.class)` obiekt nie jest mockiem z prawdziwego zdarzenia, ale tzw. *dummy object*. Zadaniem mocków jest umożliwić sprawdzenie jakie komunikaty otrzymały. Tymczasem nam obiekt klasy `District` nie służy do tego celu: my tylko potrzebujemy żeby istniał.
Niestety, używamy terminu mock do określenia wszelkiego rodzaju obiektów zastępujących prawdziwe obiekty. Frameworki tworzące mocki - takie jak Mockito - nie pomagają w rozróżnieniu pomiędzy nimi oferując wspólną metodę ich tworzenia - w przypadku Mockito jest to właśnie statyczna metoda `mock()`.

Główną zaletą zastosowania mocka nad użyciem `new District()` jest to, że mocka nie obchodzi to jak konstruuje się obiekty tej klasy. Ta cecha mocka może okazać się całkiem wartościowa, zwłaszcza w obliczu (prawdopoobnej) ewolucji klasy `District`. Być może już w tej chwili jej konstruktor przyjmuje kilka argumentów. Być może w przyszłości te argumenty "urosną" - zwiększy się ich liczba, lub być może proste typy zostaną zastąpione przez typy złożone, a więc także wymagające misternego konstruowania. Zauważmy, że wówczas nasz kod testowy zostałby mocno zaśmiecony szczegółami konstrukcji klasy, która tak naprawdę jest tylko drobnym elementem testu. Żywotność takiego testu zostałaby zdecydowanie obniżona: test byłby wrażliwy na wszelkie zmiany w konstrukcji obiektu klasy `District`.

Rozdział 5. Czytelność

> Kod dużo częściej się czyta, niż się go pisze.
>
> — Mądrość programistyczna

Dość często zdarza się nam czytać kod testowy: traktujemy go jako dokumentację opisująca jak działają poszczególne klasy. Dzięki temu możemy zrozumieć niuanse ich działania i łatwiej nam dokonywać zmian. Czytamy kod testowy również wtedy gdy któryś test padnie. Musimy wówczas zrozumieć co się stało i podjąć się naprawy.

W związku z powyższym wydaje się, że całkiem sensownie byłoby pisać testy tak, by ich czytanie było przyjemnością. W tej sekcji skupimy się właśnie na aspekcie czytelności testów. Jak zobaczymy czasami nawet bardzo niewielkie zmiany mogą tu przynieść spore korzyści.

5.1. Formatowanie też jest ważne

Autorem tej sekcji jest Tomasz Borek (https://lafkblogs.wordpress.com/). Na polski przetłumaczył Tomek Kaczanowski.

Poniżej przykład testu sprawdzającego zmienianie pytania przypominającego hasło (tzw. pytanie bezpieczeństwa). Test jest doskonałą ilustracją jak dalece formatowanie wpływa na czytelność kodu… A jak wiemy, kod **znacznie częściej** czytamy, niż piszemy.

```
@Test
public void will_getChangSecurityQuestRgtAndDetails_if_AdvUserhasRuleId25(){
    User user = createUser(userId);
    user.setAdvanced(true);
    PasswordRuleDto passwordRuleDto = new PasswordRuleDto();
    passwordRuleDto.setPasswordRuleId(rulId25);
    List<PasswordRuleDto> passwordRules = new ArrayList<PasswordRuleDto>();
    passwordRules.add(passwordRuleDto);
    given(currentUser.getUser()).thenReturn(user);
    given(userDAO.readByPrimaryKey(userId)).thenReturn(user);
    given(passwordBean.getPasswordRules()).thenReturn(passwordRules);
    UserSecurityQuestionDto dto = userChangeSecurityQuestionBean
        .getChangSecurityQuestionRgtAndDetails();
    assertNotNull(dto.getEmail());
    assertNotNull(dto.getFirstName());
    assertNotNull(dto.getLastName());
    assertEquals(dto.isChangeSecurityQuestion(), true);
}
```

A teraz spójrzmy co osiągniemy, dodając trzy komentarze i trzy puste linie:

```
@Test
public void will_getChangSecurityQuestRgtAndDetails_if_AdvUserhasRuleId25(){
    // given
    User user = createUser(userId);
    user.setAdvanced(true);
    PasswordRuleDto passwordRuleDto = new PasswordRuleDto();

    passwordRuleDto.setPasswordRuleId(rulId25);
    List<PasswordRuleDto> passwordRules = new ArrayList<PasswordRuleDto>();
    passwordRules.add(passwordRuleDto);

    given(currentUser.getUser()).willReturn(user);
    given(userDAO.readByPrimaryKey(userId)).willReturn(user);
    given(passwordBean.getPasswordRules()).willReturn(passwordRules);

    // when
    UserSecurityQuestionDto dto = userChangeSecurityQuestionBean
        .getChangSecurityQuestionRgtAndDetails();

    // then
    assertNotNull(dto.getEmail());
    assertNotNull(dto.getFirstName());
    assertNotNull(dto.getLastName());
    assertEquals(dto.isChangeSecurityQuestion(), true);
}
```

Jasne, wciąż dalekie od ideału. Ale o wiele czytelniejszy. Podział na bloki ułatwia dalszą refaktoryzację. Wyraźnie rozróżnione są przygotowania do testu, wołanie testowej metody i asercje. Łatwiej zorientować się, co się dzieje i (mam nadzieję) co dalej należy zrobić

5.2. Ceramoniał

BDD to wspaniała idea pozwalająca wyrażać w kodzie wymagania biznesowe. Wyznaczany przez *"given/when/then"* rytm sprawia, że testy napisane w tym stylu wyróżniają się jakością. Czasami jednak trudno mi oprzeć się wrażeniu, że pewne konstrukcje są przerostem formy nad treścią. Tyczy się to zwłaszcza prostych testów, takich jak ten poniżej.

```
@Test
public void shouldBuildEmailSender() {
    // given
    String senderName = "Chuck Norris";
    String senderEmail = "chuck@norris.com";

    // when
    String emailSender = EmailUtils.buildEmailSender(senderName, senderEmail);

    // then
```

```
    assertThat(emailSender).isEqualTo("Chuck Norris <chuck@norris.com>");
}
```

Patrząc na zawartość sekcji *"given"*, *"when"* and *"then"* trudno mi oprzeć się wrażeniu, że zostały one dodane tylko by uhonorować tradycję pisania testów w stylu BDD. To trochę tak jak wprowadzanie do kodu kolejnych wzorców projektowych, nie dlatego że są naprawde potrzebne, ale dlatego że ktoś powiedział, że tak byłoby fajniej. Z całym szacunkiem do BDD, ale czemu nie wystarczyłoby napisać tego testu ot tak po prostu? Na przykład tak:

```
@Test
public void shouldBuildEmailSender() {
    String emailSender = EmailUtils
        .buildEmailSender("Chuck Norris", "chuck@norris.com");
    assertThat(emailSender).isEqualTo("Chuck Norris <chuck@norris.com>");
}
```

Albo nawet tak:

```
@Test
public void shouldBuildEmailSender() {
    assertThat(EmailUtils.buildEmailSender("Chuck Norris","chuck@norris.com"))
        .isEqualTo("Chuck Norris <chuck@norris.com>");
}
```

Podobnie jak powyżej, kolejny test również wydaje mi się przerostem formy nad treścią:

```
@Test
public void shouldBeAdministrator() {
    //given
    User user = new Administrator();

    //when
    boolean administrator = user.isAdministrator();
    boolean guest = user.isGuest();
    boolean moderator = user.isModerator();

    //then
    assertThat(administrator).isTrue();
    assertThat(guest).isFalse();
    assertThat(moderator).isFalse();
}
```

Z radością powitałbym skróconą wersję, która ma tą samą siłę, ale jest dużo bardziej zwięzła i czytelna:

```
@Test
public void shouldBeAdministrator() {
    User user = new Administrator();
```

```
    assertThat(user.isAdministrator()).isTrue();
    assertThat(user.isGuest()).isFalse();
    assertThat(user.isModerator()).isFalse();
}
```

KISS[1]. Keep It Simple Stupid! :)

5.3. Tworzenie obiektów

> Wszystko trzeba robić tak prosto, jak to tylko jest możliwe, ale nie prościej.
>
> — Albert Einstein

Większość testów zdaje się powielać ten sam schemat struktury, którą często określamy mianam **stwórz**/**wykonaj**/**sprawdź**. Najpierw **tworzymy** w teście obiekty - obiekt klasy której działanie będziemy weryfikować (SUT) i obiekty współpracowników. Potem **działamy**, czyli wykonujemy jakąś metodę obiektu SUT. Na końcu **weryfikujemy** czy uzyskany rezultat jest zgodny z naszymi oczekiwaniami.

Część, w której wykonujemy metodę SUT, jest zazwyczaj bardzo krótka. Często jest to jedna linijka. Niewiele można tu poprawić.

Inaczej ma się sprawa z częściami testu, w których tworzymy obiekty i weryfikujemy wyniki. Zdarza im sie rozrosnąć do sporych rozmiarów, w związku z czym jest tam co poprawiać.

W tej sekcji zajmiemy się problemem tworzenia obiektów niezbędnych do wykonania testu. Asercjami zajmiemy się w sekcji 5.6.

Spójrzmy na pierwszy przykład. Do testów integracyjnych i end-to-end, których fragmenty poznamy w tej sekcji, potrzebujemy obiektu klasu `User`. Jest to jedna z głównych encji systemu. Posiada dobrze ponad 20 pól. W testach integracyjnych wielokrotnie zachodziła potrzeba tworzenia obiektów tej klasy. Później były one zapisywane do bazy danych i wykorzystywane w wielu innych testach.

Oto fragmenty kodu tworzącego obiekty klasy `User` w różnych klasach testowych:

```
@Before
public void initialize() {
    User user = new User("email@example.com", "Example", "Example", "qwerty",
```

[1] See https://en.wikipedia.org/wiki/KISS_principle

```
        "Europe/Warsaw", UserState.NOT_VERIFIED, new Address());
    ...
}

@Test
public void shouldCommitTransaction() {
    User user = new User("firstName", "lastName", "password",
        "email@example.com", "qwerty", UserState.ACTIVE, new Address());
    user.setRegistrationDate(oneDayAgo.toDate());
    user.setAccessCode("qwerty");
    ...
}

@Test
public void shouldGetUserByCompanyData() {
    User user = new User("email", "FirstName", "LastName", "Password",
        "Europe/Warsaw", UserState.ACTIVE, address);
    user.setRegistrationDate(new Date());
    user.setCompany(company);
    user.setAccessCode("Access Code");
    ...
}
```

Hm, dostrzegam tu kilka problemów. Po pierwsze, w wielu testach zaszyta jest wiedza na temat tworzenie obiektów klasy `User`. To oznacza, że będą one wymagały zmiany, ilekroć dodamy nowy argument do konstruktora tej klasy.
Po drugie, bardzo ciężko jest zorientować się, które z parametrów przesyłanych do konstruktora klasy `User` ma faktyczne znaczenie dla testowanego scenariusza. W niektórych przypadkach intuicja podpowiada, które z nich są zupełnie nieistotne - ale tak naprawdę nie wiadomo tego.
Po trzecie, jeżeli uważnie przeanalizować te fragmenty kodu, spostrzeżemy, że używają one różnych konstruktorów. Okazuje się, że te konstruktory zostały stworzone właśnie po to, by łatwo było konstruować obiekty w testach! Innymi słowy, dodaliśmy do kodu produkcyjnego klasy `User` konstruktory, które umożliwiają stworzenie jej obiektów z wypełnionymi wybranymi polami (i z pozostałymi polami ustawionymi na `null`). To brzmi dość groźnie. W ten sposób umożliwiamy tworzenie ułomnych obiektów klasy `User`!
Po czwarte, obok konstruktorów używane są też settery. To wydłuża jeszcze kod odpowiedzialny za tworzenie obiektów klasy `User`.

Co możemy zrobić? W pierwszym odruchu możemy próbować stworzyć kilka metod prywatnych umożliwiających tworzenie obiektów. Tak, to z pewnością byłoby lepsze od konstruktorowego szaleństwa, które właśnie widzieliśmy. Doświadczenie podpowiada mi jednak, że tego typu metody mają tendencję do rozmnażania się i rozpełzania po wielu klasach. Zazwyczaj nie są też wystaraczająco elastyczne by wyrazić intencję przyświęcającą w poszczególnych testach. Ten problem można rozwiązać poprzez parametryzację tychże prywatnych metod. Koniec końców okazuje się jednak, że i to jest niewystarczające - wówczas

w kodzie obok wywołań tych prywatnych pojawiają się też i settery. I praktycznie trafiamy do punku wyjścia.

Stąd też sugeruję inne rozwiązanie. Zainwestujmy nieco czasu w stworzenie wyspecjalizowanych klas tworzących obiekty na potrzeby testów. W kolejnej sekcji opowiem o ich implementacji, a póki co przyjrzyjmy się jak wyglądałyby testy z ich zastosowaniem:

```
@Before
public void initialize() {
    User notVerifiedUser = UserBuilder.createUser(UserState.NOT_VERIFIED)
                    .create();
    ...
}

@Test
public void shouldCommitTransaction() {
    User user = UserBuilder.createActiveUser()
                    .create();
    ...
}

@Test
public void shouldGetUserByCompanyData() {
    User user = UserBuilder.createActiveUser()
                    .withCompany(company)
                    .create();
    ...
}
```

Spójrzmy co udało się w ten sposób uzyskać. Oto zalety użycia *test data builders*:

- obiekty domeny moga być niezmienne (immutable); nie potrzeba żadnych setterów, wystarczy konstruktor,
- obiekty domeny nie muszą się zajmować logika konstruowania obiektów (w szczególności samych siebie),
- tylko jedna klasa (budowniczy) wie jak stworzyć obiekt określonej klasy, więc zmiany będą dotyczyć tylko tej jednej klasy,
- jako twórca DSLa tworzącego obiekt, możesz dodać do niego metody, które ułatwią tworzenie obiektu (i będą bardzo czytelne).

Jak Ci się wydaje - dobry to pomysł czy też nie? Niestety, ale nie sposób zaprezentować zalet stosowania budowniczych na podstawie tak krótkich fragmentów kodu. Zachęcam do wypróbowania ich w kodzie - najlepiej dla ważnych obiektów z domeny. Wówczas łatwiej będzie Ci docenić ich zalety.

IDE pomagają tworzyć budowniczych. Spróbuj!

Budowniczy obiektów testowych[2].

Budowniczy obiektów testowych to wyspecjalizowana klasa odpowiedzialna za tworzenie obiektów określonego typu na potrzeby testów. Sens ich istnienia jest następujący:

- sprawiają, że testy nie wiedzą nic o konstruowaniu obiektów, których używają,
- niweluja potrzebę zaśmiecania kodu produkcyjnego dodatkowymi konstruktorami i metodami tworzonymi wyłącznie na potrzeby testów,
- standaryzują sposób tworzenia obiektów w kodzie testowym,
- dostarczają wygodnych metod, dzięki którym tworzenie obiektów jest bardzo proste (i przyjemne do czytania).

Klasy budowniczych obiektów testowych są bardzo proste. Zazwyczaj dostarczają wielu setteropodobnych metod, zwracających `this` (instancje budowniczego). To pozwala na łączenie wywołań metod, co zobaczymy już za chwilę.
Każdy budowniczy posiada właściwą metodę tworzącą (zwyczajowo nosi ona nazwę `build()` albo `create()`), która wołana na końcu tworzy ostatecznie obiekt. Przykładowo, budowniczy klasy `MockServer` mógłby wyglądać w taki sposób:

```
public class MockServerBuilder {
    private Map<String, String> responseMap;
    private ResponseType responseType;
    private String serverUrl;
    private boolean ssl;

    public MockServerBuilder withResponse(Map<String, String> responseMap) {
        this.responseMap = responseMap;
        return this;
    }

    public MockServerBuilder withResponseType(ResponseType responseType) {
        this.responseType = responseType;
        return this;
    }
```

[2]Tzw. *test data builders* - patrz http://nat.truemesh.com/archives/000714.html

```
    public MockServerBuilder withUrl(String serverUrl) {
        this.serverUrl = serverUrl;
        return this;
    }

    public MockServerBuilder withoutSsl() {
        this.ssl = false;
        return this;
    }

    ... probably some more methods here

    public MockServer create() {
        return new MockServer(responseMap, responseType, serverUrl, ssl);
    }
}
```

Zauważmy że:

- napisanie budowniczego to nie problem - zwłaszcza że niektóre IDE pomagają w tym,
- to my decydujemy jaką formę przybiorę metody ustawiające poszczególne pola klasy i możemy je kształtować wedle naszego uznania.

5.3.1. Ah, jakże łatwo o błąd!

Kończąc powoli temat tworzenie obiektów przyjrzyjmy się jeszcze jednemu przykładowi. Tym razem zobaczymy, jak dziwacznie może się zakończyć próba użycia wieloargumentowego konstruktora w kodzie testowym. Oto początkowe linie dwóch metod testowych należących do jednej klasy:

```
@Test
public void shouldCreateUserAndFindHimById() {
    //given
    User user = new User("elvis2", "elvis@is.alive");
    ...
}

@Test
public void shouldUpdateUser() {
    // given
    User user = new User("elvis@com.pl", "elvis3");
    ...
}
```

Hm… to wydaje się być podejrzane. Czym właściwie jest pierwszy parametr - adresem email czy loginem (imieniem?) użytkownika? Mogę wyjawić tę tajemnicę, bo tak się składa że widziałem

kod produkcyjny. Otóż pierwszy parametr to email. Za to drugi… w życiu bym na to nie wpadł, ale drugi to (uwaga! uwaga!) **hasło**! Tak, hasło. Co, nie było to oczywiste? Dla mnie również. Niestety, tak to się kończy gdy używamy w testach byle jakich wartości jako parametrów przekazywanych do konstruktorów.

Przyznaję że zaskoczył mnie fakt znalezienia takiego dziwoląga w użyciu dwuargumentowego konstruktora! Sądziłem, że takie błędy mogą się zdarzyć gdy parametrów jest pięć albo więcej…

Poprawa tego fragmentu kodu nie nastarcza większych trudności. Prawdopobnie ta wersja jest nieco lepsza:

```
User user = new User(USER_EMAIL, USER_PASSWORD);
```

Można też użyć budowniczych (poniższy kod ładnie podkreśla fakt, że na potrzeby testu można użyć **jakiegokolwiek** użytkownika):

```
User user = new UserBuilder().standardUser()
    .create();
```

Można też na przykład tak (jeżeli email jest akurat istotny w testowanym scenariuszu):

```
User user = new UserBuilder().standardUser()
    .but().withEmail("elvis@presley.com")
    .create();
```

I kolejna wersja zakładająca, że oba parametry mają znaczenie:

```
User user = new UserBuilder().standardUser()
    .but().withEmail("elvis@presley.com")
    .and().withPassword("sivle")
    .create();
```

Jak widać mamy tu spory wybór. W zależności od kontekstu i naszych upodobań możemy zdecydować się na używanie konstruktorów, lub zainwestować w implementację budowniczych.

Nie poprzestawaj na budowniczych tworzonych przez IDE! Dopasuj ich do swoich potrzeb, tak by służyły do pisania czytelnego kodu testowego.

5.3.2. Używaj prostszych typów

> Obywatelu, zrób sobie dobrze sam […]

— Kabaret OTTO

Weźmy teraz na warsztat kolejny przykład tworzenia obiektów na potrzeby testów. Oto fragment kodu testowego:

```
List<ProductCategory> categories = Arrays.asList(
    product("car", new BigDecimal("100"), new BigDecimal("10000")),
    product("bike", new BigDecimal("20"), new BigDecimal("5000")),
    product("plane", new BigDecimal("1000"), new BigDecimal("500000"))
);
```

Ten kod tworzy kilka obiektów klasy `ProductCategory`. Wykorzystuje do tego celu metodę prywatną `product()`. Wydaje mi się niepokojące, że ten fragment kodu testowego używa klasy `BigDecimal`. Rozumiem, że w ten sposób odzwierciedla on złożoność domeny (domyślam się, że wartości te są cenami). Jednak gdy patrzę na to jakich wartości faktycznie używa ten test, okazuje się, że zwykłe liczby całkowite byłyby w zupełności wystarczające.
Oznacza to, że możemy ich użyć, jeżeli tylko zmodyfikujemy metodę `product()`. A to akurat z pewnością możemy zrobić, w końcu jesteśmy jej autorem. A oto ten sam kod po zmianie:

```
List<ProductCategory> categories = Arrays.asList(
    product("car", 100, 10000),
    product("bike", 20, 5000),
    product("plane", 1000, 500000)
);
```

Oczywiście można powiedzieć, że to nic takiego, ale jeżeli mamy szereg testów wykorzystujących skomplikowane obiekty (jak np. liczby reprezentowane przez `BigDecimal`), to różnica może być znacząca.

Powyższa wskazówka nie dotyczy rzecz jasna wyłącznie metod prywatnych. To samo można doradzić w kwestii metod tworzących obiekty, w które wyposażamy klasy je budujące. Powinniśmy zadbać o to, by możliwie ułatwić ich użycie i uczynić je bardzo czytelnym. Przykładowo, w przypadku obiektów, które mają kilka pól z datami, opłaca się udostępnić metody przyjmujące tekstowe ich reprezentacje (np. *"2015-08-15"*) zamiast konstruować obiekty typu `Date`.

5.4. Nazywaj rzeczy po imieniu

Odpowiednie dać rzeczy - słowo!

— Cyprian Kamil Norwid *Vade-mecum - Za wstęp. Ogólniki*

Sprawa jest dość intrygująca. W kodzie produkcyjnym nie popełniamy rażących błędów nadając nazwy zmiennym i metodom. Gdy jednak piszemy testy porzucamy te dobre nawyki i robimy

rzeczy, których nigdy nie odważylibyśmy się zrobić w kodzie produkcyjnym. Podejrzewam, że dzieje się tak głównie dlatego, że w testach często mamy do czynienia z kilkoma obiektami tego samego typu. I radzimy sobie z nazywaniem ich przyjmując najprostszy możliwy schemat, typu `productA`, `productB`, `productC` itd. Zazwyczaj takie podejście fatalnie odbija się na czytelności testu.

5.4.1. Domyśl się, to przecież oczywiste!

Pierwszy przykład pochodzi z przypadku testowego, który weryfikował funkcjonalność związaną ze sprawdzeniem uprawnień. Poniższy listing przedstawia niewielki fragment tego testu - metodę dostarczającą dane.

```
@DataProvider
public static Object[][] userPermissions() {
    return new Object[][]{
        {"user_1", READ},
        {"user_2", READ},
        {"user_2", WRITE},
        {"user_3", READ},
        {"user_3", WRITE},
        {"user_3", DELETE}
    };
}
```

Jak widać przygotowane dla testu dane przypisują trzem użytkownikom (`user_1`, `user_2` i `user_3`) pewne uprawnienia (`READ`, `WRITE`, `DELETE`). Wszystko jasne, ale dlaczego niby `user_1` posiada tylko uprawnienie `READ` podczas gdy `user_3` posiada też uprawnienie `WRITE` i `DELETE`? Nie jest to oczywiste, i gdy ten test padnie, wówczas dowiemy się przykładow, że *"oczekiwano, że* `user_3` *ma uprawnienie typu* `DELETE`*"*. A wtedy rozpocznie się proces myślowy, w którym będzie trzeba rozstrzygnąć czy aby na pewno `user_3` powinien mieć takie uprawnienie? Oczywiście nie będzie tego jak sprawdzić. W końcu od napisania kodu minęły już długie miesiące. I nikt nie wie już kim u licha jest ów `user_3`!

Na szczęście lekarstwo jest proste.

```
@DataProvider
public static Object[][] userPermissions() {
    return new Object[][]{
        {"guest", READ},
        {"logged", READ},
        {"logged", WRITE},
        {"admin", READ},
        {"admin", WRITE},
        {"admin", DELETE}
    };
}
```

I wszystko jasne. Gość serwisu może tylko czytać, zalogowany użytkownik może również pisać, a administrator może wszystko. Teraz komunikat ewentualnego błędu - *"oczekiwano, że* `admin` *ma uprawnienie typu* `DELETE`*"* - nie wymaga od nas zbyt wiele intelektualnego wysiłku.

5.4.2. Object1, Object2, Object3

Spójrzmy teraz na kolejny przykład. Oto (bardzo niewielki) fragment skomplikowanego testu integracyjnego, który tworzył szereg obiektów (głównie klasy `User`). W użyciu było kilka metod pomocniczych, których zadaniem było nie tylko stworzenie potrzebnych obiektów, ale i zapisanie ich w bazie danych.

```
@Test
public void importantTest() {
    User user1 = daoTestHelper.addUserAndAssociateWithProduct(
            product, 30l, ProductUserState.ACTIVE);
    daoTestHelper.userProduct30DayStatistics(user1, "PL", 1);
    daoTestHelper.userProduct30DayStatistics(user1, "US", 2);
    User user2 = daoTestHelper.addUserAndAssociateWithProduct(
            product, 30l, ProductUserState.ACTIVE);
    daoTestHelper.userProduct30DayStatistics(user2, "PL", 4);
    daoTestHelper.userProduct30DayStatistics(user2, "US", 8);

    // ... and so on till user5 or so
}
```

Powiem szczerze: nie cierpię takiego nazewnictwa obiektów. Pomysł dodawania sufiksów do nazw zmiennych (`user1`, `user2`, itd.) uważam za fatalny. Tak nazwa zmienna nie niesie kompletnie żadnych informacji na temat swojego stanu ani roli jaką odgrywa w teście. Na szczęście prosta zmiana nazwy zmiennych poprawia sytuację:

```
@Test
public void importantTest() {
    User userWith1PlEntry = daoTestHelper.addUserAndAssociateWithProduct(
            product, 30l, ProductUserState.ACTIVE);
    daoTestHelper.userProduct30DayStatistics(userWith1PlEntry, "PL", 1);
    daoTestHelper.userProduct30DayStatistics(userWith1PlEntry, "US", 2);
    User userWith4PlEntries = daoTestHelper.addUserAndAssociateWithProduct(
            product, 30l, ProductUserState.ACTIVE);
    daoTestHelper.userProduct30DayStatistics(userWith4PlEntries, "PL", 4);
    daoTestHelper.userProduct30DayStatistics(userWith4PlEntries, "US", 8);

    // ... and so on
}
```

Musisz mi wierzyć na słowo, że te nazwy mają jak najbardziej sens w kontekście swojej dziedziny problemu. Dobrze też współgrają z samodzielnie tworzonymi asercjami (patrz sekcja 5.6):

```
assertThat(result).hasResultsForUser(userWith4PlEntries, 4);
```

To zadziwiające, ale tak prosty refaktoring jak zwykła zmiana nazw, pozwolił mi kiedyś odkryć ciekawy błąd w kodzie testowym. Dopóki zmienne miały swoje oryginalne nazwy (`user1`, `user2`, itd.) nie byłem w stanie dostrzec tego problemu.
Spójrzmy na poniższy fragment kodu testowego. Powtarza się w nim ciągle ten sam wzorzec tworzenie obiektu użytkownika, a potem przypisywania mu pewnych statystyk. W jednym miejscu następuje odstępstwo od tego wzorca - czy potrafisz je dostrzec?

```
@Test
public void importantTest() {
    User user1 = daoTestHelper.addUserAndAssociateWithProduct(
            product, 30l, ProductUserState.ACTIVE);
    daoTestHelper.userProduct30DayStatistics(user1, "PL", 1);
    daoTestHelper.userProduct30DayStatistics(user1, "US", 2);
    User user2 = daoTestHelper.addUserAndAssociateWithProduct(
            product, 30l, ProductUserState.ACTIVE);
    daoTestHelper.userProduct30DayStatistics(user2, "PL", 4);
    daoTestHelper.userProduct30DayStatistics(user2, "US", 8);
    User user3 = daoTestHelper.addUserAndAssociateWithProduct(
            product, 30l, ProductUserState.DELETED);
    daoTestHelper.userProduct30DayStatistics(user3, "PL", 16);
    daoTestHelper.userProduct30DayStatistics(user3, "CZ", 32);
    User user4 = daoTestHelper.addUserAndAssociateWithProduct(
            product, 30l, ProductUserState.ACTIVE);
    daoTestHelper.userProduct30DayStatistics(user3, "US", 64);
    User user5 = daoTestHelper.addUserAndAssociateWithProduct(
            product, 30l, ProductUserState.DELETED);

    // ... and so on
}
```

Być może dałeś radę je wypatrzyć (gratuluję!). Mi się to nie udało. Dopiero gdy zmieniłem nazwy zmiennych, to zauważyłem, że w jednym miejscu zamiast zmiennej `user4` użyta jest zmienna `user3` (spójrz na przedostatnią linię kodu). Co ciekawe, ta pomyłka nie wpływa na wynik testu. Dzieje się tak, ponieważ akurat ta linia kodu przypisuje użytkownikowi statystyki, które nie powinny zostać uwzględnione w wyniku. Ale to nie znaczy, że ten błąd jest całkiem niewinny: z jego powodu ten test tak naprawdę nie sprawdza zakładanego scenariusza.

Łatwo o pomyłkę gdy nazwy zmiennych różnią się tylko nieznacznie.

Wnioski. To zupełnie tak samo jak z nazwami klas. Na początku używasz mało znaczących nazw typu `ClientDAOImpl` ale potem uczysz się, że można dobrać określenia lepiej opisujące specyfikę danej klasy. Wówczas zaczynasz używać nazw takich jak `HibernateClientDAO`,

`InMemoryClientDAO`, `HighPerformanceClientDAO` czy cokolwiek innego, co lepiej określa tą właśnie specyficzną implementację. Właściwie zawsze jest coś, co możesz powiedzieć o swojej klasie.
Wierzę, że podobnie jest ze zmiennymi użytymi w testach. Często zaczynamy od obiektów `client1` i `client2`, ale potem dochodzimy do wniosku, że tak naprawdę to chodzi nam o obiekty `client` i `vipClient` (podczas gdy w innym teście chodzi o obiekty `client` i `regularClient`). Spróbuj takiego znaczącego nazywania zmiennych - testy bardzo na tym zyskają!

Nie używaj sufiksów ani prefixów do odróżniania poszczególnych zmiennych w testach! Tak dobierz nazwy obiektów, by jasno określały ich specyfikę.

Obiekty w testach tworzysz ponieważ są potrzebne w jakimś konkretnym celu. Tworzysz je, bo są inne od pozostałych obiektów. Mają inne właściwości i/lub przeznaczenie. Skoro tak, to opowiedz o tym nadając im unikalne, dobrze rozpoznawalne nazwy.

5.4.3. Prawdziwa prawda i nieprawdziwa nieprawda

> Góralska teoria poznania mówi, że są trzy prawdy: święta prowda, tyż prowda i gówno prowda.
>
> — ks. Józef Tischner

Jeżeli uparliśmy się, że nasze testy mają mieć wartość dokumentacyjną, to musimy włożyć co nieco wysiłku w ich czytelność. Spójrzmy na poniższy przykład:

```
MockServer server = new MockServer(responseMap, true,
        new URL(SERVER_ROOT).getPort(), false);
```

Ta pojedyncza linijka kodu tworzy obiekt serwera klasy `MockServer`. Później obiekt ten jest wykorzystywany w testach.

Wszystko dobrze, ale do czego właściwie odnoszą się wartości `true` i `false`? Jakiż to rodzaj serwera tworzym?

Nie spodziewam się byśmy znaleźli obszerną dokumentację klasy `MockServer` (która to klasa zdaje się być klasą pomocniczą, stworzoną na potrzeby testów). Nie ma więc rady, trzeba zajrzeć w kod źródłowy by wyjaśnić znaczenie `true` i `false`. Prawdopodobnie nie będzie to bardzo trudne (choć może być - w zależności od stopnia skomplikowania tej klasy), ale nie zmienia to faktu, że jest to pewna uciążliwość. Test tworzący ów `MockServer` przestaje być wygodną dokumentacją. By go zrozumieć trzeba odwołać się do innego źródła - do kodu źródłowego. Nie jest to tragedia, ale też nic z czego należałoby się cieszyć.

Co więc można z tym zrobić? Istnieje tu kilka możliwych rozwiązań. Przyjrzyjmy się im.

Stałe

Po pierwsze można wprowadzić stałe i dzięki dobremu ich nazwaniu uzyskać dużo łatwiejszy do zrozumienia kod:

```
private static final boolean RESPONSE_IS_A_FILE = true;
private static final boolean NO_SSL = false;

MockServer server = new MockServer(responseMap, RESPONSE_IS_A_FILE,
        new URL(SERVER_ROOT).getPort(), NO_SSL);
```

Teraz nie ma wątpliwości jaki to serwer powołujemy do istnienia: taki, który nie używa SSL i odpowiada plikami. Taki zapis wydaje się być o wiele bardziej czytelny. No i nie napracowaliśmy się zbytnio by go uzyskać.

Metody prywatne

Inną możliwością jest przeniesienie konstrukcji obiektu do metody prywatnej o opisowej nazwie. Na przykład takiej:

```
private MockServer noSslFileServer() throws MalformedURLException {
    return new MockServer(responseMap, true,
        new URL(SERVER_ROOT).getPort(), false);
}
```

A potem już możemy tworzyć nasz serwer wywołując ją:

```
MockServer server = noSslFileServer();
```

Z pewnością jest to czytelniejsze niż oryginalna wersja kodu. Użycie metod prywatnych ma jednak swoje ograniczenia, o których nie powinniśmy zapominać[3]. Jeżeli w testach będziemy tworzyli różne typy serwerów, wówczas powinniśmy też stworzyć więcej takich metod prywatnych. Im więcej kombinacji opcji i parametrów tym będzie ich więcej.

Test Data Builders

Trzeci sposób wymaga od nas dużo większego nakładu pracy. Tak, dobrze się domyślasz: proponuję skorzystać ze znanego nam już wzorca *test data builder* (patrz sekcja 5.3), co pozwoli nam bardzo precyzyjnie opisać typ serwera jaki chcemy stworzyć.

```
MockServer server = new MockServerBuilder()
        .withResponse(responseMap)
        .withResponseType(FILE)
```

[3]Więcej na ten temat w sekcji 5.6.2.

```
        .withUrl(SERVER_ROOT)
        .withoutSsl()
        .create();
```

Czy to najlepsze rozwiązanie? Niekoniecznie, choćby dlatego że wymaga zdecydowanie najwięcej pracy. Jednak zachętą jest możliwość dowolnego ukształtowania API. Być może wygodniej byłoby dla nas, gdyby pewne kwestie były domyślne (np. brak obsługi SSL i zwracanie plików). Wówczas wywołanie wyglądałoby tak:

```
MockServer server = new MockServerBuilder()
        .createFileServer(SERVER_ROOT)
        .withResponse(responseMap)
        .create();
```

Oceniając przydatność *test data builderów*, ograniczę się do stwierdzenia, że im bardziej rozbudowane i częściej wykorzystywana jest dana klasa, tym większy sens ma tworzenie dla niej *test data buildera*.

5.4.4. should jest lepsze niż test

Działo się to dawno, dawno temu, w czasach gdy królestwem testów javowych władał Jego Wysokość Król JUnit Trzeci. W swojej niezmierzonej mądrości łaskawy ten władca nakazał by wszystkie metody testowe rozpoczynały się od prefiksu *test*. O dziwo zwyczaj ten przeżył śmierć swego stwórcy[4]. I wynikają stąd różne kłopoty. O czym zaraz się przekonamy.

Spójrzmy na następny przykład (po braku średników sądząc, ten kod to Groovy):

```
@Test
public void testOperation() {
    configureRequest("/validate")
    rc = new RequestContext(parser, request)
    assert rc.getConnector() == null
    assert rc.getOperation().equals("validate")
}
```

Jaki jest cel tego testu? Jego nazwa mówi, że powinien on '"testować operację"… Hm, no nie jest to szczególnie precyzyjne określenie. Jaką niby operację? Jakiego zachowania oczekujemy od systemu? A jeżeli ten test nie przejdzie, to czy wówczas będzie oczywiste jaka funkcjonalność systemu nie działa prawidłowo?

Spójrzmy prawdzie w oczy: wartość dokumentacyjna tego testu jest zerowa, jeżeli nie ujemna. Raczej wprowadza zamęt niż wyjaśnia cokolwiek.

[4]Niektórzy twierdzą, że zabił go jego młodszy brat - JUnit Czwarty. Inni obwiniają za śmierć władcy babarzyńskie plemię TestNG. Miłośnicy Javy w wersji 5 wskazują na brak annotacji jako na przyczynę jego upadku. Reszty mało to obchodzi, ale wszyscy są zadowoleni, że rządy JUnita Trzeciego dobiegły końca.

Jak pokazuje powyższy przykład, pierwszym problemem z metodami testowymi, których nazwy zaczynają się od słówka `test` jest taki, że często testują one *coś*. A kiedy nie przejdą, wówczas wiadomo tylko tyle, że *coś* w systemie nie działa tak jak powinno. A wtedy trzeba dociekać o co właściwie chodzi… A to może potrwać bardzo długo…

Spójrzmy teraz na kolejny przykład, który pozwoli nam przedyskutować inną wadę nazywania metod testowych z użyciem prefiksu `test`. Nazwa poniżej metody testowej obiecuje nam, że jej celem jest *"przetestowanie zapytania"*. Zobaczmy jak wywiązuje się z tej obietnicy.

```
@Test
public void testQuery(){
    when(q.getResultList()).thenReturn(null);
    assertNull(dao.findByQuery(Transaction.class, q, false));
    assertNull(dao.findByQuery(Operator.class, q, false));
    assertNull(dao.findByQuery(null, null, false));

    List result = new LinkedList();
    when(q.getResultList()).thenReturn(result);
    assertEquals(dao.findByQuery(Transaction.class, q, false), result);
    assertEquals(dao.findByQuery(Operator.class, q, false), result);
    assertEquals(dao.findByQuery(null, null, false), null);

    when(q.getSingleResult()).thenReturn(null);
    assertEquals(dao.findByQuery(Transaction.class, q, true).size(), 0);
    assertEquals(dao.findByQuery(Operator.class, q, true).size(), 0);
    assertEquals(dao.findByQuery(null, null, true), null);

    when(q.getSingleResult()).thenReturn(t);
    assertSame(dao.findByQuery(Transaction.class, q, true).get(0), t);
    when(q.getSingleResult()).thenReturn(o);
    assertSame(dao.findByQuery(Operator.class, q, true).get(0), o);
    when(q.getSingleResult()).thenReturn(null);
    assertSame(dao.findByQuery(null, null, true), null);
}
```

Ten test ma wiele na sumieniu:

- zdecydowanie łamie regułę SRP (patrz sekcja 4.6),
- używa magicznych przełączników (konia z rzędem temu, kto mi wyjaśni co oznaczają te wszystkie `true` i `false`; więcej na ten temat w sekcji 5.4.3.),
- używa zupełnie niezrozumiałych nazw zmiennych (`q`, `o` i `t`).

Niemniej jednak ten test coś jednak testuje. Uwierz mi proszę, że tak jest: przeanalizowałem go dokładnie i zapewniam, że ma to sens. A jednak nie mogę się powstrzymać od określenia

tego testu mianem absolutnie paskudnego. Jest zbyt długi i weryfikuje zbyt wiele scenariuszy testowych.

I w ten oto sposób dotarliśmy do drugiego problemu, który występuje często w metodach testowych nazywanych z użyciem prefiksu `test`. Test o nazwie takiej jak `testQuery()` po prostu zaprasza by wepchnąć do niego wszystko, co tylko dotyczy owego `query`. I w ten oto sposób rodzą się takie właśnie monstrualne metody testowe.

Jestem przekonany, że metoda taka jak ta przedstawiona powyżej nie powstałaby, gdyby programista nadał jej nazwę rozpoczynającą się od `should`. Pisząc `should`... szukamy szczególnego scenariusza, który powinien się wydarzyć w określonym przypadku. W efekcie powstają nazwy metod takie jak `shouldReturnNullWhenDaoReturnsNull()` albo `shouldReturnSingleValueReturnedByDao()` i im podobne. Dużo trudniej jest nazwać metodę: `shouldTestQuery()`. No chyba że naprawdę chcesz mnie zirytować. ;)

Spójrzmy jak wyglądałby ten sam test rozbity na kilka mniejszych metod.

```
@Test
public void shouldReturnNullListWhenDaoReturnsNull {
    ...
}

@Test
public void shouldReturnEmptyListWhenDaoReturnsIt {
    ...
}

@Test
public void shouldReturnNullSingleResultWhenDaoReturnsNull {
    ...
}

@Test
public void shouldReturnSingleResultReturnedByDao {
    ...
}
```

Zacznij od `should`. Zastanów się nad scenariuszem. Nie używaj prefiksu `test`.

5.4.5. Kiedy nazwa metody testowej kłamie

Kłamstwo nie różni się niczym od prawdy, prócz tego, że nią nie jest.

— Stanisław Jerzy Lec

Często natykam się na metody testowe z mylącymi nazwami. Na przykład na takie jak ta:

```
public void shouldInsertNewValues() {
    //given
    //when
    reportRepository.updateReport(ReportColumn.DATE,
        ReportColumn.PLACE, reportMap(BigDecimal.TEN));
    reportRepository.updateReport(ReportColumn.DATE,
        ReportColumn.PLACE, reportMap(new BigDecimal("5")));

    //then
    assertThat(reportRepository
        .getCount(ReportColumn.DATE, ReportColumn.PLACE))
        .isEqualTo(1);
}
```

Czy ten test rzeczywiście sprawdza czy *"nowe wartości zostają wstawione"*?. Nie wydaje mi się. Nie jestem w 100% pewny co ten test właściwie robi, ale odnoszę wrażenie, że on raczej weryfikuje czy nowa wartość nadpisuje starszą. Skoro tak, to powinnien chyba nosić nazwę `shouldOverrideOldReportWithNewValues`.

Czy to jest problem? Może nie jest to wielki problem, ale niby czemu test miałby kłamać? Skoro testy są dokumentacją, to test o wprowadzającej w błąd nazwie wydaje się być ponurym żartem.

Dopóki taki test przechodzi nie jest jeszcze najgorzej. Ale co gdy nagle, powiedzmy po sporej zmianie w systemie[5], ten test nagle stanie się czerwony? No, to wtedy mamy kłopot. Jak określić czy scenariusz weryfikowany w tej metodzie testowej jest nadal aktualny? Może powinniśmy usunąć ten test? A może trzeba go zmienić? Tak, pewnie trzeba go jakoś przerobić… ale jak? Pozostaje mi tylko życzyć powodzenia!

Niech nazwy metod testowych odzwierciedlają testowany scenariusz. Szczególną ostrożność należy zachować dokonując zmian w testach. Upewnij się czy nazwy metod po wprowadzeniu zmian nadal są poprawne.

5.4.6. Szczegóły implementacyjne nie są istotne

Doprawdy, szczegóły implementacyjne nie zasługują na to by trafiać do nazw metod testowych. Spójrzmy na ten przykład:

```
@Test
```

[5] A powinniśmy pamięta, że zmienność jest jedyną pewną rzeczą w procesie tworzenia oprogramowania.

```
public void shouldReturnFalseIfTransactionIsPending() {
    //given
    transaction.setState(PayoutTransactionState.PENDING);

    //when
    boolean paid = transaction.isPaid();

    //then
    assertThat(paid).isFalse();
}
```

W tym przypadku nazwa metody testowej nie wprowadza nas w błąd ale za to specyfikuje pewne nieistotne szczegóły. Spróbujmy zmienić ją tak, by koncentrowała się na scenariuszu biznesowym. Na przykład:

```
@Test
public void pendingTransactionShouldNotBeConsideredAsPaid() {

    ...
}
```

Zauważmy że jeżeli dokonamy teraz zmiany w API, w wyniku której metoda `isPaid()` nie będzie zwracałą wyniku (`void`) za to rzucała wyjątek w przypadku niepowodzenia, wówczas nazwa metody testowej nie będzie musiała ulec zmianie. I dobrze, bo i scenariusz biznesowy się nie zmienił. Zmienił się tylko sposób implementacji.

Myśląc o nazwie metody testowej zostawmy w spokoju implementację. Liczą się wymagania.

5.5. Mocki są przydatne

W sekcji *"given"* kolejnego testu tworzone są cztery obiekty. Jeden z nich to mock klasy `TrafficTrendProvider`, który jest później wykorzystywany przez SUT (obiekt klasy `TrafficService`). Pozostałe trzy obiekty tworzone są z użyciem słówka `new`. Kiedy przyjrzeć się uważniej temu, jak są wykorzystywany, okazuje się, że są one potrzebne tylko po to, by stworzyć obiekt `trafficTrend`.

```
@Test
public void shouldGetTrafficTrend() {
    //given
    TrafficTrendProvider trafficTrendProvider
            = mock(TrafficTrendProvider.class);
    Report report = new Report(null, "", 1, 2, 3,
            BigDecimal.ONE, BigDecimal.ONE, 1);
```

```
    TrafficTrend trafficTrend = new TrafficTrend(report, report,
            new Date(), new Date(), new Date(), new Date());
    given(trafficTrendProvider.getTrafficTrend()).willReturn(trafficTrend);
    TrafficService service = new TrafficService(trafficTrendProvider);

    //when
    TrafficTrend result = service.getTrafficTrend();

    //then
    assertThat(result).isEqualTo(trafficTrend);
}
```

Podstawowy problem jaki zauważam w tym teście, to fakt że tworzy on wiele prawdziwych obiektów kompletnie nieistotnych z punktu widzenia testowanej funkcjonalności. W ten sposób trwałość testu została mocno osłabiona. Dowolna zmiana w konstruktorze klas `Report` albo `TrafficTrend` spowoduje konieczność uaktualnienia testu.
Inną wadą jest to, że nagromadzenie obiektów odrywa uwagę czytelnika od sedna sprawy. Dodatkowo, dobór parametrów przekazywanych do konstruktora budzi wątpliwości (zupełnie nieistotne, ale skutecznie rozpraszające uwagę czytelnika). Co to za pusty String przekazywany jest do konstruktora klasy `Report`, i czy dla testowanego scenariusza ma jakiekolwiek znaczenie fakt, że wszystkie cztery daty przekazywane do konstruktora klasy `TrafficTrend` są identyczne?

Sugerowałbym uwolnić ten test od nieistotnych szczegółów stosując mocki w miejsce prawdziwych obiektów:

```
@Test
public void shouldGetTrafficTrend() {
    //given
    TrafficTrendProvider trafficTrendProvider
            = mock(TrafficTrendProvider.class);
    TrafficTrend trafficTrend = mock(TrafficTrend.class);
    given(trafficTrendProvider.getTrafficTrend()).willReturn(trafficTrend);
    TrafficService service = new TrafficService(trafficTrendProvider);

    //when
    TrafficTrend result = service.getTrafficTrend();

    //then
    assertThat(result).isEqualTo(trafficTrend);
}
```

Ten test jest zdecydowanie bardziej czytelny od oryginału. Już nie wodzi czytelnika na manowce. Zawiera tylko to, co istotne dla zweryfikowania poprawności pewnego scenariusza.
Dodatkowo, dzięki użyciu mocków, test stał się mniej podatny na zmiany. Teraz zmiany w konstruktorach klas `Report` i `TrafficTrend` nie wpłyną zupełnie na aktualność testu.

Tworzenie obiektów tylko po to by utworzyć inne obiekty, by z kolei można było z nich zbudować jeszcze inne obiekty? Nie, to na pewno nie jest dobry pomysł!

Czy naprawdę potrzeba konstruować te wszystkie obiekty? Jeżeli odpowiedź brzmi *"niestety tak"* to spróbujmy przynajmniej ograniczyć skalę zniszczeń. Kilka wskazówek znajdziesz w sekcji 5.3.

5.6. Asercje

> Aby mieć pewność, trzeba rozpocząć od wątpienia.
>
> — Stanisław August Poniatowski

Weryfikacja przy pomocy asercji jest sednem każdego testu. To właśnie tam sprawdzamy czy coś działa zgodnie z oczekiwaniami. Z tego powodu asercje piszemy bardzo starannie. Dodatkowo, asercje mają też wielki wpływ na wartość dokumentacyjną testów.

Przyjrzymy się teraz kilku przykładom niedoskonałych asercji. Ale nim to zrobimy, proponuję byśmy wpierw przyjrzeli się *własnym assercjom* (z ang. *custom assertions*).

[6] http://assertj.org
[7] http://hamcrest.org/

Własne asercje

Jeżeli używasz JUnit albo TestNG, to z pewnością rozpoznajesz metody takie jak `assertEquals()`, `assertTrue()`, `assertSame()` i im podobne. Przy ich pomocy można sprawdzić wszystko. Można, ale nie zawsze jest to eleganckie i czytelne. Do prostych weryfikacji te metody sa w zupełności wystarczające. Gorzej, gdy mamy do czynienia z bardziej skomplikowanymi przypadkami.

Jeżeli wyjrzysz poza JUnit i TestNG to z pewnością znajdziesz kilka dodatkowych projektów, które oferują co nieco specjalizowanych asercji. Umożliwiają one na przykład łatwą weryfikację zawartości kolekcji. To bywa przydatne, ale nie jest wystarczające. Problem w tym, że te projekty nie mają pojęcia o dziedzinie problemu stanowiącego centrum zainteresowania twoich testów.

Kolejny krok - i tym właśnie się zaraz zajmiemy - to użycie narzędzie takich jak `AssertJ`[6] albo `Hamcrest`[7]. Dostarczają one co nieco gotowych do użycia asercji, ale co ważniejsze, dostarczają możliwość tworzenia własnych. Rzecz jasna wymaga to pewnego wysiłku. Wydaje mi się, że podjęcie go jest w wielu przypadkach uzasadnione.

5.6.1. Skomplikowanym asercjom mówimy NIE!

W testach jednostkowych asercje rzadko kiedy stanowią problem. Testy te są zazwyczaj skoncentrowane na weryfikacji jednej tylko rzeczy, więc często mamy do czynienia tylko z jedną asercją. Trudno byłoby napisać ją w sposób skomplikowany.
Wyzwanie zaczyna się w testach integracyjnych, gdzie część weryfikująca potrafi urosnąć do sporych rozmiarów.

Spójrzmy na ten oto przykład takiej sytuacji. Poniższy test sprawdza czy pewien artefakt (plik `WAR`) został pomyślnie skopiowany na serwer.

```
@Test
public void shouldPreDeployApplication() {
    // given
    Artifact artifact = mock(Artifact.class);
    when(artifact.getFileName()).thenReturn("war-artifact-2.0.war");
    ServerConfiguration config = new ServerConfiguration(
            ADDRESS, USER, KEY_FILE, TOMCAT_PATH, TEMP_PATH);
    Tomcat tomcat = new Tomcat(HTTP_TOMCAT_URL, config);
    String destDir = new File(".").getCanonicalPath()
            + SLASH + "target" + SLASH;
    new File(destDir).mkdirs();
```

```
    // when
    tomcat.preDeploy(artifact, new FakeWar(WAR_FILE_LENGTH));

    //then
    JSch jsch = new JSch();
    jsch.addIdentity(KEY_FILE);
    Session session = jsch.getSession(USER, ADDRESS, 22);
    session.setConfig("StrictHostKeyChecking", "no");
    session.connect();
    Channel channel = session.openChannel("sftp");
    session.setServerAliveInterval(92000);
    channel.connect();
    ChannelSftp sftpChannel = (ChannelSftp) channel;

    sftpChannel.get(TEMP_PATH + SLASH + artifact.getFileName(), destDir);
    sftpChannel.exit();

    session.disconnect();

    File downloadedFile = new File(destDir, artifact.getFileName());

    assertThat(downloadedFile).exists().hasSize(WAR_FILE_LENGTH);
}
```

Jak widać część *"then"* tego testu jest naprawdę spora. I co gorsza, jest dość skomplikowana. Zawiera sporo niskopoziomowych szczegółów implementacyjnych, które zaciemniają obraz logiki testu. Nie tak łatwo z tego gąszczu kanałów, sesji i plików odczytać testowany scenariusz. Można zauważyć, że tak naprawdę asercja mieści się w ostatniej linii tego testu. Cała reszta części *"then"* to tylko przygotowanie do tego kulminacyjnego momentu.

Spróbujmy poprawić ten test. W pierwszym odruchu można wyodrębnić kłopotliwą część kodu w postaci pojedynczej metody prywatnej. Tak, to jest jakieś rozwiązanie, ale jak już za chwilę się przekonamy (w sekcji 5.6.2), takie rozwiązanie nie jest pozbawione wad. Spróbujmy zatem innego podejścia.

```
public void shouldPreDeployApplication() {
    //given
    Artifact artifact = mock(Artifact.class);
    when(artifact.getFileName()).thenReturn(ARTIFACT_FILE_NAME);
    ServerConfiguration config = new ServerConfiguration(
            ADDRESS, USER, KEY_FILE, TOMCAT_PATH, TEMP_PATH);
    Tomcat tomcat = new Tomcat(HTTP_TOMCAT_URL, config);

    // when
    tomcat.preDeploy(artifact, new FakeWar(WAR_FILE_LENGTH));

    // then
    SSHServerAssert.assertThat(ARTIFACT_FILE_NAME)
```

```
        .existsOnServer(config).hasSize(WAR_FILE_LENGTH);
}
```

Wygląda to o wiele lepiej. Kod testu jasno opisuje teraz **co** podlega testowaniu bez wdawania się w szczegóły **jak** to się dzieje. I dobrze.

Sugeruję stosować następującą zasadę. Jeżeli część kodu z asercjami rośnie, należy odłożyć na chwilę klawiaturę i zastanowić się jak powinna wyglądać sama asercja. Potem trzeba zastąpić cały kod tą właśnie jednoliniową asercją i postarać się ją zaimplementować korzystając z AssertJ czy Hamcresta. W wielu przypadkach okaże się to łatwiejsze i mniej pracochłonne niż się wydawało.

Po prostu wyraź to co chcesz by robiła asercja. Potem zapisz ją dokładnie w takiej formie. Na końcu zaimplementuj.

Spójrzmy na kolejny przykład (zbyt) skomplikowanej asercji.

W kodzie przedstawionym na kolejnym listingu widać, że do przeglądania listy wyników używana jest zmienna `i`. Asercje są raczej niezbyt czytelne - ja w każdym razie nie jestem w stanie szybko stwierdzić dlaczego ważne jest by trzeci element listy zawierał dwanaście *views* (czymkolwiek by one nie były).

```
@Test
public void shouldGetDataFromReport() {
    //given
    ... some complex set-up here

    //when
    List<ReportData> result
        = someDao.getReport(filter, WITH_EXCLUDED, 100, 0);

    //then
    int i = 0;
    assertThat(result.size()).isEqualTo(5);
    assertThat(result.get(i).getReportId()).isEqualTo(filter.getId());
    assertThat(result.get(i++).getViews()).isEqualTo(16);
    assertThat(result.get(i++).getViews()).isEqualTo(12);
    assertThat(result.get(i++).getViews()).isEqualTo(3);
    assertThat(result.get(i++).getViews()).isEqualTo(0);
    assertThat(result.get(i++).getViews()).isEqualTo(0);
}
```

Jak już wspominałem narzucającym się w pierwszej chwili rozwiązaniem jest zastąpienie fragmentów kodu metodami prywatnymi (o czym porozmawiamy zaraz w sekcji 5.6.2). Znów

jednak zaproponuję napisanie zamiast tego własnej asercji. Zobaczmy jakie to przyniesie rezultaty:

```
@Test
public void shouldGetDataFromReport() {
    //given
    ... some complex set-up here

    //when
    List<ReportData> result
        = someDao.getReport(filter, WITH_EXCLUDED, 100, 0);

    //then
    assertThat(result)
        .hasStatisticsForDays(5)
        .allStatisticsAreForReport(filter.getId())
        .isSortedByViews()
        .hasResultsForDay(monday, 3)
        .hasResultsForDay(tuesday, 12)
        .hasResultsForDay(wednesday, 16)
        .hasResultsForDay(thursday, 0)
        .hasResultsForDay(friday, 0);
}
```

Hm, w mojej opinii jest to znaczny krok naprzód. Sytuacja z tajemniczą `12` wygląda o wiele jaśniej. Mam nadzieje, że zgadzasz się ze mną co do tego.

Spraw by testy i asercje mówiły w języku dziedziny problemu, a nie w języku szczegółów implementacyjnych.

5.6.2. Metody prywatne jako asercje

Załóżmy, że mamy do czynienia z takim oto testem. Jest to skomplikowany test integracyjny, który pokrywa pewną niebanalną i bardzo istotną funkcjonalność systemu. Na końcu testu piszemy asercję, w której upewniamy się, że obiekty w bazie danych są takie, jak oczekiwaliśmy. Powiedzmy, że test wygląda jakoś tak:

```
@Test
public void testChargeInRetryingState() throws Exception {
    // given
    TxDTO request = createTxDTO(RequestType.CHARGE);
    AndroidTx androidTx = ...
    Processor processor = ...
    ... much more complex set-up code here
```

```
    // when
    final TxDTO txDTO = processor.processRequest(request);

    // then
    assertEquals(txDTO.getResultCode(), ResultCode.SUCCESS);
    final List<AndroidTxStep> steps = new ArrayList<AndroidTxStep>(
            androidTx.getTxSteps());
    final AndroidTxStep lastStep = steps.get(steps.size() - 1);
    assertEquals(lastStep.getTxState(),
        AndroidTxState.CHARGE_PENDING);
    assertEquals(lastStep.getMessage(), ClientMessage.SUCCESS);
    ... some more assertions here
}
```

Umówmy się, że póki co nie jest jeszcze źle. Test jest dość skomplikowany, ale wydaje się że wynika to ze złożoności problemu, więc póki co postanawiamy nic z tym nie robić. Niech zostanie tak jak jest.

Oczywiście nie jest to jedyny przypadek testowy, jaki warto zweryfikować. Pisząc kolejne testy szybko odkrywamy, że są one w zasadzie bardzo podobne do tego pierwszego. Różnica polega tylko na nieco innych oczekiwaniach w stosunku do obiektów w bazie danych. Powiedzmy, że w innym teście będziemy oczekiwać stanu `AndroidTxState.SUBMITTED` zamiast `AndroidTxState.CHARGE_PENDING`. Cóż, przypominamy sobie starą zasadę programistyczną, która mówi że re-użycie kodu jest dobre i robimy najbardziej oczywistą rzecz pod słońcem: tworzymy metodę prywatną. Ta metoda przyjmie oczekiwany stan jako argument, dzięki czemu będzie można ją użyć w obu testach.
Szybko okazuje się, że ta metoda jest naprawdę przydatna - coraz więcej testów zaczyna z niej korzystać. No bomba, po prostu bomba. Wiele testów przyjmuje teraz podobną postać (z dokładnością do przekazywanego stanu):

```
@Test
public void testChargeInRetryingState() throws Exception {
    // given
    ...

    // when
    ...

    // then
    assertState(ResultCode.SUCCESS, androidTx,
        AndroidTxState.CHARGE_PENDING, ClientMessage.SUCCESS;
}
```

I wszystko wydaje się być cudowne… no przynajmniej przez jakiś czas. Bo niestety w miarę jak powstają nowe testy okazuje się, że ta metoda jest jednak niewystarczająca. W niektórych scenariuszach (wspominałem już przecież, że dziedzina problemu jest niebanalna, prawda?)

przydałyby się nieco inne asercje niż te, które zapisaliśmy w naszej metodzie `assertState()`. Są prawie takie jak potrzeba, ale niestety nie do końca takie. Hm… Znowu powtarzamy, że przecież re-użycie kodu jest naprawdę ok i robimy naszą metodę nieco bardziej generyczną. Dzięki temu jesteśmy w stanie obsłużyć jeszcze więcej przypadków.
Koniec końców metoda `assertState()` przyjmuje taką oto postać:

```
private void assertState(TxDTO txDTO, AndroidTx androidTx,
    AndroidTxState expectedAndroidState,
    AndroidTxState expectedPreviousAndroidState,
    ExtendedState expectedState,
    String expectedClientStatus,
    ResultCode expectedRequestResultCode) {

    final List<AndroidTxStep> steps = new ArrayList<AndroidTxStep>(
        androidTx.getTxSteps());
    final boolean checkPreviousStep = expectedAndroidState != null;
    assertTrue(steps.size() >= (checkPreviousStep ? 3 : 2));

    if (checkPreviousStep) {
        AndroidTxStep lastStep = steps.get(steps.size() - 2);
        assertEquals(lastStep.getTxState(),
        expectedPreviousAndroidState);
    }

    final AndroidTxStep lastStep = steps.get(steps.size() - 1);
    assertEquals(lastStep.getTxState(), expectedAndroidState);
    assertEquals(lastStep.getMessage(), expectedClientStatus);

    assertEquals(txDTO.getResultCode(), expectedRequestResultCode);
    assertEquals(androidTx.getState(), expectedAndroidState);
    assertEquals(androidTx.getExtendedState(), expectedState);

    if (expectedClientStatus == null) {
        verifyZeroInteractions(client);
    }
}
```

Oj! Chyba zabrnęliśmy w ślepy zaułek… Zaczęliśmy z niewinną, prostą metodą prywatną, a teraz mamy tę ohydną, niezrozumiałą plątaninę asercji, która zdaje się weryfikować wyniki wszystkich możliwych scenariuszy testowych! Przyjmuje siedem parametrów i ma w sobie sporo logiki. Patrząc na nią, czujemy, że utrzymywanie jej będzie koszmarem. Nie ma szans, żeby tak po prostu przeczytać test i zrozumieć jaki scenariusz jest tak naprawdę testowany. Trzeba się naprawdę sporo wysilić by dojść do tego co, jak i dlaczego. Znajomość dziedziny problemu trochę pomoże, ale i tak nie będzie bułka z masłem. Ratunku! Nie tak miało być!

Jak możemy się uratować? Sugeruję sposób, który już polecałem: trzeba napisać własną asercję. Dzięki temu nasze testy będą mogły przyjąć dużo czytelniejszą postać:

```
public void testChargeInRetryingState() throws Exception {
    // given
    TxDTO request = createTxDTO(RequestType.CHARGE);
    AndroidTx androidTx = ...
    Processor processor = ...
    ... much more complex set-up code here

    // when
    final TxDTO txDTO = processor.processRequest(request);

    // then
    assertEquals(txDTO.getResultCode(), ResultCode.SUCCESS);
    assertThat(androidTx)
        .hasState(AndroidTxtate.CHARGED)
        .hasMessage(ClientMessage.SUCCESS)
        .hasPreviousState(AndroidTxState.CHARGE_PENDING)
        .hasExtendedState(null);
}
```

Główną zaletą tego testu nad poprzednikiem jest to, że teraz można go tak **po prostu przeczytać** i zrozumieć.

Nie powiem, żeby ten test był banalnie prosty, ale to wynika z dziedziny problemu. Sam test nie dokłada w tej mierze niczego od siebie. Z pewnością w tej wersji o wiele łatwiej zrozumieć jaki jest oczekiwany wynik. Nie trzeba w tym celu przedzierać się przez żadne wyrażenia logiczne (żadnych `if`-ów wśród asercji!). Nie ma żadnych wątpliwości co do tego, które z asercji zostaną wykonane.

Wprowadź własną asercję ilekroć czujesz, że ilość asercji rośnie ponad miarę (pytanie ile dla Ciebie wynosi ta miara: dwie linie? trzy? a może pięć? - nie powiem Ci tego; to w końcu **twój** limit, nie mój). Całkiem sensowne wydaje się też stworzenie własnych asercji dla najważniejszych obiektów domeny - istnieje spora szansa, że użyjesz ich zarówno w testach jednostkowych jak i integracyjnych.

Pozostaje pytanie, jak stworzyć wygodną w użyciu asercję? Ja po prostu piszę ją tak, żeby łatwo było ją przeczytać. Od tego zaczynam - pisze linijkę kodu, na przykład taką jak poniżej, nie przejmując się, że nie mam jeszcze implementacji:

```
assertThat(client).isVip().and().hasDiscount(0.2);
```

A potem biorę się do pisania implementacji. Oczywiście, czasami zdarza się, że muszę nieco zmienić asercję tak że powstały w wyniku DSL różni się nieco od tego wymarzonego. Trudno, czasami tak jest. Ale zazwyczaj udaje mi się zrobić dokładnie to co sobie zaplanowałem.

Uzbrojeni w tą wiedzę przejdźmy do kolejnego przykładu ilustrującego napięcie pomiędzy metodami prywatnymi a własnymi asercjami.

Oto w innym teście napotkałem taką oto metodę testową.

```
@Test
public void testCompile_32Bit_FakeSourceFile() {
    CompilerSupport _32BitCompilerSupport
        = CompilerSupportFactory.getDefault32BitCompilerSupport();
    testCompile_FakeSourceFile(_32BitCompilerSupport);
}
```

Nie jest oczywiste, jaki to scenariusz sprawdza ten test. W zasadzie nie da się tego stwierdzić, bez przyjrzenia się temu, co dzieje się w środku metody `testCompile_FakeSourceFile()`. Zajrzyjmy więc do niej:

```
private void testCompile_FakeSourceFile(CompilerSupport compilerSupport) {
    String[] compiledFiles = compilerSupport
            .compile(new File[] { new File("fake") });
    assertThat(compiledFiles, is(emptyArray()));
}
```

Hm… czy wszystko jasne? Dla mnie nie całkiem. Ciągle nie jestem pewien jaką historię chce mi opowiedzieć metoda testowa `testCompile_32Bit_FakeSourceFile`. Być może jest tak ponieważ nie mam pojęcia o dziedzinie problemu. Tak, do pewnego stopnia to właśnie jest przyczyną mojego zagubienia. Wydaje mi się jednak, że nawet lepiej orientując się w obszarze tematycznym poruszanym przez ten test, nadal miałbym kłopoty z rozumieniem go. Jestem przekonany, że można by napisać ten test w taki sposób, by łatwiej było go zrozumieć każdemu czytelnikowi.

Wydaje mi się, że kłopot bierze się stąd, że metoda `testCompile_FakeSourceFile` robi dwie rzeczy na raz:

- uruchamia jedną z metod testowanego obiektu,
- dokonuje weryfikacji wyników.

Spróbujmy pozbyć się jej całkiem. Nie będzie to trudne, jest przecież bardzo krótka.

```
@Test
public void testCompile_32Bit_FakeSourceFile() {
    CompilerSupport _32BitCompilerSupport
        = CompilerSupportFactory.getDefault32BitCompilerSupport();
    String[] compiledFiles = compilerSupport.compile(SINGLE_FAKE_FILE);
    assertThat(compiledFiles, is(emptyArray()));
}
```

Teraz, gdy widzę już całość kodu składającego się na ten test, mam kolejny pomysł na usprawnienie. Spróbujmy zmienić nazwę metody testowej oraz dopisać własną asercję, która lepiej wyrazi cel testu.

```
@Test
public void compiler32BitShouldNotBotherToCompileFakeSourceFile() {
    CompilerSupport _32BitCompilerSupport
        = CompilerSupportFactory.getDefault32BitCompilerSupport();
    String[] compiledFiles = compilerSupport.compile(SINGLE_FAKE_FILE);
    assertThat(compiledFiles).nothingWasCompiled();
}
```

Czy teraz jest lepiej? W mojej opinii o wiele. Mogę przeczytać test i zrozumieć go, bez konieczności skakania po kodzie i przeglądania innych metod.

Koszt pisania własnych asercji

Analizując dotychczasowe przykłady, w których zalecałem tworzenie własnych asercji, można zapytać o konieczność testowania ich. W końcu zdarzyć się może, że taka samodzielnie napisana asercja będzie zawierać jakąś logikę. Jak więc zyskać pewność, że nie kryje się w niej jakiś błąd?

W tej kwestii mogę się przyznać: nie piszę testów do własnych asercji. Nigdy nie czułem takiej potrzeby. Wynika to z dwóch powodów.

Po pierwsze, zazwyczaj we własnych asercjach nie pojawia się żadna logika. Ich działanie często sprowadza się do porównywania pól obiektów z oczekiwaniami. Pisząc własne asercje dla obiektów domeny raczej nie napotkasz potrzeby tworzenia żadnych pętli ani logiki w postaci wyrażeń warunkowych. Często zdarza się, że większość kodu dotyczy tworzenia bardzo czytelnych komunikatów o błędach.
Czasami w testach integracyjnych, na przykład takich jak ten omawiany w sekcji 4.5.4, asercje są nieco bardziej skomplikowane. Jednak po przeniesieniu ich do własnej klasy zostają podzielone na mniejsze metody, co zazwyczaj eliminuje złożoność.

Drugi powód jest innej natury. Otóż zazwyczaj własne asercje tworzę po tym, gdy już mam działający test z wieloma asercjami. W pewnym momencie po prostu przestaję być zadowolony z części *"then"* i wtedy zamieniam ją na własną asercję. Ta nowa klasa powstaje przez przenoszenie fragmentów kodu z oryginalnego testu. Poszczególne fragmenty trafiają do osobnych metod, i to w zasadzie tyle. Raczej trudno tu o błędy, a sytuację dodatkowo poprawia fakt, że nim przystąpię do przenoszenia kodu, mam już działający test. Gdy już przerobię go tak, by używał własnoręcznie stworzonej asercji, wystarczy go znów uruchomić by zobaczyć że wszystko nadal gra.

5.6.3. Porównywanie z oczekiwanym obiektem

Testy powinny opowiadać historię. Powinny informować jasno co robią, i jakie są ich oczekiwania. Każda część testu uczestniczy w tej opowieści dodając jakiś istotny szczegół.

Często zdarza się, że to właśnie w części z asercjami opowieść jakby nagle się urywa. Spójrzmy na poniższy przykład:

```
@Test
public void shouldReturnModelWithCorrectValuesCalculated() {
    //given
    ...

    //when
    DataModel result = ...;

    //then
    DataModel expectedResult = new DataModel<>(3.275, 1, 100);
    // value should be (1+5+50+75=131)/(10+10+10+10=40) = 3.275
    assertThat(result).isEqualTo(expectedResult);
}
```

Opuściłem części *"given"* i *"when"* bo to co naprawde interesujące dzieje się na końcu tego testu.

Nazwa metody testowej mówi nam, że w wyniku oczekujemy zwrócenia poprawnych wartości. Interesujący jest komentarz w przedostatniej linii testu, który mówi mi, co tak naprawdę było ważne dla osoby, która stworzyła ten test. Wydaje się, że liczona jest jakaś średnia, i że powinna ona wynosić `3.275`. Skoro tak, to może zamiast porównywać całe obiekty, sprawdźmy tylko tą jedną wartość.

```
    // then
    assertThat(result.getAverageValue()).isEqualTo(3.275);
```

Po tej zmianie test dużo lepiej opowiada swoją historię. Ostatnia linijka asercji pozostaje w związku z nazwą metody testowej sprawdzając poprawność dokonanych obliczeń.

Zmieniając asercję w ten sposób zyskujemy jeszcze jedno. Nie wspomnieliśmy o tym, ale oryginalny test posiada pewną słabość. Jeżeli klasa `DataModel` nie posiada rozsądnej implementacji metody `toString()`, wówczas jeżeli test nie przejdzie, dowiemy się raczej niewiele. Być może zobaczymy taki oto komunikat:

```
org.junit.ComparisonFailure:
Expected :my.company.DataModel@e6037658
Actual   :my.company.DataModel@4af66537
```

Tymczasem porównując wybrane pola zamiast całych obiektów otrzymamy dokładną informację co do różnic między nimi.

Innym problemem jest to, że taki test porównujący całe obiekty może nie przejść z powodu różnicy w zawartości obiektów zupełnie nieistotnej w kontekście rozpatrywanego scenariusza.

Jak widać porównując całe obiekty można natknąć się na pewne niespodzianki. Gdy tak się stanie do wyboruj pozostaje kilka możliwości:

- można uruchomić test ponownie w trybie debug i badając zawartość obiektów zorientować się gdzie leży przyczyna problemu,
- można dopisać rozsądną metodę `toString()`,
- lub napisać własną asercję.

Porównywanie obiektów sprawia, że warstwa biznesowa testu staje się mniej czytelna, a na pierwszy plan wysuwają się szczegóły techniczne. Historia opowiadana przez test zanika: końcówka testu mówi tylko, że jakieś dwa obiekty powinny być identyczne. Raczej sugerowałbym napisanie kilku asercji (lub stworzenie jednej własne), co pozwoli na lepsze wyrażenie intencji testu.

Rozdział 6. Co warto zapamiętać

A więc przeczytałeś już wszystkie przykłady "złych" testów, i pewnie nauczyłeś się przy okazji kilku rzeczy. Zachęcam Cię, byś przejrzał jeszcze raz książkę zatrzymując wzrok na wskazówkach i ostrzeżeniach (tych wyraźnie widocznych z ikonami po lewej). Nie bez powodu umieściłem je w książce: wierzę, że naprawdę warto je sobie przyswoić.

Poniżej jeszcze kilka wskazówek, które nie zostały wyrażone dosłownie w książce. Są jednak na tyle istotne, że polecam je Twojej uwadze.

- Wykonywanie jakiejkolwiek pracy w sposób właściwy zazwyczaj zajmuje więcej wysiłku niż robienie jej ot, tak sobie. Ale w długim okresie to się opłaca.
- Ciężko napisać test? Być może kod produkcyjny jest słaby? …a może by tak spróbować zacząć od pisania testów?
- Programowanie to praca zespołowa. Nie zapomnij przedyskutować z kolegami tych (nowych) sposobów pisania testów.
- Warto przestudiować dokumentację narzędzi, których używasz. Istnieje spora szansa, że znajdziesz tam ukryte skarby.
- Bądź pragmatyczny i opieraj się na swoich doświadczeniach. Co z tego, że ktoś (nawet autor książki) poleca Ci pewne rzeczy? Jeżeli nie czujesz korzyści z postępowania we wskazany sposób, wypracuj własne techniki i sposoby!

Kończąc chciałbym wyrazić nadzieję, że pisanie wysokiej jakości testów sprawi Ci sporo frajdy! :)

Practical Unit Testing

Jeżeli czujesz, że powinieneś dowiedzieć się więcej o testach jednostkowych, mockach, asercjach i wszystkich innych magicznych zaklęciach, które sprawią, że Twój kod będzie bezbłędny, to zapraszam do lektury moich pozostałych książek:

- *"Practical Unit Testing with **TestNG** and Mockito"*
- *"Practical Unit Testing with **JUnit** and Mockito"*

Lektura powyższych książek z pewnością pozwoli Ci pisać wysokiej jakości testy jednostkowe.

Więcej dowiesz się na stronie http://practicalunittesting.com - zapraszam!

1,2,3 KANBAN!

Jako człowiek Renesansu popełniłem też książkę o Kanbanie. Nie znajdziesz tam nic o testach za to dowiesz się bardzo ciekawych rzeczy na temat organizacji pracy zespołu. Książka w języku polskim.

Zapraszam na http://123kanban.pl.

Pomocy!

Postanowiłem, że pisząc tę książkę o testach będe opierał się wyłącznie na prawdziwym kodzie. Niestety, odnalezienie godnych uwagi fragmentów nie jest zadaniem łatwym. Moi koledzy z pracy od dawna już nie dostarczają mi nowych okazów do kolekcji. :) Podjąłem próbę znalezienia interesujących przypadków przeglądając kod projektów open-source. Okazało się jednak, że nie znając kontekstu biznesowego danego test bardzo trudno jest zdecydować czy dany kod testowy jest dobry czy zły. Inną przeszkodą jest to, że moja kolekcja "złych" testów jest już dość pokaźna - wydaje się, że znajdują się w niej już wszystkie typowe przypadki. Teraz poluję na bardziej wyszukane okazy. A te są niestety o wiele trudniejsze do znalezienia.

I dlatego też bardzo **proszę Cię o pomoc**! Być może napotkałeś kod testowy, który mógłby mnie zainteresować? Jakiś test, którego wykonywanie trwało 3 minuty zanim poprawiłeś ten czas do ułamków sekund? Jakiś pseudo-test, którego jedynym celem było podbicie metryki pokrycia kodu? 300-linijkowy potwór, który testował po prostu wszystko (tylko że nigdy nie działał)? Pełne mocków okropieństwo testujące framework do mocków zamiast kodu produkcyjnego? A może znasz test, którego wszyscy boją się dotknąć, bo tak jest kruchy? Oh, jakże chciałbym poznać bliżej te wszystkie wspaniałe testowe okropności! :)

Więc proszę, bardzo, ale to bardzo proszę: jeżeli napotkasz kawałek kodu testowego, który uznasz za interesujący, bądź tak miły i wyślij go na adres kaczanowski.tomek@gmail.com. Dziękuję!