Autor

Refaktoryzacja testów legacy w kierunku wykonywalnych specyfikacji

Część I: Podstawowy refaktoring testów

W swej pracy często spotykam się z sytuacją, gdzie zespół początkowo ogarnięty entuzjazmem do testowania automatycznego traci go, gdy testy zaczynają „przykro pachnąć". Artykuł jest pierwszym z serii tekstów poświęconych refaktoryzacji istniejących testów. W kolejnych odsłonach zaprezentuję techniki utrzymywania testów w długiej perspektywie czasu.

KONTEKST

Pisanie utrzymywalnych testów automatycznych jest niemałym wyzwaniem w dużych projektach. W niniejszym artykule chciałbym przedstawić, jak przekształcić jeden przykładowy test, który zawiera wiele błędów, a największym z nich jest brak czytelności.

Wyobraźmy sobie, że pracujemy przy takim wieloletnim projekcie, w którym wszystko było testowane głównie manualnie. Nad projektem pracowało wielu programistów, skład osobowy częściowo uległ zmianie. Każdy miał swój pomysł na to, jak powinno się testować automatycznie. Dodajmy, że – jak często zdarza się w takich przypadkach – kod produkcyjny ma nie najlepszy design i, co za tym idzie, utrzymujący go programista ma obawy przed jakąkolwiek zmianą. Przyczyna jest prosta – nikt nie chce podjąć ryzyka związanego z popełnieniem błędu. Testów automatycznych jest za mało, żeby zapewnić brak bugów, a testerzy manualni mają zajęty grafik na parę tygodni do przodu. Koło się zamyka, lepiej więc niczego nie ruszać!

MISJA

Otrzymaliśmy zadanie, które wymaga zmiany w wieloletnim kodzie. Klasa, którą musimy zmodyfikować, składa się z jednej metody rozpościerającej się na kilka tysięcy linii kodu. Na nasze szczęście, dla tej klasy istnieje test JUnit. Hurra! Będziemy mieć „siatkę bezpieczeństwa", która pomoże przy zmianach i szybko wyłapie błędy regresji – myślimy. Nic bardziej mylnego – test wygląda tak jak na Listingu 1. Czy ktoś jest w stanie w ciągu 30 sekund powiedzieć, co robi ten kod? Jeżeli nie, wówczas taki test nie spełnia swojej roli jako artefakt, który pomaga w zrozumieniu projektu.

Listing 1. Początkowa wersja testu – bardzo nieczytelna. Deklaracja metody wraz z nazwą zostały pominięte, żeby zaoszczędzić miejsce.

```
PatientFeatures patientFeatures = new PatientFeatures();
Patient patient = new Patient();
patient.setName("John Doe");
patient.setBirthDate(new DateTime(1990, 6, 12, 0, 0));
patientFeatures.setPatient(patient);
PatientHospitalizationHistory hospitalization = new
PatientHospitalizationHistory();
hospitalization.setHospitalizationDaysLastYear(5);
patientFeatures.setHospitalizationHistory(hospitalization);
PatientInsuranceHistory insuranceHistory = new
PatientInsuranceHistory();
insuranceHistory.setInsurancePlan(InsurancePlanType.
AMBULATORY);
insuranceHistory.setFirstInsuredDate(new DateTime(2008, 1, 1,
0, 0));
insuranceHistory.setActiveInsuranceStartDate(new DateTime(2012,
5, 1, 0, 0));
insuranceHistory.setActiveInsuranceEndingDate(new
DateTime(2013, 1, 1, 0, 0));
insuranceHistory.setFunds(new Money(new BigDecimal(2000),
                          Currency.getInstance("EUR")));
patientFeatures.setInsuranceHistory(insuranceHistory);
List<InsuranceProduct> products = FixtureLoader.
loadXml("bug3532-products.xml");
InsuranceProductQualifier qualifier = new
InsuranceProductQualifier();
QualificationResult result = qualifier.qualify(patientFeatures,
products);
List<QualifiedProduct> qualifiedProducts = result.
getQualifiedProducts();
Assert.assertEquals(2, qualifiedProducts);
QualifiedProduct qualifiedProduct1 = qualifiedProducts.get(0);
Assert.assertEquals(QualifiedStatus.AVAILABLE,
qualifiedProduct1.getStatus());
Assert.assertEquals(products.get(0), qualifiedProduct1.
getProduct());
QualifiedProduct qualifiedProduct2 = qualifiedProducts.get(1);
Assert.assertEquals(QualifiedStatus.AVAILABLE,
qualifiedProduct2.getStatus());
Assert.assertEquals(products.get(1), qualifiedProduct2.
getProduct());
```

Domena – sprzedaż ubezpieczeń

System zajmuje się zarządzaniem i sprzedażą ubezpieczeń zdrowotnych. W tym przypadku pojęciami domenowymi, które będą zaimplementowane jako klasy w Javie, będą: pacjent, ubezpieczenie, kwoty, ryzyko i wiele innych. Na potrzeby artykułu domena została znacznie uproszczona. Funkcjonalność, która jest testowana, to kwalifikacja produktów (w tym przypadku produktem jest ubezpieczenie). Kwalifikator decyduje o tym, które ubezpieczenia przy danych kryteriach można zaoferować pacjentowi). Aby dokonać kwalifikacji, wymaganych jest bardzo wiele informacji o pacjencie (dane personalne, historia ubezpieczenia, historia hospitalizacji itp.) oraz dostępne do zaoferowania produkty.

Klasy produkcyjne:

- **`InsuranceProductQualifier`** – zawiera algorytmy decyzyjne, które decydują o tym, czy można zaoferować klientowi to ubezpieczenie
- **`PatientFeatures`** – dane pacjenta zagregowane z różnych modułów systemu, jak i systemów zewnętrznych (wszystko, od czego może zależeć decyzja o przyznaniu produktu), składa się z
  - **`Patient`** – dane personalne pacjenta (pacjentem jest osoba posiadająca nr karty pacjenta)
  - **`PatientHospitalizationHistory`** – historia hospitalizacji pacjenta
  - **`PatientInsuranceHistory`** – historia ubezpieczenia pacjenta (w naszym systemie lub innych)
- **`InsuranceProduct`** – produkt ubezpieczeniowy, który nasz system wybrał dla klienta

  - **`QualificationResult`** – rezultat kwalifikacji (zakwalifikowane produkty są posortowane według kryterium opłacalności)
  - **`QualifiedProduct`** – zakwalifikowany produkt wraz ze statusem oraz punktacją
- **`InsuranceProductQualifierTest`** – test z Listingu 1. – który będziemy refaktoryzować

Klasy testowe:

- **`FixtureLoader`** – narzędzie testowe do wczytywania danych (InsuranceProduct) z pliku XML

Schemat działania jest zaprezentowany na Rysunku 1. Jak widać, kwalifikator przyjmuje jako parametry pacjenta i produkty (są to zwykłe struktury danych), a zwraca rezultat kwalifikacji. Działa on jak zwykła funkcja, więc powinien być prosty do przetestowania…

PROBLEMY

Nieczytelny kod

W postaci, jaką widzimy na Listingu 1., trudno stwierdzić, co właściwie ten kod testuje. Nie mamy pewności, czy ma on jakąkolwiek wartość, ani nawet tego, czy przypadkiem nie testuje *niepoprawnego* zachowania systemu. Być może kod produkcyjny zawiera błąd, a test przechodzi tylko dlatego, że oczekuje on właśnie tego błędu. Test powinien być czytelny, abyśmy mogli w ogóle zacząć zastanawiać się nad tym, czy ten przypadek jest w jakimś sensie szczególny i warty testowania. Dopiero, gdy osiągniemy czytelność kodu naszego testu, możemy zastanawiać się nad innymi celami, do których będziemy dążyć. Nie bez powodu czytelność jest wymieniona na pierwszym miejscu. Nieczytelny test sprawia więcej problemów niż pożytku i czasem najłatwiejszym sposobem naprawy jest… skasowanie go.

Duplikacja kodu

Jak z jednego kawałka kodu (w którym nie ma duplikacji) wywnioskować, że projekt ma problem z duplikacją kodu? Czasami jest to bardzo proste…

Jeżeli test został napisany w sposób, jaki widzimy na Listingu 1., musi to oznaczać jedną z 2 rzeczy:

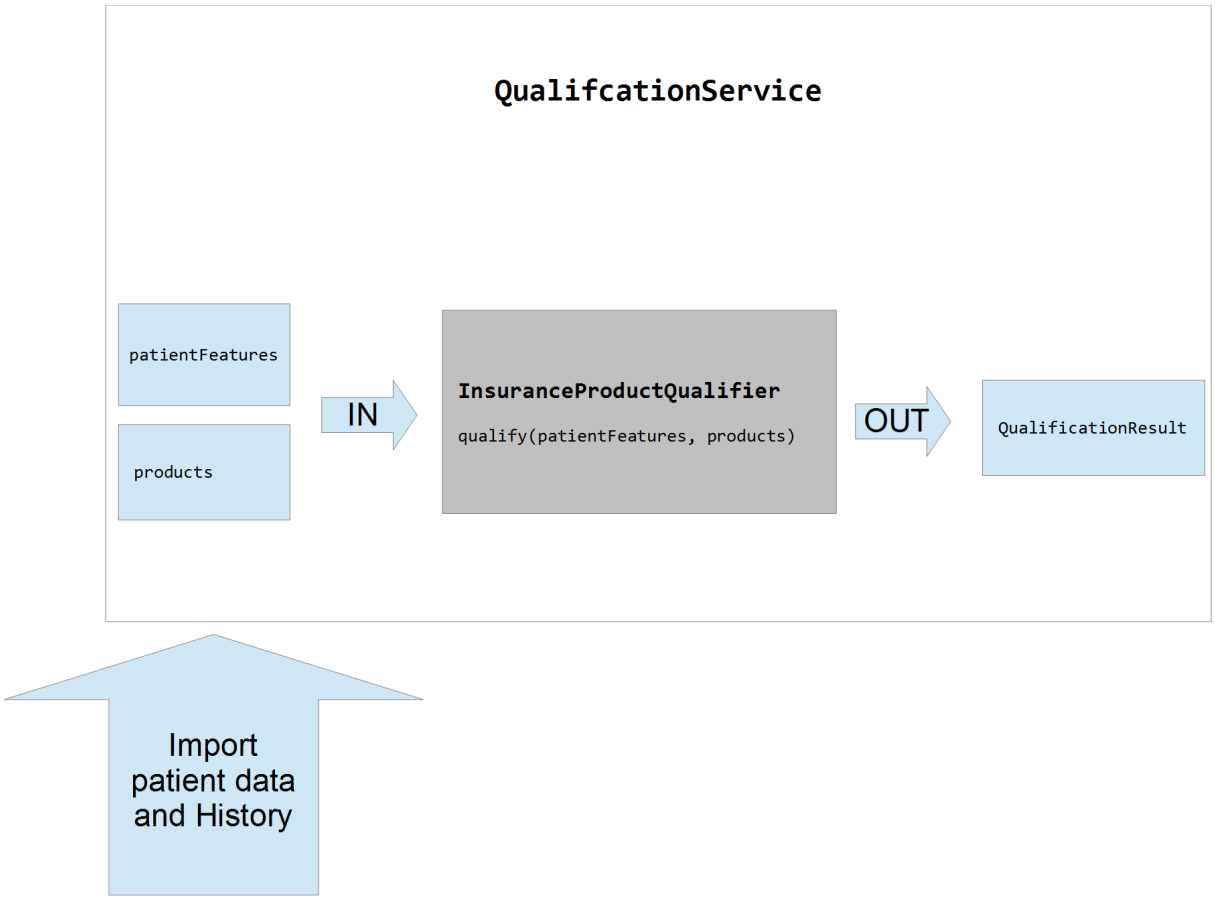
1. Inne przypadki testowe też są tak napisane

Powoduje to naturalnie bardzo dużą duplikację pomiędzy testami. Jedna zmiana w kodzie produkcyjnym pociąga za sobą lawinę zmian w kodzie testów. Takie testy szybko stają się kosztowne w utrzymaniu.

2. Istnieje tylko jeden przypadek testowy

O ile pierwszy punkt to problem w wynikający braku umiejętności programistów, to drugi to symptom problemów na poziomie całego projektu. Najprawdopodobniej automatyzacja procesu testowania jest kompletnie zaniedbana. Nic dziwnego – pisząc testy w ten sposób, niełatwo nadążyć z naprawianiem wszystkich testów przy drobnych zmianach API klas testowanych czy nawet jedynie implementacji. W takich przypadkach najłatwiej jest usunąć lub wyłączyć test. W miarę jak taka praktyka staje się normą, automatyzacja testów coraz bardziej zaczyna przypominać syzyfową pracę.

Pisanie słabych testów to jeden z głównych powodów, które wspierają mit o tym, że testy nie mogą się opłacać. Często w niektórych firmach można usłyszeć: *"kiedyś pisaliśmy testy, ale ich utrzymanie stało się po prostu zbyt drogie, więc daliśmy sobie spokój"*.



Rysunek 1. Poglądowy projekt domeny

Autor

PROBLEMY Z UTRZYMANIEM

Należy zadać sobie dwa pytania:

1. Czy jeśli implementacja nieco się zmieni, czy test będzie nadal „przechodził"? Jeżeli implementacja wciąż będzie poprawna, to test powinien być dalej „zielony". Żeby to zapewnić, test powinien używać kodu produkcyjnego tylko przez jego API (najlepiej dobrze opisane i stabilne). Im bardziej test jest związany z implementacją, tym częściej będzie się psuł, wydawałoby się – bez powodu (tzw. *Fragile Test*).
2. Jeżeli test nagle z jakiegoś powodu przestanie „przechodzić", to jaki będzie koszt jego naprawy? Programista, który będzie miał naprawić ten test, będzie musiał poświęcić dużo czasu (porównywalnie z napisaniem tego testu od nowa), żeby test znów zadziałał.

Poprawienie testu, w takis sposób, aby działał zgodnie z pierwotnymi założeniami jego twórcy jest bardzo trudne, ponieważ nie wiemy jaka była jego oryginalna intencja i co miało być zweryfikowane.

REFAKTORYZACJA TESTU

Na początku należy zadać sobie pytanie, czy nie lepiej po prostu skasować ten test. Zaoszczędzimy sobie i innym czas. Testy, które niewiele wnoszą, a często się psują, potrafią istotnie spowalniać development.

Jest jednak przypadek, w którym warto się zastanowić nad poprawą takiego testu. Być może autor napisał go na podstawie specyfikacji, a może nawet rozmowy z ekspertem, który podzielił się z nim wiedzą o tym, jak tworzony system powinien działać. Programista całą uzyskaną wiedzę biznesową zakodował w teście (dosłownie zaszyfrował!). Załóżmy, że tak było i ten przypadek testowy jest potencjalnie ciekawy i wartościowy.

Od czego zacząć

Jako, że nasz test nie jest trywialny musimy szczególnie zadbać o odpowiedni poziom abstrakcji, czyli o ukrycie szczegółów, które nic nie wnoszą do zrozumienia testu.

Ponadto, niektóre stałe i wywołania metod mogą nie mieć znaczenia w tym przypadku testowym i powinny być gdzieś ukryte (np. we wnętrzu wyekstrahowanych metod). Informacje, które nie mają wpływu na hipotezę są tylko szumem i powinniśmy się ich pozbyć.

Nazwa testu

Nazwa testu jest bardzo istotna i powinna opisywać zachowanie testowanej klasy (w tym przypadku naszego kwalifikatora). Nazwa metody testowej nie powinna pochodzić od nazwy metody testowanej, a powinna opisywać szczególny przypadek, który weryfikujemy. Dopuszczalne jest, aby nazwa była dość długa (dłuższa niż przyjęty w projekcie limit dla nazw metod produkcyjnych) Niepoprawne nazwy:

- test1, smokeTest – nazwa nic nie znaczy
- testProductManagement – nazwa jest zbyt ogólna
- testQualify – ta nazwa jest również zbyt ogólna (qualify to nazwa testowanej metody)

Poprawne nazwy:

- doesntQualifyProductIfAgeRestrictionNotMet – dokładny opis oczekiwanego zachowania
- qualifiesProductBasedOnPatientsInsuranceHistory_hasRequiredFunds – po znaku podkreślenia znajduje się opis szczególnego przypadku

Poprawna struktura

Powszechnie przyjętą konwencją jest podział testu na 4 części: przygotowanie, wykonanie, sprawdzenie i opcjonalne posprzątanie po teście. W przypadku testów jednostkowych:

- Przygotowanie (given) powinno zawierać parametry i utworzone obiekty. Jest to potencjalnie najbardziej skomplikowana część.
- Wykonanie (when) wywołuje testowaną metodę. Zazwyczaj jest to jedna linijka.
- Sprawdzenie (then) weryfikowanie hipotezę.
- Sprzątanie (tear down) występuje w testach jednostkowych tylko wtedy, gdy manipulujemy stanem globalnym (np. zmienne statyczne).

Słownictwo given, when, then wzięło się z metodyki Behavior Driven Development i jest powszechnie używane przez programistów zamiast setup, exercise, verify.

Do każdej części testu możemy dopisać nagłówki w formie komentarzy. Wymusi to poprawną strukturę i łatwiej będzie dostrzec ewentualne odstępstwa (np. given/when/then/when/then, która sugeruje, że próbujemy przetestować zbyt dużo jedną metodą testową – jest to tzw. zachłanny test).

Okazuje się, że test był już ułożony według tej struktury. Dodając komentarze, łatwiej będzie to zauważyć. Ponadto, kiedy następnym razem ktoś będzie patrzył na test, skieruje uwagę na najważniejsze części (nazwę metody testowej, when, then, a na końcu given). Widać to na Listingu 2.

Listing 2. Nadanie testowi struktury.

```
// given
// ... przygotuj dane pacjentów i produktów
// when
QualificationResult result = qualifier.qualify(patientFeatures, products);
// then
// ... zweryfikuj, że tylko te produktu zostały zakwalifikowane
```

Hipoteza testu

Cały test powinien obracać się wokół jednej hipotezy. W tym przypadku, klasa którą testujemy ma sens biznesowy (pacjenci, ubezpieczenia, kwalifikacje), najlepiej więc żeby hipoteza była sformułowana w języku biznesowym. Jeżeli wypowiem hipotezę słowami: „pacjentowi można zaoferować produkt, jeżeli fundusze pacjenta mieszczą się w zakresie wymaganym przez specyfikację produktów"

wówczas osoba nieprogramująca, ale znająca się na domenie (w tym przypadku ubezpieczeń zdrowotnych), powinna być w stanie jednoznacznie stwierdzić, czy ta hipoteza ma sens biznesowy.

Dążymy do tego, żeby w ten właśnie sposób czytało się kod testu. Żeby to osiągnąć, musimy zadbać o odpowiedni poziom abstrakcji.

KROK PIERWSZY – EKSTRAKCJA METOD

Korzystając jedynie z techniki ekstrakcji metod, jesteśmy w stanie osiągnąć bardzo dużo. Pozwoli nam to naturalnie zmniejszyć duplikację, ale co ważniejsze stworzymy metody, które mają nazwy o sensie biznesowym i widoczną intencję. Z części *given* i *then* możemy wyekstrahować metody widoczne na Listingu 3.

Listing 3. Wyekstrahowane metody.

```
private Patient createPatientBornOn(DateTime birthDate) {
   Patient patient = new Patient();
   patient.setName("John  Doe");
   patient.setBirthDate(birthDate);
   return patient;
}

private PatientHospitalizationHistory
hospitalizationHistoryWithDaysInHospital(int daysInHospital) {
   // utwórz i zwróć PatientHospitalizationHistory w
   odpowiedniej konfiguracji
}

// uwaga: jest SPECYFICZNA historia hospitalizacji
private PatientInsuranceHistory
insuredPatientHistoryWithFunds(InsurancePlanType
```

```
insurancePlanType, Money currentFunds) {
   PatientInsuranceHistory insurance = new
   PatientInsuranceHistory();
   insurance.setFirstInsuredDate(new DateTime(2008, 1, 1, 0, 0));
   insurance.setActiveInsuranceStartDate(new DateTime(2012, 6,
   1, 0, 0));
   insurance.setActiveInsuranceEndingDate(new DateTime(2013,
   1, 1, 0, 0));
   insurance.setInsurancePlan(insurancePlanType);
   insurance.setFunds(currentFunds);
   return insurance;
}

private PatientFeatures createPatientFeatures(Patient patient,
      PatientHospitalizationHistory hospitalizationHistory,
      PatientInsuranceHistory insuranceHistor) {
   // utwórz i zwróć PatientFeatures złożony z przekazanych
   parametrów
}

private void assertContainsExactProductsInOrderAsAvailable(Qual
ificationResult result, InsuranceProduct... expectedProducts) {
   // weryfikuj rezultat kwalifikacji
}
```

createPatientBornOn – pacjent urodzony zadanego dnia. Zwrócony pacjent będzie miał domyślne nazwisko, gdyż nie ma ono wpływu na reguły kwalifikacji.

createHospitalizationHistoryWithDaysInHospital – historia hospitalizacji, w której istotna jest tylko liczba dni spędzonych w szpitalu w poprzednim roku. Inne parametry nie mają znaczenia, tak jak w oryginalnej wersji testu.

createPatientFeatures – buduje cechy pacjenta z parametrów: pacjenta, historii hospitalizacji i historii ubezpieczeń. Istnieje tylko po to, żeby zmniejszyć ilość kodu w metodzie testowej.

createInsuredPatientHistoryWithFunds – tworzy specyficzną historię hospitalizacji. Ta metoda może okazać się niereużywalna, ale gdybyśmy chcieli ją sparametryzować, miałaby zbyt dużo parametrów i kod stałby się mniej czytelny. W tym miejscu poszliśmy na kompromis. W drugiej części tej serii artykułów zobaczymy, jak można rozwiązać ten problem w elegancki sposób.

assertContainsExactProductsInOrderAsAvailable – sprawdzamy czy: „w rezultacie kwalifikacji zostały zwrócone odpowiednie produkty ze statusem *dostępny* oraz w zadanej kolejności". Ważne, żeby nazwa metody odpowiadała jeden do jednego temu właśnie założeniu, do czego udało się nam doprowadzić. Efekt jest widoczny na Listingu 4.

Listing 4. Metoda testowa po wyekstrahowaniu metod.

```
// given
PatientFeatures patientFeatures = createPatientFeatures(
   createPatientBornOn(new DateTime(1990, 6, 12, 0 ,0)),
   createHospitalizationHistoryDaysInHospital(5),
   createInsuredPatientHistoryWithFunds(InsurancePlanType.
   AMBULATORY,
      new Money(new BigDecimal(2000), Currency.
      getInstance("EUR"))));

List<InsuranceProduct> products = FixtureLoader.loadFromXml(
"bug3532-products.xml");
InsuranceProductQualifier qualifier = new
InsuranceProductQualifier();

// when
QualificationResult result = qualifier.qualify(patientFeatures,
products);

// then
assertContainsOnlyThoseAvailableProductsInOrder(result,
products.get(0),
   products.get(1));
```

KROK DRUGI – BRAKUJĄCE PARAMETRY

Plik XML o nazwie **bug3532-products.xml** zawiera dane potrzebne do uruchomienia testu. Trzymanie ważnych parametrów w zewnętrznych plikach jest nieczytelne i trudne w utrzymaniu. Jeżeli nie wiemy, co jest w środku tego pliku – test czyta się następująco: "ładujemy JAKIEŚ dane z czarnej skrzynki, co powoduje, że hipoteza jest spełniona". Najlepiej gdyby wszystkie parametry, które mają wpływ na hipotezę, były widoczne w kodzie testu (i tylko one!).

Kolejnym krokiem będzie pozbycie się wczytywania danych z zewnętrznego pliku. Zamiast tego będziemy tworzyć produkty w kodzie Javy jak na Listingu 5.

Listing 5. Dane z XML przeniesione do kodu jako metody tworzące produkty.

```
List<InsuranceProduct> products = new
ArrayList<InsuranceProduct>();
products.add(createProductWithMaxAgeRestriction(
InsurancePlanType.CATASTROPHIC, 25));

products.add(createProductWithMaxAgeRestrictionAndGoals(
InsurancePlanType.AMBULATORY, 45, InsurancePlanGoalType.
VACCINATIONS));

products.add(createProductWithMaxAgeRestrictionAndGoals(
InsurancePlanType.HOSPITALIZATION, 45,
InsurancePlanGoalType.COVER_TREATMENT_EXPENSES));
```

W efekcie zmiany test zawiera więcej kodu, a co za tym idzie więcej szczegółów, np. stałych, których intencja może nie być do końca jasna. Czytelność kodu pozostała na podobnym poziome, co poprzednio (czyli jest jeszcze trochę do zrobienia). Mogłoby się wydawać, że robimy krok wstecz, ale dzięki temu zwiększymy czytelność i sprawimy, że wszystkie parametry, które są istotne dla hipotezy, są widoczne dla czytającego test (wcześniej były ukryte w zewnętrznym pliku).

Problem z reprezentacją czasu

Niektóre reguły naszego kwalifikatora wymagają porównania dat typu **org.joda.time.DateTime,** przekazanych jako parametry, z aktualnym czasem systemowym (np. żeby sprawdzić wiek pacjenta). W powyższym teście argumentami są konkretne daty, np. 12 czerwca 1990 roku. Pisanie testów w ten sposób jest niebezpieczne! To, że test „przechodzi" teraz, nie oznacza, że zadziała za kilka lat (nawet jeżeli kod się nie zmieni). Na Listingu 6. pokazane są sposoby, jak zmienić aktualną datę na potrzeby testu oraz przywrócić czas systemowy po nim. Jest to możliwe, gdyż projekt używa biblioteki joda-time. Gdyby korzystał ze standardowego API Javy (java.util.Date), zmiana aktualnego czasu nie byłaby możliwa. Uwaga: po każdym teście, który manipuluje aktualnym czasem, należy przywrócić czas systemowy. Inaczej możemy mieć problem z efektami ubocznymi w testach, które są czasami bardzo trudne do wykrycia.

Listing 6. Sposób na zatrzymanie i wznowienie upływu czasu.

```
// ustawienie aktualnej daty na 7 lipca 2012
DateTimeUtils.setCurrentMillisFixed(new DateTime("2012-07-21").
getMillis());
// przywrócenie domyślnego systemu liczenia upływu czasu
DateTimeUtils.setCurrentMillisSystem();
```

KROK TRZECI – POPRAWA API

Kod jest już napisany proceduralnie w poprawny sposób. Metody są wyekstrahowane i odpowiednio nazwane. Można teraz użyć kilku prostych trików na poprawienie czytelności. Głównym celem jest usunięcie kodu Javy, który nic nie wnosi. Na tym etapie nie usuwamy jeszcze parametrów, ani nie dzielimy testów na mniejsze – zostawimy to sobie na koniec, kiedy czytelność wzrośnie.

API wyekstrahowanych metod

Datę urodzenia pacjenta przechowujemy w systemie jako typ DateTime, ale nie oznacza to, że wyekstrahowane metody również muszą z niej korzy-

stać. Jaki typ parametru powinna przyjmować metoda createPatientBornOn, żeby było to najwygodniejsze z perspektywy osoby czytającej test i zarazem jednoznaczne? Wystarczy String! To, że argumentem jest tak naprawdę czas, wynika bezpośrednio z nazwy metody, a format podany na Listingu 7. jest domyślnym w Javie (zgodnym z ISO 8601). W tym przypadku jest to jednoznaczne i nie powinno stwarzać problemów.

To samo tyczy się typu Money. Jego konstruktor wymaga typu BigDecimal, ale do reprezentacji wartości w teście lepiej jest użyć typu integer lub double (jeżeli Money jest dobrze zaimplementowane, to nie powinno być problemów z niedokładnością dla małych liczb). Z racji tego, że metoda **createInsuredPatientHistoryWithFunds** przyjmuje kilka argumentów, lepiej wyekstrahować metodę tworzącą kwotę (metoda money(...)), a domyślną kwotę jako stała (**EUR = Currency.getInstance("EUR")**) dzięki czemu wszystko jest jednoznaczne.

Listing 7. Zmiana API metod w celu poprawy czytelności.

```
// zamiast
createPatientBornOn(new DateTime(1990, 6, 12, 0, 0))
createInsuredPatientHistoryWithFunds(..., new Money(new
BigDecimal(2000),
Currency.getInstance("EUR")))
// możemy napisać
createPatientBornOn("1990-06-12")
createInsuredPatientHistoryWithFunds(..., money(2000, EUR));
```

Deklaratywny język

Języki typu Java czy C++ nie były stworzone z myślą o programowaniu deklaratywnym, jednak możemy zbliżyć się trochę do języka naturalnego, formułując nazwy metod w sposób deklaratywny zamiast imperatywnie. Jednocześnie skróci to nieco ilość kodu, jak widać na Listingu 8. Gdyby nie deklaracja zmiennej (którą zaraz usuniemy), kod można by czytać niemalżę jak język naturalny: „given patient features, where patient is born on … and has hospitalizaton history … and is insured with funds …„

Dodatkowo należy pamiętać, że w części przygotowującej test powinny być widoczne wszystkie parametry, które mają wpływ na testowaną hipotezę, i tylko one! Reszta to niepotrzebny szum informacyjny.

Listing 8. Deklaratywny styl skraca nazwy metod i sprawia, że kod czyta się wygodniej.

```
// zamiast
// given
PatientFeatures patientFeatures = createPatientFeatures(
   createPatientBornOn(...),
   createHospitalizationHistoryDaysInHospital(...),
   createInsuredPatientHistoryWithFunds(...));
// możemy napisać
// given
PatientFeatures patientFeatures = patientFeatures(
   patientBornOn(...),
   hospitalizationHistoryDaysInHospital(...),
   insuredWithFunds(...));
```

Statyczne importy

Statyczne importy pozwalają skrócić kod, ale nie zaleca się ich nadużywania, gdyż osoba czytająca może nie wiedzieć, z jakiej klasy pochodzi metoda zaimportowana statycznie. W przypadku popularnych metod z API Javy, JUnita czy innych znanych bibliotek narzędziowych (Apache Commons, Guava) możemy korzystać z nich prawie do woli. Pisząc własne klasy należy upewnić się, że nazwy metod do zaimportowania statycznego są zrozumiałe i jednoznacznie. Jeżeli importujemy statycznie stałą i przekazujemy ją do metody, to nazwa tej metody powinna informować nas o tym, jakiego typu jest argument. Przykłady znajdują się na Listingu 9.

Listing 9. Przykład dobrego zastosowania statycznych importów.

```
// zamiast
createProductWithMaxAgeRestrictionAndGoals(InsurancePlanType.
AMBULATORY, 45,
InsurancePlanGoalType.VACCINATIONS));
Assert.assertEquals(QualifiedProductStatus.AVAILABLE,
qualifiedProduct.getStatus() );
// mozemy napiac
createProductWithMaxAgeRestrictionAndGoals(AMBULATORY, 45,
VACCINATIONS));
assertEquals(AVAILABLE, qualifiedProduct.getStatus());
```

Używanie pól klasy testowej

Popularną praktyką w JUnit jest przenoszenie zmiennych lokalnych do pól klasy testowej. Skoro i tak każda metoda testowa zawsze deklaruje takie zmienne jak **patientFeatures** czy **result**, lepiej jest je zadeklarować raz. JUnit uruchamiając następną metodę testową, stworzy nowy obiekt klasy testowej (w naszym przypadku InsuranceProductQualifierTest), więc nie będzie problemu z tym, że stan ustawiony w jednym teście będzie widziany przez testy uruchomione później (uwaga: inne frameworki do testowania, np. TestNG na jednym obiekcie klasy testowej uruchamiają wiele testów).

Zmieniając implementację metod oraz dodając metodę **qualify**, uda się nam skrócić kod testu jeszcze bardziej. Dzięki temu, że metoda **qualify** zamiast zwracania wyniku umieszcza go w polu **result**, nie musimy go przekazywać do asercji jako parametr. Efekt widać na Listingu 10.

Listing 10. Pośredni refaktoring.

```
private void patientFeatures(Patient patient,
PatientHospitalizationHistory
 hospitalizationHistory, PatientInsuranceHistory
insuranceHistory) {
   patientFeatures = new PatientFeatures();
   patientFeatures.setPatient(patient);
   patientFeatures.
   setHospitalizationHistory(hospitalizationHistory);
   patientFeatures.setInsuranceHistory(insuranceHistory);
}

private void qualify(InsuranceProduct... products) {
   InsuranceProductQualifier qualifier = new
   InsuranceProductQualifier();
   result = qualifier.qualify(patientFeatures,
   asList(products));
}

// given
patientFeatures(patientBornOn("1990-06-12"),
   hospitalizationHistoryDaysInHospital(5),
   magicInsuranceHistory(AMBULATORY, money(2000, EUR)));

InsuranceProduct product1 = productWithAge(CATASTROPHIC, 25);
InsuranceProduct product2 = productWithAgeAndGoals(AMBULATORY, 45,
   VACCINATIONS);
InsuranceProduct product3 =
productWithAgeAndGoals(HOSPITALIZATION, 45,
   COVER_TREATMENT_EXPENSES);

// when
qualify(product1, product2, product3);

// then
assertQualifiedExactProductsAsAvailable(product1, product2);
```

Tę samą technikę możemy zastosować przy tworzeniu produktów. Wszystkie produkty stworzone przez metody productWithAge i productWithAgeAndGoals będą przekazane do kwalifikatora. W przypadku tego testu nie ma sensu tworzyć produktów i go nie kwalifikować.

Jednak w tym przypadku przetrzymywanie każdego produktu w swojej zmiennej ma sens, gdyż później przekazujemy niektóre z tych produktów do asercji, żeby upewnić się, że pojawią się one w wyniku. Innym sposobem na zweryfikowanie tego, że w wyniku wystąpiły dokładnie produkty o które nam chodzi jest użycie identyfikatorów produktu. Możmy nadać identyfikator podczas tworzenia produktu, a później wywoływać asercje na tych właśnie identyfikatorach. Ostatecznie kod będzie wyglądał tak jak na Listingu 11.

Listing 11. Ostateczna wersja testu.

```
// given
patientFeatures(patientBornOn("1990-06-12"),
   hospitalizationHistoryDaysInHospital(5),
   insuredWithFunds(AMBULATORY, money(2000, EUR)));

productWithTypeAndAge("CAT01", CATASTROPHIC, 25);
productWithTypeAgeAndGoals("AMB01", AMBULATORY, 45,
VACCINATIONS);
   productWithTypeAgeAndGoals("HOSP01", HOSPITALIZATION, 45,
   COVER_TREATMENT_EXPENSES);
// when
qualify();
// then
assertQualifiedExactProductsAsAvailable("CAT01", "AMB01");
```

CZYTELNOŚĆ I CO DALEJ?

Czy nie kupiliśmy przypadkiem kota w worku?

Udało się sporo poprawić, ale to jeszcze nie oznacza, że test jest wystarczająco dobry. Czytelność jest zawsze pierwszym krokiem, bo jeśli nie jest zapewniona, test kwalifikuje się najpewniej do skasowania. Gdy już zrefaktoryzujemy test do czytelnej postaci, może okazać się że:

1. Przypadki testowe albo się powtarzają, albo nie są wcale szczególne (graniczne), a co za tym idzie – warte testowania. Testów nie należy mnożyć bez potrzeby.
2. Hipoteza nie zależy od ustawionych parametrów. Wówczas warto usunąć te, które zapewnią nam potwierdzenie lub zaprzeczenie hipotezie. Jeżeli null nie jest akceptowaną wartością, należy zastąpić ją domyślną stałą.
3. Nic nie jest weryfikowane! W ekstremalnych przypadkach zdarzają się takie sytuacje, że test, który na początku wydawał się bardzo skomplikowany (przez swoją nieczytelność), w rzeczywistości nie sprawdza żadnego sensownego zachowania klasy testowanej.

„Czytelny test" nie oznacza jeszcze „dobry test"

Jeżeli żaden z wyżej wymienionych problemów nie wystąpił, to już prawie osiągnęliśmy sukces. Prawie, gdyż należy się jeszcze upewnić, że test wnosi jakąś wartość i nie będzie sprawiał kłopotów z utrzymaniem. Można skorzystać z poniższej listy.

1. Czy testuje przypadek, który ma szansę wystąpić podczas działania systemu?
2. Czy ten przypadek jest szczególny?
3. Czy hipoteza jest jednostkowa?
4. Czy pomaga zrozumieć domenę lub kod produkcyjny?
5. Czy kontekst nie jest zbyt szeroki? (czy wszystkie parametry ustawione w części „given" mają wpływ na hipotezę)
6. Czy weryfikujemy tylko pożądane zachowanie? A nie np. zbyt restrykcyjne założenia, albo oczekujemy efektów ubocznych.
7. Czy wszystkie przypadki w teście skupiają się na wspólnym temacie? Jeżeli nie, to warto rozbić klasę testową na kilka klas zorientowanych na temat (np.: PacjenciVIP, PacjenciZalegającyZeSkładkami)
8. Czy wyekstrahowanych metod nie jest za dużo, lub nie są zbyt specyficzne?
9. Czy test nie jest zbyt mocno zależny od implementacji, ale od API (możliwie stabilnego). W przeciwnym wypadku pojawią się problemy z utrzymaniem tego testu.

Nasz test nie spełnia wszystkich tych cech, więc można go jeszcze poprawić. W następnej części serii zajmiemy się niektórymi z tych problemów i dalej będziemy ulepszać czytelność testu. Pomimo, że jest on w tym momencie już dość czytelny, to da się jeszcze dużo zrobić, aby go ulepszyć.

Rafał Jamróz rafal.jamroz@bottega.com.pl

Trener w firmie Bottega IT Solutions specjalizujący się w technologiach Java EE, JS i RoR. Zajmuje się osobistym coachingiem z zakresu technik pracy z kodem legacy i zapewnianiu jakości poprzez testowanie automatyczne. Do jego zainteresowań należą: metodyki Agile, Test Driven Development, Behaviour Driven Development.