Autor

Refaktoryzacja testów legacy w kierunku wykonywalnych specyfikacji

Część II: Techniki ułatwiające utrzymanie testów

W artykule przedstawiam techniki, które pomogą w pracy z testami w projekcie legacy (i nie tylko), a które profesjonalny programista powinien mieć w swojej „skrzynce z narzędziami", czyli m.in. Test Data Builder i Assert Object. Mimo to należy pamiętać, że zrozumienie problemu, nad którym pracujemy, jest znacznie ważniejsze niż stosowane techniki i powoduje, że testy posiadają jeszcze większą wartość.

WPROWADZENIE

Celem poprzedniej części tej serii artykułów (Programista 10/2013) było zrefaktoryzowanie testu z postaci kompletnie nieczytelnej i „zaciemnionej" do postaci w miarę zrozumiałej nawet dla programistów, którzy czytali kod testu po raz pierwszy. Zadanie było o tyle trudne, że w pierwotnej formie test był bardzo obszerny (25 linijek kodu Java) oraz posiadał wiele stałych i parametrów (kilkanaście). Udało się poprawić następujące cechy:

- nazwy metod
- nadanie testowi struktury (przygotowanie / wykonanie / weryfikacja)
- usunięcie zbędnych szczegółów (przez schowanie parametrów, które nie mają wpływu na wynik testowanej funkcjonalności)

Głównym wykorzystanym narzędziem była technika ekstrahowania metody (ang. *extract method*). Jest to najbardziej podstawowa i przydatna technika poprawy jakości i zawsze powinna być stosowana jako pierwsza, przed kolejnymi bardziej zaawansowanymi refaktoryzacjami. Ostatecznie udało się doprowadzić test do postaci podobnej do tego z Listingu 1. W tym artykule przedstawię, jak dalej poprawić ten test, aby był wartościowy i zarazem utrzymywalny.

Listing 1. Postać jednego z testów w klasie `InsuranceProductQualifierTest` po wstępnym refaktoringu

```
// given
patient = createPatientBornOn("1990-06-12");
patient.setCurrentInsurance(
        insuranceWithPlanAndFundsUntil(
        AMBULATORY, money(2000, "EUR"), "2009-12-31"));
patient.setHospitalizationHistory(lastHospitalized("2001-01-01"));
List<Insurance> insurances = ...;
// when
QualificationResult result = qualify(patient, insurances);
// then
assertTrue(result.isSuccessful());
assertEquals(new DateTime("2008-01-01"), result.
     getExpirationDate());
assertFalse(result.getNeedsFurtherVerification());
assertQualifiedWithStatus(result, "Ambulatory", PREFFERED);
assertQualified(result, "Regular");
```

Kontekst przykładu

Przykład został dobrany tak, aby przypominał kod rzeczywistego systemu i można było uwypuklić problemy, z którymi programiści spotykają się na co dzień podczas utrzymywania kodu testowego. Jeżeli system jest złożony to wymaga przetestowania jego krytycznych części w bardzo dogłębny i często skomplikowany sposób. Mnogość przypadków, które są wymagane, powoduje znaczną duplikację kodu, a to z kolei sprawia, że system usztywnia się na zmiany. Gdy zmieni się wymaganie, programiści są zmuszeni nie tylko poprawić kod produkcyjny, ale też testowy.

Celowo pominięto aspekty, spoza zakresu tego artykułu, takie jak zależności i używanie zaślepek (mock, stub), gdyż sam ten temat zasługuje na osobny artykuł

Testowana funkcjonalność

Testujemy metodę `qualify(...)` klasy `InsuranceProductQualifier`. Decyduje ona o tym, które z dostępnych produktów ubezpieczeniowych (`Insurance` – ich lista jest parametrem wejściowym) można przyznać pacjentowi na podstawie jego danych (klasa `Patient`).

Co jeszcze jest do zrobienia

Mimo że na pierwszy rzut oka test wygląda poprawnie, to wciąż może nam dostarczyć problemów w przyszłości. Zawsze należy się zastanowić, czy jeśli napiszemy kilkadziesiąt testów w podobny sposób, to nadal będą one utrzymywalne.

W tym artykule chciałbym przedstawić, w jaki sposób można zrefaktoryzować kod testu tak, aby test wnosił jakąś wartość (m.in. pomagał zrozumieć system, który jest tworzony), a nieoczekiwana zmiana wymagań nie powodowała konieczności naprawy kilkuset testów tylko dlatego, że w API metody pojawił się jeden dodatkowy parametr.

Przedstawione techniki dotyczą refaktoryzacji sekcji przygotowania testu oraz weryfikacji, gdyż zazwyczaj są to najbardziej skomplikowane części.

TECHNIKA – KLASA POMOCNICZA (TEST HELPER)

Ekstrakcja metod, choć jest najbardziej podstawową i najważniejszą techniką refaktoryzacji – czasem nie wystarczy, aby osiągnąć utrzymywalne testy. Może się okazać, że w różnych klasach testowych potrzebujemy bardzo podobnych metod prywatnych i warto by je uwspólnić, aby zlikwidować duplikację kodu. W prostych przypadkach warto przenieść metody (jako statyczne) do nowej dedykowanej klasy helpera (popularne „utilsy"). W tym celu stworzymy klasę pomocniczą `PatientTestHelper`, która będzie zawierała metody przygotowujące obiekt (pacjenta), asercje oraz stałe z nimi związane. Dzięki temu metody, które wcześniej były prywatne, stają się dostępne w każdym teście, który ich potrzebuje. Dla przykładu wszystkie metody pomocnicze i stałe związane z pacjentem z klasy `InsuranceQualifierTest` zostaną przeniesione do klasy `PatientTestHelper`. Jest to zobrazowane na Listingu 2.

Listing 2. Przykład klasy pomocniczej

```
public class PatientTestHelper {
  public final static String DEFAULT_NAME = ...;
  public final static String DEFAULT_AGE = ...;
  // inne stałe

  public Patient createPatientBornOn(String birthDate) {
    Patient patient = new Patient(...);
    patient.setBirthDate(new DateTime(birthDate));
    // ... ustaw inne atrybuty
    return patient;
  }
  // inne metody tworzące

  public void assertHas...(Patient patient, ...) {
    Assert.assertEquals(..., ...);
  }
  // inne asercje
}
```

Mimo swojej prostoty technika ta potrafi zmniejszyć duplikację oraz lepiej pogrupować kod. W momencie, gdy zdecydujemy się na przeniesienie metody do ***helpera***, warto się zastanowić, czy jej nazwa jest wystarczająco klarowna (w końcu będzie używana w wielu miejscach i przez wielu programistów).

Jeżeli z jakiegoś powodu nie chcemy zmieniać sygnatury metody na statyczną (np. używa pól klasy testowej i/lub pogorszyłoby to czytelność), to ta technika nie znajdzie zastosowania i trzeba szukać innego rozwiązania.

Zalety

- Dobre miejsce na metody tworzące, asercje i stałe związane z pewną klasą produkcyjną lub nawet funkcjonalnością (np. `AuthenticationTestHelper`)
- Refactoring przeniesienia metody jest dostępny w popularnych IDE (`move method`)

Wady

- Wymagane jest, aby metoda pomocnicza, którą przenosimy, mogła działać statycznie
- Podejście nie sprawdza się wówczas, gdy jest bardzo wiele wariantów potencjalnych parametrów. Może to skutkować:
  - zbytnim uogólnieniem (metoda z wieloma parametrami – jest to nieczytelne)
  - eksplozją kombinatoryczną przypadków (np. pacjent z peselem i diagnozą, pacjent z diagnozą i ubezpieczeniem itd.)

Technika – matka obiektów

Specjalny wariant helpera to tzw. `Object Mother`, czyli klasa, która potrafi stworzyć bardzo konkretne, reużywalne obiekty. Bardzo istotne są nazwy metod tworzących i warto w nich użyć słownictwa domenowego. Wolno nam użyć określeń w stylu „niezwykły klient", „podejrzany klient" tylko w przypadku, gdy w naszej domenie oznacza coś konkretnego i jest zrozumiałe dla wszystkich członków zespołu.

Przykładowe metody tworzące w klasie `PatientObjectMother`:

- `PatientObjectMother.vipPatient()`
- `ClientObjectMother.unusuallClient()`
- `ClientObjectMother.suspiciousClient()`

TECHNIKA: WYKORZYSTANIE PÓL (JAKO PARAMETRÓW Z WARTOŚCIAMI DOMYŚLNYMI)

W teście, nad którym pracujemy, najbardziej skomplikowaną częścią jest sekcja przygotowawcza testu. Testowana metoda przyjmuje strukturę danych (typu `Patient`) składającą się potencjalnie z kilkudziesięciu parametrów. Jednakże tylko kilkanaście z nich ma wpływ na wynik testowanej funkcjonalności (kwalifikacji pacjentów). Spośród tych kilkunastu parametrów, w danym przypadku testowym jedynie kilka ma wpływ na wynik. Trudne jest to, że w każdym teście trzeba ustawić inne kilka parametrów.

Technika, którą tu zastosujemy, wynika wprost z obserwacji, że każdy test można przedstawić w postaci takiej jak na Listingu 3. Metoda `test()` odpowiada za użycie parametrów i wykonanie całej interakcji z testowanym obiektem lub systemem. Parametrów może być potencjalnie bardzo wiele.

Listing 3. Alternatywna postać testu

```
// przygotowanie wszystkich parametrów
p1, p2, p3 = …
// cała interakcja zamknięta w jednej metodzie
wynik = test(p1, p2, p3, …)
// sprawdzenie wyniku
assert wynik == …
```

Implementacja polega na zadeklarowaniu użytecznych parametrów jako pola klasy i, jeżeli tylko jest to możliwe, nadaniu im domyślnych wartości. Wszystkie te parametry zostaną użyte jako dane wejściowe testowanej funkcjonalności. Teraz w każdym teście wystarczy tylko zmienić te kilka istotnych parametrów i sprawdzić wynik.

Warto zauważyć, że test wyglądałby bardzo podobnie w przypadku gdybyśmy potrzebowali przetestować metodę, która przyjmuje wiele parametrów (np. ponad dziesięć), co sprawia, że ta technika jest szczególnie użyteczna do testowania kodu legacy.

Listing 4. Test wykorzystujący pola jako parametry

```
// wszystkie parametry, które mogą zmieniać się
// pomiędzy testami, ustawiamy w polach
String birthDate = "1970-01-01";
int numberOfDaysInHospital = 0;
String patientName = "John Doe";
// .. inne parametry

@Test
public void test...() {
  // given – tylko te parametry,
  // które mają wpływ na hipotezę
  birthDate = "1990-06-12";
  numberOfDaysInHospital = 5;
  insurancePlanType = AMBULATORY;
  currentFunds = money(2000, EUR);
  // when
  qualify( … );
  // then
  assertThat(result, … );
}

private void qualify(InsuranceProduct... products) {
  // given – stwórz pacjenta używając pól jako parametrów
  PatientFeatures patientFeatures = createPatientFeatures();
  // WHEN
  InsuranceProductQualifier qualifier = …
  result = qualifier.qualify(patientFeatures, asList(products));
}
```

Jak widać, udało się usunąć wszystkie nieistotne szczegóły, przez co kod stał się bardziej czytelny, ale wciąż jest otwarty na dodawanie nowych, specyficznych przypadków testowych.

Zalety

- Można obsłużyć bardzo wiele przypadków małą ilością kodu
- Każdy parametr jest dobrze nazwany
- Podobne do testów sparametryzowanych, ale bardziej czytelne
- Testowanie metod z bardzo wieloma parametrami (przydatne w projektach legacy)

Uwagi implementacyjne

- Nigdy nie należy przesadzać z użyciem pól, robimy to tylko wtedy, gdy możliwy jest zysk na czytelności i łatwiejszym utrzymaniu testu

- Nie zadziała, jeżeli parametrów będzie za dużo (ale jest to już symptom nieprzemyślanego designu kodu produkcyjnego)
- Dla programistów, którzy nie spotkali się z tą techniką, może być na początku nieoczywiste, w jaki sposób wykorzystujemy pola klasy testowej

TECHNIKA: TEST DATA BUILDER

Siłą poprzedniej techniki było to, że mogliśmy stworzyć nowy przypadek testowy, kompletnie różny od pozostałych jedynie poprzez zmianę jednego lub kilku kluczowych parametrów. Wadą było to, że cała technika była zaimplementowana w klasie testu i inne klasy testowe nie mogły go zastosować bez kopiowania kodu. Dlatego właśnie kolejnym krokiem będzie przeniesienie tego mechanizmu do oddzielnej klasy, tak aby była reużywalna, otrzymując w ten sposób Test Data Builder. Na Listingu 5 można zobaczyć, jak wygląda sekcja przy przygotowaniu testu przed i po użyciu tej techniki.

Listing 5. Refaktoryzacja z użyciem Test Data Builder

```
// given – tylko wyekstrahowane metody
patient = createPatientBornOn("1990-06-12");
patient.setCurrentInsurance(
  insuranceWithPlanAndFundsUntil(
    AMBULATORY, money(2000, "EUR"), "2009-12-31"));
patient.setHospitalizationHistory(
  lastHospitalized("2001-01-01"));

// given – użyto buildera
Patient patient = buildPatient()
  .bornOn("1990-06-12")
  .withInsurancePlan(AMBULATORY)
  .withInsuranceFunds(2000, "EUR")
  .insuredUntil("2009-12-31")
  .lastHospitalizedOn("2001-01-01")
  .build();

}
```

Implementacja

Technika ta jest uproszczoną wersją wzorca budowniczego użytą na potrzeby testu. Dla własnej wygody, projektujemy API tak, aby było to możliwie najbardziej użyteczne do testowania. Dla zwiększenia czytelności testu warto zaimplementować buildera w stylu fluent API oraz skorzystać ze statycznych importów, gdzie tylko jest to możliwe. Słówko Builder nie pochodzi tutaj od wzorca projektowego Builder Pattern, a od Builder Idiom (z książki „Effective Java", J. Bloch).

Listing 6. Przykład implementacji techniki Test Data Builder

```
public class PatientBuilder {
  private int currentInsuranceFunds = 0;
  private int patientAge;
  private String activeInsuranceStartDate;
  private String firstInsuredDate;

  public Patient build() {
    Patient patient = new Patient();
    patient.setFirstInsured(new DateTime(firstInsuredDate));
    // użyć pozostałych parametrów
    return patient;
  }

  public PatientBuilder withAge(int patientAge) {
    this.patientAge = patientAge;
    return this;
  }

  public PatientBuilder withInsuranceActiveSince(
     String activeInsuranceStartDate) {
    this.activeInsuranceStartDate = activeInsuranceStartDate;
    return this;
  }

  public PatientBuilder insuredSince(String firstInsuredDate) {
    this.firstInsuredDate = firstInsuredDate;
    return this;
  }
}
```

Zalety

- Nowa warstwa abstrakcji – jeżeli sposób budowania obiektu w pewnym momencie ulegnie zmianie, to poprawiamy tylko budowniczego, a testy, które go używały, pozostają nietknięte
- Wykorzystanie domyślnych wartości sprawia, że nie musimy w teście podawać nieistotnych parametrów i test jest bardziej czytelny

Wady

- Nowa warstwa abstrakcji – czasami jest nadmiarowa i trzeba utrzymywać więcej kodu
- Metoda `build()` zwraca jeden obiekt. Użycie buildera byłoby nienaturalne, gdyby miałby produkować dwa niezależne obiekty
- Jeżeli z jakiegoś powodu builder przestanie mieć zastosowanie (np. jedna duża klasa zostanie rozbita na wiele mniejszych), to testy mogą wymagać przepisania.

Jak pisać lepsze buildery

Aby wycisnąć z tej techniki jeszcze więcej, warto zadbać o:

- Domyślne parametry
  - gdy nie ustawiliśmy parametrów, które są wymagane, builder nada domyślne wartości
  - zmniejsza ona liczbę stałych w teście i zwiększa jego czytelność
- Intencjonalne API w stylu DSL (Domain-specific language):
  - używanie języka deklaratywnego, tak jakbyśmy specyfikowali oczekiwany obiekt (efekt na Listingu 9)
  - na przykład: zamiast ustawiać datę trwania ubezpieczenia, ilość funduszy i status pacjenta, lepiej użyć metody `insured()`, jeżeli to właśnie jest sednem testu (żeby podjąć taką decyzję, musimy dysponować dostępem do źródła wiedzy domenowej)
- Mimo że nazwa techniki sugeruje, że coś jest budowane, możemy wykroczyć poza to ograniczenie i pozwolić builderowi na wchodzenie w dowolne interakcje z obiektem lub systemem (metoda `build()` może nawet łączyć się z zewnętrznymi usługami czy systemami) , np.:
  - `ExternalTrafficBuilder` – symuluje połączenia przychodzące z systemem
  - `SystemDeploymentBuilder` – uruchamia środowisko wdrożeniowe na potrzeby testu
  - `OrderingProcessBuilder` – przechodzi przez wszystkie fazy procesu składania zamówienia

Technika: Assert Object

Poprzednie techniki skupiały się na ulepszeniu części przygotowawczej testu, a teraz zajmiemy się poprawą sekcji weryfikacyjnej. Zamiast pisać asercje jako metody prywatne lub statyczne w klasie pomocniczej możemy stworzyć dedykowany obiekt (który posiada swój stan). Jest to lustrzane odbicie buildera, z tą różnicą, że zamiast specyfikować tworzony obiekt, określamy jakiego rezultatu oczekujemy. Na Listingu 7 znajduje się przykład zastosowania assert object.

Listing 7. Przykład zastosowania biblioteki assert object

```
// then - tylko wyekstrahowane metody
assertTrue(result.isSuccessful());
assertEquals(new DateTime("2008-01-01"), result.
getExpirationDate());
assertFalse(result.getNeedsFurtherVerification());
assertQualifiedWithStatus(result, "Ambulatory", PREFFERED);
assertQualified(result, "Regular");

// then - użyto assert object
new QualificationResultAssert(result).isSuccessfull()
  .validUntil("2008-01-01")
  .verified()
  .qualifiedWithStatus("Ambulatory", PREFFERED)
  .qualified("Regular").andNoMore();
```

Implementacja Assert Object jest dość prosta (przykład na Listingu 8). Wszystkie metody AO są asercjami (zwracającymi this). Każda metody zwraca this (nie ma odpowiednika metody `build()` z buildera). Przez to, że Assert Object nie tworzy żadnych obiektów, jest prostszy do napisania i zastosowania.

Listing 8. Implementacja klasy Assert Object dla rezultatów kwalifikacji

```
public class QualificationResultAssert {
  private QualificationResult result;
  private List<String> expectedQualified;
  // inne pola

  public QualificationResultAssert(QualificationResult result) {
    this.result = result;
    // ...
  }

  public QualificationResultAssert qualified(String insuranceName) {
    expectedQualified.add(insuranceName);
    // sprawdź, że element jest zakwalifikowany
    assertThat(result, ...);
    return this;
  }

  public QualificationResultAssert andNoMoreQualified() {
    // sprawdz, że tylko elementy z listy
    // expectedQualified sa zakwalifikowane w result
    assertThat(result, ...);
    return this;
  }
  // inne asercje
```

Zalety

- Prosta implementacja i użycie
- Szczególnie przydatny, gdy rezultat nie jest przystosowany do wygodnego wywoływania na nim asercji

Uwagi implementacyjne

- Assert Object nie musi się ograniczać do jednego obiektu, może np. testować cały system – wówczas w konstruktorze przekazujemy usługi do odczytu stanu (np. ExpectedSentEmailsAssert)
- Można przechowywać dodatkowy stan (np. lista obiektów, która została oznaczona jako oczekiwana), jeżeli to pomoże stworzyć bardziej przydatne AO. Nie trzeba się przejmować efektami ubocznymi, gdyż nie reużywamy AO pomiędzy testami
- Nie narzuca używanych bibliotek asercji, w implementacji metod AO możemy użyć dowolnych bibliotek asercji
- Istnieją biblioteki, które implementują podobny mechanizm (np. FEST Assert), ale nie zwalnia nas to z potrzeby pisania własnych assert objectów, gdyż tylko w ten sposób możemy napisać asercje specyficzne dla naszej domeny.

TECHNIKI TO NIE WSZYSTKO

Przedstawione techniki są najbardziej użyteczne, gdy projekt zawiera już wiele testów. Wówczas można łatwiej zauważyć części wspólne i dobrać odpowiedni refactoring. Aby poprawnie zastosować którąś z tych technik (np. Test Data Builder, Assert Object), należy też dysponować pewną wiedzą biznesową. Bez niej nie będziemy w stanie opracować dobrego API metod tak, żeby nam służyło przez dłuższy czas. Słabe API spowoduje, że często będzie trzeba je zmieniać, aby dostosować się do nowych wymagań.

Ostateczna wersja testu

Aby jeszcze bardziej poprawić kod z przykładu, musimy zaczerpnąć wiedzy domenowej. Wówczas może się okazać, że niektóre parametry nie mają dużego znaczenia (są szumem). Załóżmy, że ekspert wytłumaczył nam, o co dokładnie chodzi w tym szczególnym przypadku, który testujemy, i okazało się, że:

- pacjent powinien być:
  - dorosły
  - ubezpieczony
  - ostatnio nie hospitalizowany (w tym ani poprzednim roku)
- rezultat kwalifikacji powinien być:
  - zakończony sukcesem i niewymagający dodatkowej weryfikacji
  - produkty ubezpieczeniowe zakwalifikowały się z odpowiednim statusem

… i tylko tyle. Konkretne daty i inne dane pacjenta i produktów nie mają znaczenia w tym przypadku. Trudność polegała na zrozumieniu reguł rządzących procesem kwalifikacji, a do tego wymagana była duża wiedza biznesowa.

Kiedy już mamy tak opracowany przypadek testowy, możemy użyć tej wiedzy i prawie dosłownie przepisać ją na kod testu. Odpowiednie użycie technik jak builder czy Assert Object oraz rozważne wykorzystanie pól w klasie testowej pozwoli napisać test tak, że nawet komentarze given / when / then będą zbędne, ponieważ wystąpią w kodzie. Ostateczny efekt jest widoczny na Listingu 9.

Listing 9. Ostateczna postać testu

```
@Test
public void qualifyBasedOnAgeRestrictionsAndLimitation() {
  givenPatient().
    adult().
    insuredWithFunds().
    notHospitalizedRecently();
  givenInsurance("Ambulatory").promoted();
  givenInsurance("Catastrophic")
    .maximumAge(18).limitted();
  givenInsurance("Regular");

  whenQualificationOccurs();

  thenResult().isSuccessfull()
    .validUntil("2008-01-01")
    .verified()
    .qualifiedWithStatus("Ambulatory", PREFFERED)
    .qualified("Regular").andNoMoreQualified();
}
```

PODSUMOWANIE

Przedstawione w tym artykule techniki należy stosować z rozwagą. Dobrze jest na początek napisać kilka prostych testów i z czasem, gdy odkrywamy powtarzające się elementy („stałe fragmenty gry"), stosować kolejne, bardziej zaawansowane refaktoringi.

Wiedza domenowa bardzo pomoże nam w pisaniu utrzymywalnych testów, których nie trzeba ciągle naprawiać. To, co na początku wydawało się „ciągle zmieniającymi się wymaganiami", tak naprawdę może się okazać niewystarczająco zrozumianymi wymaganiami. Często zamiast poznać reguły panujące w domenie, patrzymy tylko na jeden aspekt, za każdym razem inny, co daje złudzenie, że wszystko ciągle jest w ruchu.

Zrozumienie domeny oczywiście może być bardzo trudne (a nawet może być najtrudniejszą częścią całego projektu!).

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.