Politechnika Wrocławska

Wydział Informatyki i Zarządzania

Kurs: **Projektowanie Oprogramowania**

Prowadzący: **Dr hab. inż. Lech Madeyski**

Testy funkcjonalne oraz akceptacyjne w środowisku Spring Web-MVC 3.x

z wykorzystaniem bibliotek JBehave oraz Selenium

Autor: Paweł Kozak

Wrocław, 2013

Wstęp

Definicja testów akceptacyjnych jest wyraźnie określona przez Prawo Polskie w Rozporządzeniu Ministra Nauki i Informatyzacji z dnia 19 października 2005r. w sprawie testów akceptacyjnych oraz badania oprogramowania interfejsowego i weryfikacji tego badania. W myśl tego rozporządzenia, test akceptacyjny polega na formalnym udokumentowaniu następujących badań ([1]):

1) wprowadzeniu do oprogramowania testowanego danych wejściowych pochodzących, w zależności od funkcjonalności oprogramowania testowanego, z przypadków testowych albo scenariuszy testowych;
2) przekazaniu do oprogramowania testowego danych, o których mowa w pkt 1;
3) porównaniu danych wyjściowych otrzymanych z oprogramowania testowego z danymi wzorcowymi opisanymi w specyfikacji przypadków testowych albo specyfikacji scenariuszy testowych.

Testy funkcjonalne z kolei nie posiadają tak precyzyjnie określonej definicji. Nie wnikają one w budowę oraz szczegóły implementacyjne badanego oprogramowania, stąd też nazywa się je testami czarnej skrzynki (Black-box testing) - zlecają one wykonanie oprogramowaniu operacji wykorzystując przekazane dane, a następnie porównują otrzymane wyniki z oczekiwaniami.

Jak widać testy akceptacyjne oraz funkcjonalne opierają się na tej samej zasadzie – można więc przeprowadzić je wykorzystując te same narzędzia. Jak jednak wykonać tego typu testy w aplikacji webowej?

Skrajnym przypadkiem jest wykorzystanie zwykłej przeglądarki internetowej i manualna analiza otrzymanych wyników w kontekście wprowadzonych danych. Rozwiązanie to ma jednak szereg wad, z których główną jest brak automatyzacji wykonywanych operacji.

W celu przeprowadzenia omawianych testów, przygotowana została biblioteka **JBehave** (http://jbehave.org/), która na podstawie scenariuszy napisanych w języku naturalnym, uruchamia testy. Ponadto, jako że, aplikacja webowa oparta jest na bezstanowym protokole HTTP, którego specyfika opiera się na modelu żądanie-odpowiedź (request-response), wykorzystana zostanie biblioteka **Selenium**, pozwalająca na łatwe manipulowanie żądaniami, oraz nawigację po uzyskanych odpowiedziach.

Testy przedstawione w niniejszym dokumencie przeprowadzone zostaną na fikcyjnej witrynie InternetShop, do której skierowane żądania zaowocować powinny wygenerowaniem strony zawierającej szczegółowe informacje o wybranym produkcie.

Zakłada się zaznajomienie czytelnika z podstawami biblioteki JUnit (odsyłam do artykułu *Testy jednostkowe w środowisku Spring 3.x z wykorzystaniem bibliotek JUnit oraz EasyMock*).

Przygotowanie środowiska

Pierwszym etapem rozpoczęcia pisania testów jest zaopatrzenie projektu w wymagane biblioteki. W projektach wykorzystujących narzędzie Maven proces ten można zautomatyzować poprzez skonfigurowanie pliku pom.xml. Konfiguracja ta widoczna jest na **Listingu 1**.

```
<dependencies>
	...
	<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-test</artifactId>
		<version>3.1.2.RELEASE</version>
	</dependency>
	<dependency>
		 <groupId>org.jbehave</groupId>
		 <artifactId>jbehave-spring</artifactId>
		 <version>3.7.1</version>
	</dependency>
	<dependency>
		<groupId>org.springframework.integration</groupId>
		<artifactId>spring-integration-core</artifactId>
		<version>2.2.0.RELEASE</version>
	</dependency>
	<dependency>
		<groupId>org.jbehave.web</groupId>
		<artifactId>jbehave-web-selenium</artifactId>
		<version>3.5.5</version>
	</dependency>
</dependencies>
```

Listing 1 Fragment konfiguracji pliku pom.xml narzędzia Maven, służący dołączeniu do projektu bibliotek JBehave oraz Selenium

Pisanie scenariuszy

Kolejnym etapem stworzenia testów akceptacyjnych, jest napisanie scenariuszy, według których przeprowadzane będą testy. Scenariusze takie napisane są w języku naturalnym – domyślnie jest to język angielski, jednak JBehave zezwala na konfigurację dopuszczającą inne języki. W przykładach wykorzystane zostaną ustawienia domyślne.

Scenariusze grupowane są w historie zawarte w plikach *.story. Struktura takiego pliku jest relatywnie prosta. Pierwsza linia pliku to krótki opis na temat tego, czego dotyczy historia. Następnie znajduje się narracja historii w postaci **In order to … As a … I want to …**

Kolejny fragment pliku to definicje scenariuszy. Całość przedstawiona została na **Listingu 2**

```
Client requests product details page

Narrative:
In order to gain informations
As a client
I want to see product details page

Scenario:
Given client wants to gain information
When he requests product page with id 1
Then product Sony Vaio VPCEB1M1E details page is displayed

Scenario:
Given client wants to gain information
When he requests product page with id 2
Then product Asus EEEPC details page is displayed

Scenario:
Given client wants to gain information
When he requests product page with id 3
Then product not found page is displayed
```

Listing 2 Zawartość pliku show_product_scenarios.story

Mapowanie kroków w Java – logika testu

Następny etap to zmapowanie każdego kroku scenariusza do Javy. Odbywa się to poprzez mechanizm adnotacji – wykorzystane zostaną trzy adnotacje na poziomie metod: **@Given**, **@When**, **@Then**. Każda z nich odpowiada za kolejny krok scenariusza.

Parametrem każdej z omawianych adnotacji jest treść kroku scenariusza którego dotyczy oznaczona nią metoda. Ponadto, dowolny fragment można zastąpić poprzez identyfikator poprzedzony znakiem dolara. Pozwoli to na przekazanie zastąpianego fragmentu scenariusza do oznaczonej metody jako parametr o tym samym identyfikatorze.

Przykład omawianej klasy zaprezentowany został na **Listingu 3**. W przykładzie wykorzystana została klasa **WebDriver**. Jest to element biblioteki **Selenium**, który pozwala na właściwą manipulację żądaniami oraz analizą odpowiedzi.

Klas ata tworzy specjalną instancję programu Firefox, która kontrolowana jest nie przez użytkownika, lecz przez wykonywany kod.

```
public class Steps {
	private String action;
	private String form_error;
	private int productId;
	private String productName;

	private WebDriver webDriver;

	public Steps() {
		webDriver = new FirefoxDriver();
	}

	@Given("client wants to gain information")
	public void gainInformation() {
	}

	@When("he requests product page with id $productId")
	public void requestProductPage(int productId) {
		this.productId = productId;
		webDriver.get("http://localhost:8080/InternetShop/product/"+productId);
	}

	@Then("product $productName details page is displayed")
	public void productDetailsPage(String productName) {
		this.productName = productName;
		WebElement title = webDriver.findElement(new By.ById("product-name"));
		assertEquals(productName + " - InternetShop.pl", webDriver.getTitle());
		assertEquals(productName, title.getText().trim());
	}
	@Then("product not found page is displayed")
	public void productNotFoundPage() {
		this.productName = "";
		assertEquals("Wybrany produkt nie istnieje - InternetShop.pl", webDriver.getTitle());
	}
}
```

Listing 3 Mapowanie kroków scenariusza do klasy Steps

Jak widać, w konstruktorze tworzymy nową instancję klasy WebDriver. W tym momencie uruchomione zostanie nowe okno programu Mozilal Firefox. W metodzie **requestProductPage()**, która odpowiada momentowi zażądania przez klienta

wybranego zasobu, każemy załadować zasób dostępny pod wskazanym adresem. Żądanie takie powinno zwrócić stronę zawierającą szczegółowe informacje o produkcie z wskazanym numerem identyfikacyjnym.

Ostatnim etapem scenariusza jest sprawdzenie czy zwrócona odpowiedź spełnia nasze oczekiwania. W tym przypadku ograniczono się jedynie do sprawdzenia czy tytuł strony oraz zawartość elementu HTML o identyfikatorze product-name spełniają założenia.

Zamieszczony kod pokazuje również przypadek scenariusza kończącego się w nieco inny sposób – produkt o wskazanym numerze identyfikacyjnym nie został odnaleziony. Jako że zarówno element @Given, jak i @When niczym się nie różnią, jedynym uzupełnieniem jakie trzeba wykonać jest stworzenie metody **productNotFoundPage()** oznaczonej adnotacją @Then dla badanego przypadku.

Konifgurowanie oraz uruchamianie historii

Za uruchomienie testów JBehave odpowiedzialna jest biblioteka JUnit. Należy więc napisać klasę, która pozwoli na odpowiednie zainicjowanie oraz uruchomienie testów.

W tym celu pomocne okazują się klasy **JUnitStory**, **JUnitStories** oraz **JUnitStoryMaps**. Kolejne wykorzystane adnotacje na poziomie klasy to:

- **@RunWith** – adnotacja zdefiniowana w bibliotece JUnit, wskazuje na klasę odpowiedzialną za uruchomienie testów. W przypadku biblioteki JBehave, odpowiednia klasa to **SpringAnnotatedEmbedderRunner**
- **@ContextConfiguration** – adnotacja zdefiniowana w framework'u Spring, służy do konifguracji kontekstu aplikacji, w jakiej uruchomione zostaną testy. W omawianym przypadku interesujący jest jedynie parametr locations, definiujący ścieżkę do pliku *.xml zawierającego ową konfigurację (definicję fasolek – JavaBeans).
- **@UsingEmbedder** – adnotacja zdefiniowana w bibliotece JBehave, służy wskazaniu oraz konfiguracji klasy Embedder, odpowiedzialenej za integrację biblioteki JBehave z środowiskiem, m.in. IDE.
- **@UsingSpring** – adnotacja zdefiniowana w bibliotece JBehave, pomocna przy konfigurowaniu środowiska do obsługi aplikacji napisanej w frameworku Spring
- **@UsingSteps** – adnotacja zdefiniowana w bibliotece JBehave, pomocna przy konfiguracji oraz wskazywaniu

Całość konfiguracji przedstawiona została na **Listingu 4**. Głównym elementem tej konfiguracji jest metoda **configuration()**. Wykorzystujemy w niej klasę **MostUsefullConfiguration** (której nazwa idealnie obrazuje jej przeznaczenie) a następnie nadpisujemy ustawienia odpowiedzialne za generowanie raportów.

Przygotowane w ten sposób testy są gotowe do uruchomienia – traktujemy klasę **TestScenarios** jako skrypt JUnit, uruchamiając go w trybie testu.

```
@RunWith(SpringAnnotatedEmbedderRunner.class)
@ContextConfiguration(locations={"/dispatcher-servlet.xml"})
@UsingEmbedder(embedder = Embedder.class, generateViewAfterStories = true, ignoreFailureInStories
= true, ignoreFailureInView = false, stepsFactory = true)
@UsingSpring()
@UsingSteps
public class TestScenarios extends JUnitStories {
	@Autowired
	private ApplicationContext context;

	public ApplicationContext getContext() { return context; }
	public void setContext(ApplicationContext context) { this.context = context; }

	@Override
	public Configuration configuration() {
		return new MostUsefulConfiguration()
		.useStoryReporterBuilder(new
	StoryReporterBuilder().withDefaultFormats().withFormats(Format.CONSOLE, Format.HTML));
	}

	@Override
	protected List<String> storyPaths() {
		return Arrays.asList("test/java/acceptance/stories/show_product_scenario.story");
	}

	@Override
	public InjectableStepsFactory stepsFactory() {
		// varargs, can have more that one steps classes
		return new InstanceStepsFactory(configuration(), new Steps());
	}
}
```

Listing 4 Implementacja klasy konfigurującej oraz uruchamiającej testy

Wykorzystane źródła

[1] Rozporządzenie Ministra Nauki i Informatyzacji z dnia 19 października 2005r. **w sprawie testów akceptacyjnych oraz badania oprogramowania interfejsowego i weryfikacji tego badania** (Dz. U. Nr 217 poz.1836 z dnia 31 października 2002 r.)

[2] Arndt Rafał, Jaśkowski Mikołaj, Szydłowska Anna, *Rodzaje testów oprogramowania*, 21.04.2008, Prezentacja wykonana na potrzeby przedmiotu PIO420 2007/2008 <http://www.staff.amu.edu.pl/~ynka/courses/PIO2007/pio-materialy/prezentacje.html>

[3] http://jbehave.org, 12.01.2012r.

[4] http://www.lordofthejars.com/2011/04/are-you-locked-up-in-world-thats-been.html, 12.01.2012r.