Politechnika Wrocławska

Wydział Informatyki i Zarządzania
Kurs: **Projektowanie Oprogramowania**
Prowadzący: **Dr hab. inż. Lech Madeyski**

Testy jednostkowe w środowisku Spring 3.x

z wykorzystaniem bibliotek JUnit oraz EasyMock

Autor: Paweł Kozak

Wrocław, 2013

Wstęp

Tworzenie testów jednostkowych dla aplikacji wykorzystującej framework Spring Web-Mvc 3.0 z wykorzystaniem pakietów JUnit (http://www.junit.org) oraz EasyMock (http://www.easymock.org) zostało mocno uproszczone w stosunku do poprzednich wersji. Dokument ten stara się pokazać proces tworzenia takich testów od strony praktycznej z wykorzystaniem mechanizmu adnotacji bibliotek JUnit 4.1 oraz EasyMock 3.1 na platformie Spring Tool Suite, opartej na platformie Eclipse.

Zaprezentowane testy przeprowadzone zostały na kontrolerze ProductsController aplikacji sklepu internetowego. Nieco uproszczony kod tego kontrolera przedstawiony został na **Listingu 1**.

```
public class ProductsController {
	public ProductsDAO getProductsDAO() { … }
	public void setProductsDAO(ProductsDAO productsDAO) { … }
	public ReviewsDAO getReviewsDAO() { … }
	public void setReviewsDAO(ReviewsDAO reviewsDAO) { … }

	@RequestMapping(value = "/product/{id}", method = RequestMethod.GET)
	public ModelAndView showProduct(@PathVariable(value="id") int id, ModelMap map) {
		try {
			Product product = productsDAO.findById(id);
			map.put("product", product);

			List<Review> reviews = reviewsDAO.find("productId = "+product.getId());
			map.put("reviews", reviews);

			map.put("review", new Review(product.getId()));

			return new ModelAndView("ProductDetailsPage");
		}
		// produktu nie znaleziono
		catch (ProductNotFoundException ex) {
			map.addAttribute("productId", id);

			return new ModelAndView("ProductNotFoundPage");
		}
	}
}
```

Listing 1 Kontroler Produktów napisany z wykorzystaniem mechanizmu adnotacji

Jak widać mapuje on jedynie jedno żądanie – próbę wyświetlenia szczegółów produktu oferowanego przez sklep – obsługą tego żądania zajmuje się metoda showProduct(), którą to będziemy testować.

Testy jednostkowe z wykorzystaniem biblioteki JUnit

Aby rozpocząć pisanie testów, należy dodać do naszego projektu bibliotekę JUnit. Jeżeli projekt wykorzystuje narzędzie Maven, można mu zlecić ściągnięcie owej biblioteki poprzez mechanizm zależności - dodanie kodu zaprezentowanego na **Listingu 2** do pliku konfiguracyjnego **pom.xml**. W przeciwnym wypadku należy ręcznie dodać bibliotekę do projektu.

```
	<dependency>
		<groupId>junit</groupId>
		<artifactId>junit</artifactId>
		<version>4.11</version>
	</dependency>
```

Listing 2 Fragment konfiguracji pliku pom.xml narzędzia Maven, służący dołączeniu do projektu bibliotek JUnit w wersji 4.11

Mając dodaną bibliotekę do projektu, można przystąpić do tworzenia klasy testującej. Klasa nazwana zostanie ProductsControllerDBTest i posiadać będzie dwie adnotacje - **@RunWith** oraz **@ContextConfiguration**.

Adnotacja **@RunWith** jest adnotacją zdefiniowaną w bibliotece JUnit i służy wskazywaniu klasy uruchamiającej testy. W środowisku Spring przygotowana została specjalna klasa **SpringJUnit4ClassRunner**, którą należy wskazać.

Kolejna adnotacja **@ContextConfiguration** zdefiniowana jest w framework'u Spring, i służy do konifguracji kontekstu aplikacji, w jakiej uruchomione zostaną testy. W naszym przypadku interesuje nas jedynie parametr locations, przez który podajemy ścieżkę do pliku *.xml zawierającego ową konfigurację (definicję fasolek – JavaBeans).

Ponadto, poza omówionymi, wykorzystamy dwie kolejne adnotacje – **@Autowired** na poziomie pola oraz **@Test** na poziomie metody. Pierwsza jest adnotacją środowiska Spring i służy łączeniu pola którego dotyczy z odpowiadającą jej fasolką z wskazanego pliku konfiguracyjnego. Łączenie to odbywa się na podstawie typu – przypisana więc zostanie fasolka której typ jest zgodny z typem pola. Oznacza to że w momencie rozpoczęcia testów, do wskazanego pola zostanie przypisana referencja gotowego do pracy obiektu. Mechanizm ten użyty zostanie do uzyskania referencji obiektu ProductsController. Jak wiemy, obiekt ten potrzebuje również obiektów klas ProductsDAO oraz ReviewsDAO – one również zostaną skonfigurowane na podstawie wskazanego pliku konfiguracyjnego.

Adnotacja **@Test** z kolei zdefiniowana jest w bibliotece JUnit i przypisywana jest do metod. Jej jedynym zadaniem jest poinformowanie klasy uruchamiającej testy, iż oznaczona metoda wykonuje test.

Kolejnym ważnym elementem testów jednostkowych są **asercje**. Asercje są to predykaty którą muszą zostać spełnione aby test wypadł pozytywnie. Biblioteka JUnit posiada całą gamę predefiniowanych asercji, jak np. **assertNotNull()** sprawdzająca czy referencja przekazanego obiektu nie jest pusta, czy **assertEquals()** testująca czy podane obiekty są identyczne – lub w przypadku liczb zmiennoprzecinkowych, czy ich różnica zawiera się w granicy błędu. Należy tutaj nadmienić, iż jeśli jedna z asercji nie zostanie spełniona, kolejne w obrębie testu nie są wykonywane.

Znając wszystkie podstawowe pojęcia oraz wiedząc w jaki sposób uruchamiane są testy, można przystąpić do ich definiowania. Zdefiniowane zostaną trzy testy :

- testShowProductWithoutReviews() – badający żądanie przeglądu szczegółów produktu nie zawierającego recenzji
- testShowProductWithReviews() – badający żądanie wyświetlenia szczegółów produktu zawierającego recenzje
- testShowProductNotExist() – badający żądanie wyświetlenia produktu nie istniejącego w bazie danych ()

Przede wszystkim należy wywołać badaną metodę showProduct() badanego kontrolera. Przyjmuje ona dwa argumenty – nr ID interesującego nas produktu oraz obiekt klasy **ModelMap** w którym zapisane zostaną dane, które następnie przesłane zostaną do widoku. Badana przez nas metoda zwraca obiekt klasy **ModelAndView**, przechowującym informację między innymi o tym, jaki widok należy uruchomić.

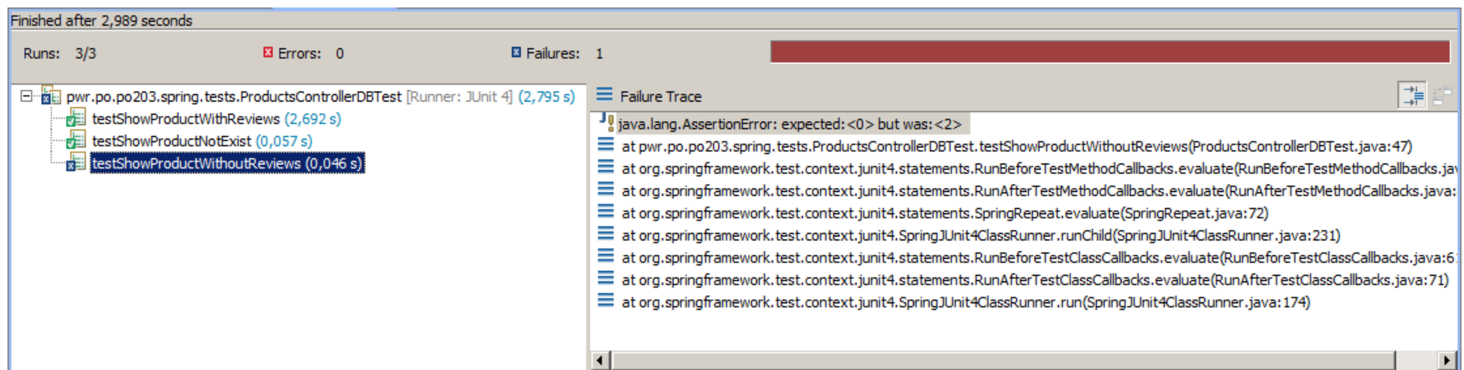
Po wywołaniu odpowiedniej metody, pobieramy uzyskane dane, a następnie sprawdzamy przy pomocy asercji czy zgadzają się one z oczekiwaniami. Zaczynamy od sprawdzenia nazwy widoku, po czym badamy czy pola produktu oraz listy recenzji mają poprawne wartości. W przypadku tym ograniczono się do badania pól produktu, ilości recenzji oraz wartości pól pierwszej recenzji.

Całość kodu testującego przedstawiona została na **Listingu 3**.

```
package pwr.po.po203.spring.tests;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

import java.util.List;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.ui.ModelMap;
import org.springframework.web.servlet.ModelAndView;

import pwr.po.po203.spring.buisnessLogic.Product;
import pwr.po.po203.spring.buisnessLogic.Review;
import pwr.po.po203.spring.controllers.ProductsController;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"/dispatcher-servlet.xml"})
public class ProductsControllerDBTest {
	@Autowired
	private ProductsController controller;

	@SuppressWarnings("unchecked")
	@Test
	public void testShowProductWithoutReviews() {
		// Testowanie Kontrolera
		ModelMap map = new ModelMap();
		ModelAndView result = controller.showProduct(2, map);

		List<Review> returnedReviews = (List<Review>)map.get("reviews");
		Product returnedProduct = (Product)map.get("product");

		// Asercje - sprawdzamy poprawnosc wyników
		assertEquals(result.getViewName(), "ProductDetailsPage");	// czy otrzymamy odpowiedni widok

		assertNotNull(returnedProduct);					// czy referencja produktu nie jest pusta
		assertEquals(returnedProduct.getId(), 2);			// czy ID produktu jest poprawny
		assertEquals(returnedProduct.getName(), "Asus EEEPC");		// czy nazwa produktu jest poprawny
		assertEquals(returnedProduct.getRetailPrice(), 799.99, 0);// czy cena produktu jest poprawna z dokładnością równą 0 (czy jest identyczna)
		assertEquals(returnedProduct.getTax(), 0.07, 0);		// czy cena produktu jest poprawna z dokładnością 0 (czy jest identyczna)
		assertEquals(returnedProduct.getTaxValue(), 0.07*799.99, 1);	// czy cena produktu jest poprawna z dokładnością równą jedności (1)

		assertNotNull(returnedReviews);// czy referencja listy opinii nie jest pusta
		assertEquals(0, returnedReviews.size());	// czy ilosc opinii jest zerowa
	}

	@SuppressWarnings("unchecked")
	@Test
	public void testShowProductWithReviews() {
		// Testowanie Kontrolera
		ModelMap map = new ModelMap();
		ModelAndView result = controller.showProduct(1, map);

		List<Review> returnedReviews = (List<Review>)map.get("reviews");
		Product returnedProduct = (Product)map.get("product");

		// Asercje
		assertEquals(result.getViewName(), "ProductDetailsPage");	// czy otrzymamy odpowiedni widok

		assertNotNull(returnedProduct);					// czy referencja produktu nie jest pusta
		assertEquals(returnedProduct.getId(), 1);			// czy ID produktu jest poprawny
		assertEquals(returnedProduct.getName(), "Sony Vaio VPCEB1M1E");	// czy nazwa produktu jest poprawny
		assertEquals(returnedProduct.getRetailPrice(), 2999.99, 0);// czy cena produktu jest poprawna z dokładnością równą 0 (czy jest identyczna)
		assertEquals(returnedProduct.getTax(), 0.07, 0);		// czy podatek produktu jest poprawny z dokładnością 0 (czy jest identyczna)
		assertEquals(returnedProduct.getTaxValue(), 0.07*2999.99, 1);	// czy cena wartość podatku produktu jest poprawna z dokładnością równą
jedności (1)

		assertNotNull(returnedReviews);					// czy referencja listy opinii nie jest pusta
		assertEquals(returnedReviews.size(), 4);			// czy ilosc opinii jest odpowiednia
		assertEquals(returnedReviews.get(0).getId(), 1);		// czy ID pierwszej opinii się zgadza
		assertEquals(returnedReviews.get(0).getProductId(), 1);		// czy ID produktu którego dotyczy opinia się zgadza
		assertEquals(returnedReviews.get(0).getTitle(), "Bardzo fajny!");// czy tytuł opinii się zgadza
		assertEquals(returnedReviews.get(0).getContent(), "Laptop jest swietny! Mam go 3 lata i nadal mnie zadziwia!");// czy treść opinii się zgadza
		assertEquals(returnedReviews.get(0).getRating(), 5);		// czy ocena opinii pierwszej opinii się zgadza
	}


	@Test
	public void testShowProductNotExist() {
		// Testowanie Kontrolera
		ModelMap map = new ModelMap();
		ModelAndView result = controller.showProduct(3, map);

		int returnedProductId = (Integer)map.get("productId");

		// Asercje - sprawdzamy poprawnosc wyników
		assertEquals(result.getViewName(), "ProductNotFoundPage");	// czy otrzymamy odpowiedni widok
		assertEquals(returnedProductId, 3);				// czy otrzymamy nr ID produktu jest poprawny
	}
}
```

Listing 3 Testy jednostkowe kontrolera Spring Web-MVC wykonane przy użyciu konfiguracji aplikacji

W celu uruchomienia stworzonego testu, należy uruchomić go w trybie JUnit Test poprzez wybranie opcji: **Run -> Run As -> JUnit Test**. Można również do tego celu wykorzystać skrót klawiszowy **ALT+SHIFT+X,T**. Wyniki przeprowadzonych testów przedstawione zostały na **Rysunku 1**. Jak widać, testy testShowProductWithReviews() oraz testShowProductNotExist() wykonane zostały pomyślnie, natomiast wynik testu testShowProductWithoutReviews() jest negatywny. Prawa ramka na omawianym rysunku zawiera ścieżkę wywołań, w której znajduje się niespełniona asercja, a także wartość oczekiwaną oraz faktyczną.



Rysunek 1 Wykonanie testów ProductsControllerDBTest w środowisku Spring Tool Suite

Testy jednostkowe z wykorzystaniem bibliotek JUnit oraz EasyMock

Jak widać omówiony sposób przeprowadzania testów jest relatywnie prosty. Posiada on jednak wady:

- Dane pobierane są z bazy danych, nie można więc zagwarantować czy nie uległy one zmianie, co może doprowadzić do negatywnego wyniku testu mimo iż wszystkie operacje wykonane zostały poprawnie
- Pierwsze wywołanie testu związane jest z inicjalizowaniem oraz konfigurowaniem środowiska, co ma spory wpływ na czas w jakim testy się zakończą – jak widać, pierwszy test wykonywał się ponad 50 razy dłużej niż pozostałe

Sposobem na rozwiązanie tych problemów jest użycie obiektów naśladujących (**Mock**) w miejsce wykorzystywanych pakietów. W omawianym przypadku, rozwiązanie to polegać będzie na rezygnacji z automatycznej konfiguracji na rzecz ręcznego definiowania parametrów obiektów – w szczególności wykorzystana zostanie biblioteka **EasyMock** która imitować będzie działanie obiektów klas ProductsDAO oraz ReviewsDAO.

Podobnie jak w przypadku biblioteki JUnit, należy rozpocząć od dołączenia do projektu wymaganych bibliotek. Również można wykorzystać mechanizm zależności narzędzia Maven, dodając kod widoczny na **Listingu 4**. Można również dodać bibliotekę ręcznie.

```
	<dependency>
		<groupId>org.easymock</groupId>
		<artifactId>easymock</artifactId>
		<version>3.1</version>
	</dependency>
```

Listing 4 Fragment konfiguracji pliku pom.xml narzędzia Maven, służący dołączeniu do projektu biblioteki EasyMock w wersji 3.1

W celu wygodnego wykorzystania możliwości oferowanych przez bibliotekę, należy ją statycznie zaimportować w pliku testowym. Każde wywołanie którejkolwiek metody obiektu imitującego przed jego aktywacją (wykonaniem na nim operacji **replay()**), powoduje iż wywołanie to będzie oczekiwane przez bibliotekę po jego aktywacji. Sprawdzenie czy wywołanie to faktycznie się odbyło dokonywane jest poprzez operację **verify()**. Ponadto operacja **expect()**, pozwala zdefiniować dodatkowe informacje dotyczące wywołania – jak np. wartość jaka ma zostać zwrócona, czy wyjątek jaki ma zostać wyrzucony. Funkcje biblioteki które zostaną wykorzystane w prezentowanym przykładzie to:

- **EasyMock.createMock()** – tworzy obiekt imitujący instancję podanej w argumencie klasy
- **EasyMock.expect()** – informuje o tym iż oczekujemy wywołania metody obiektu Mock dla konkretnych parametrów. Funkcja ta zwraca obiekt klasy IExpectationsSetters, który można wykorzystać do zdefiniowania akcji jakie wykona wywołana funkcja (np. zwrócony wynik, wyrzucone wyjątki itd.).
- **EasyMock.replay()** – przeprowadza obiekt Mock w stan aktywny – oznacza to iż możemy wykonać test
- **EasyMock.verify()** – sprawdzenie czy oczekiwana przez nas metoda obiektu imitującego faktycznie została wywołana.
- **IExpectationsSetters.andReturn()** – zdefiniowanie wyniku wywołania metody o zadanych parametrach
- **IExpectationsSetters.andThrow()** – zdefiniowanie wyjątku jaki wyrzuca wywołanie metody o zadanych parametrach

Listng 5 pokazuje praktyczny przykład wykorzystania omawianych funkcji, dla obiektu reviewsDaoMock imitującego połączenie z bazą danych. Definiujemy w nim wywołanie metody find() imitowanego obiektu z argumentem typu String „productId = 1”. Wywołanie takie zakończy się zwróceniem pustej listy ArrayList obiektów klasy Review .

```
/*
 * Oczekujemy wywolania metody find() z parametrem "productId = 1" obiektu reviewsDaoMock
 * Obiekt Mock dla danego wywolania zwroci pusta ArrayListe obiektow klasy Review
 */
ReviewsDAO reviewsDaoMock = createMock(ReviewsDAO.class);
expect(reviewsDaoMock.find("productId = 1")).andReturn(new ArrayList<Review>());
replay(productsDaoMock);  // aktywacja obiektu productsDaoMock

// ... wykonanie testów

verify(reviewsDaoMock);         // Upewniamy sie ze oczekiwane metody zostaly wywolane
```

Listing 5 Przykład wykorzystanie biblioteki EasyMock w celu imitacji obiektu implementującego interfjes ReviewsDAO

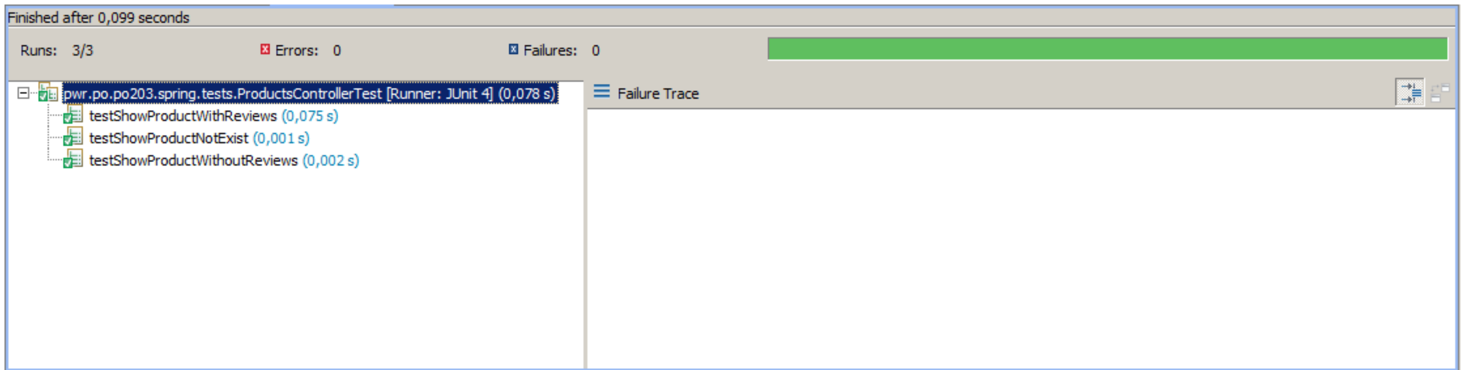
Jako że przed wykonaniem funkcji **expect** nie aktywowano obiektu (funkcją **replay**), wywołanie metody imitowanego obiektu nie przyniesie żadnych efektów – w szczególności nie wyrzuci ono wyjątków (co okaże się istotne w dalszych przykładach).

W celu przypisania zdefiniowanych obiektów imitujących DAO do kontrolera ProductsController, należy posłużyć się adnotacją **@Before** biblioteki JUnit, informującą o metodzie która wykonana zostanie przed rozpoczęciem testów.

Listing 6 pokazuje przykład testów testShowProductWithReviews() oraz testShowProductNotExist() wykonanych z wykorzystaniem omawianych technik zdefiniowanych w klasie ProductsControllerTest.

```java
package pwr.po.po203.spring.tests;

import ... ;

public class ProductsControllerTest {
	private ProductsController controller;
	private ProductsDAO productsDaoMock;
	private ReviewsDAO reviewsDaoMock;

	@Before
	public void setup() {
		controller = new ProductsController();
		productsDaoMock = createMock(ProductsDAO.class);
		reviewsDaoMock = createMock(ReviewsDAO.class);
		controller.setProductsDAO(productsDaoMock);
		controller.setReviewsDAO(reviewsDaoMock);
	}

	@SuppressWarnings("unchecked")
	@Test
	public void testShowProductWithReviews() {
		try {
			/*
			 * Oczekujemy wywolania metody findById() z parametrem 2
			 * Obiekt Mock dla danego wywolania zwroci zdefiniowany obiekt klasy Product
			 */
			expect(productsDaoMock.findById(2)).andReturn(new Product(2, "nazwa", "opis", 99.99, 0.07));

			ArrayList<Review> reviews = new ArrayList<Review>(); // tworzymy liste Review ktora bedziemy oczekiwac od obiektu Mock
			reviews.add(new Review(1, 2, "opinia 1", "tresc opinii 1", 5));
			reviews.add(new Review(2, 2, "opinia 2", "tresc opinii 2", 4));
			reviews.add(new Review(3, 2, "opinia 3", "tresc opinii 3", 2));
			reviews.add(new Review(4, 2, "opinia 4", "tresc opinii 4", 3));
			expect(reviewsDaoMock.find("productId = 2")).andReturn(reviews);
		} catch (ProductNotFoundException e) {
			e.printStackTrace();  // ta instrukcja nigdy nie zostanie wykonana - nie wykonano operacji replay na obiekcie productsDaoMock
		}
		replay(productsDaoMock);
		replay(reviewsDaoMock);

		// Testowanie Kontrolera
		ModelMap map = new ModelMap();
		ModelAndView result = controller.showProduct(2, map);

		List<Review> returnedReviews = (List<Review>)map.get("reviews");
		Product returnedProduct = (Product)map.get("product");

		// Upewniamy sie ze oczekiwane metody zostaly wywolane
		verify(productsDaoMock);
		verify(reviewsDaoMock);

		// Asercje
		assertEquals(result.getViewName(), "ProductDetailsPage");       // czy otrzymamy odpowiedni widok

		assertNotNull(returnedProduct);                                  // czy referencja produktu nie jest pusta
		assertEquals(returnedProduct.getId(), 2);                        // czy ID produktu jest poprawny
		assertEquals(returnedProduct.getName(), "nazwa");                // czy nazwa produktu jest poprawny
		assertEquals(returnedProduct.getDescription(), "opis");          // czy opis produktu jest poprawny
		assertEquals(returnedProduct.getRetailPrice(), 99.99, 0);// czy cena produktu jest poprawna z dokładnością równą 0 (czy jest identyczna)
		assertEquals(returnedProduct.getTax(), 0.07, 0);                 // czy podatek produktu jest poprawny z dokładnością 0 (czy jest identyczna)
		assertEquals(returnedProduct.getTaxValue(), 0.07*99.99, 1);// czy cena wartość podatku produktu jest poprawna z dokładnością równą jedności
(1)
		assertNotNull(returnedReviews);                                  // czy referencja listy opinii nie jest pusta
		assertEquals(returnedReviews.size(), 4);                         // czy ilosc opinii jest odpowiednia
		assertEquals(returnedReviews.get(0).getId(), 1);                 // czy ID pierwszej opinii się zgadza
		assertEquals(returnedReviews.get(0).getProductId(), 2);          // czy ID produktu którego dotyczy opinia się zgadza
		assertEquals(returnedReviews.get(0).getTitle(), "opinia 1");     // czy tytuł opinii się zgadza
		assertEquals(returnedReviews.get(0).getContent(), "tresc opinii 1");// czy treść opinii się zgadza
		assertEquals(returnedReviews.get(0).getRating(), 5);             // czy ocena opinii pierwszej opinii się zgadza
	}


	@SuppressWarnings("unchecked")
	@Test
	public void testShowProductNotExist() {
		// Przygotowanie Mock obiektów DAO
		try {
			expect(productsDaoMock.findById(3)).andThrow(new ProductNotFoundException(3));
		} catch (ProductNotFoundException e) {
			e.printStackTrace();
		}
		replay(productsDaoMock);        // aktywacja obiektu productsDaoMock

		// Testowanie Kontrolera
		ModelMap map = new ModelMap();
		ModelAndView result = controller.showProduct(3, map);

		int returnedProductId = (Integer)map.get("productId");

		verify(productsDaoMock);        // Upewniamy sie ze oczekiwane metody zostaly wywolane

		// Asercje - sprawdzamy poprawnosc wyników
		assertEquals(result.getViewName(), "ProductNotFoundPage");       // czy otrzymamy odpowiedni widok
		assertEquals(returnedProductId, 3);                              // czy otrzymamy nr ID produktu jest poprawny
	}
}
```

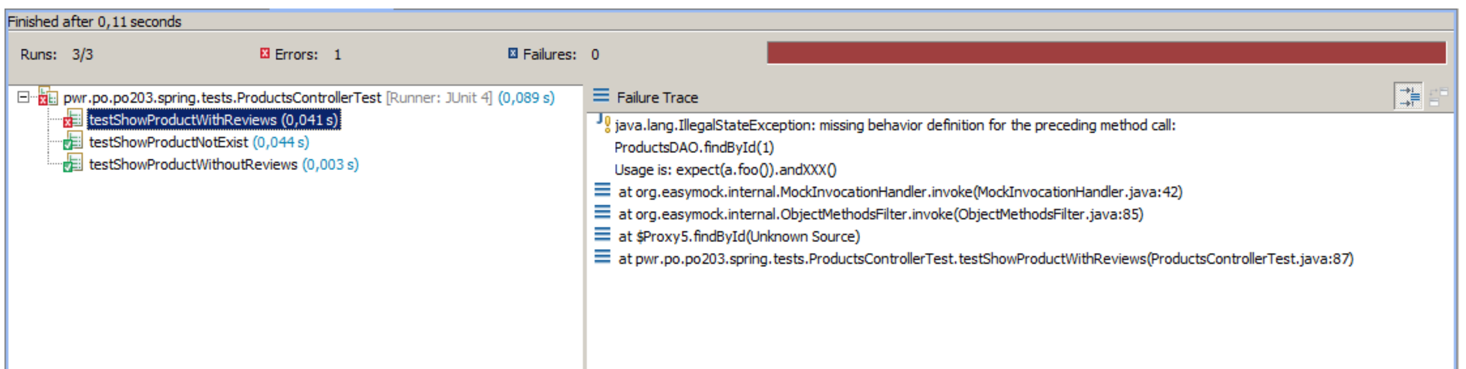
Listing 6 Testy jednostkowe kontrolera Spring Web-MVC wykonane z wykorzystaniem biblioteki EasyMock



Rysunek 2 Wykonanie testów ProductsControllerTest w środowisku Spring Tool Suite

Rysunek 2 pokazuje pozytywne przejście wszystkich zdefiniowanych testów. Widać na nim, iż wykorzystanie omawianej biblioteki nie tylko wpłynęło na ograniczenie czasu konfiguracji testów, ale także na drastyczne zmniejszenie czasu wykonywania pojedynczych testów. Sumarycznie czas zmniejszył się aż 35 razy, a w przypadku większej ilości testów, współczynnik ten będzie rosnąć. Wpływ na to ma fakt nie tylko ominięcia czasochłonnej konfiguracji aplikacji, ale również zrezygnowania z komunikacji z serwerem bazy danych i oczekiwania na jego odpowiedzi.

Co się jednak stanie w przypadku gdy oczekiwana metoda imitowanego obiektu nie zostanie wywołana? Wynik działania takiego przypadku zaprezentowanych został na **Rysunku 3**. Jak widać efekt jest identyczny z negatywnym wynikiem asercji - otrzymujemy pełne informacje o niespełnionym oczekiwaniu.



Rysunek 3 Wykonanie testów ProductsControllerTest w środowisku Spring Tool Suite

Wykorzystane źródła

[1] http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/testing.html, 05.01.2013

[2] https://github.com/KentBeck/junit/wiki, 05.01.2013

[3] http://www.easymock.org/EasyMock3_1_Documentation.html, 05.01.2013

[4] http://www.easymock.org/api/easymock/3.1/index.html, 05.01.2013

[5] http://www.jroller.com/habuma/entry/unit_testing_spring_mvc_it, 05.01.2013