INTRODUCTION TO SPRING MVC TEST FRAMEWORK

Avoid the danger which has not yet come

Testing is Essential

Industry Average: "about 15 - 50 errors per 1000 lines of ***delivered code***.“ [1980]

Microsoft Applications: finds "about 10 - 20 defects per 1000 lines of code ***during in-house testing***. [1992]

SDI – “Star Wars”

…the X-rays emitted by an exploding H-bomb would be focused on the travelling missile. One of the many technical issues was that the device involved some millions of lines of computer code, and could not be tested in advance. ***One computer expert said that the maximum number of lines of untested code without a bug was about 9;***

Test Driven Development

The strictest definition of Test Driven Development (TDD) is to always write the tests first, then the code.

A looser interpretation is Test Oriented Development (TOD), where we alternate between writing code and tests as part of the development process.

The most important thing is for a codebase to have as complete a set of unit tests as possible

The quality of tests is always higher when they are written at around the same time as the code that is being developed

Unit Testing

Unit testing is so widespread, developers who don't practice it should hold their heads down in shame.

Wikipedia defines unit testing as:

A software testing method by which individual units of source code... are tested to determine whether they are fit for use.

Requires stubbing dependencies of the method/unit. Typically, stubbing is done through a library/tool that ***mocks*** dependency objects.

A mock is ACTUALLY more that a stub, ***it verifies behavior****…*

Martin Fowler: Mocks Aren't Stubs

Test Doubles

any kind of pretend object used in place of a real object for testing purposes.

TYPES

Dummy objects are passed around but never actually used. Usually they are just used to fill parameter lists.

Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory database is a good example).

Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.

Mocks objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

Mockito

Tasty mocking framework for unit tests in Java

Mockito

Enables mock creation, verification and stubbing.

Creates simulated objects that mimic the behavior of real objects in controlled ways.

Allows specification of which, and in what order, methods will be invoked on a mock object and what parameters will be passed to them, as well as what values will be returned.

```
// Need data to mock Product Repository getAll
ListBuilder listBuilder = new ListBuilder();
// Declare mock Product Repository getAll
when(productRepositoryMock.getAll()).thenReturn(listBuilder.getProductList());
 // Invoke getAll
         List<Product> products = productService.getAll();
         // Validate results ….
```

with HAMCREST…

Hamcrest

“Matchers that can be combined to create flexible expressions of intent”

Framework for 'match' rules to be defined declaratively.

Strives to make your tests as readable as possible.

Integrates with Junit ; TestNG…and Mock frameworks

“Third Generation” Matcher framework

JUnit's ***assertEquals*** replaced by Hamcrest ***assertThat***

The product list “hasItem” that matches with “allOf” the list properties

```
assertThat(products, hasItem(
           allOf(
                 hasProperty("id", is(1L)),
                 hasProperty("category", hasProperty("name", is("Sports"))),
                 hasProperty("description", is("Two wheels")),
                 hasProperty("name", is("Bicycle"))
                )
          ));
```

Hamcrest

Organizing Unit Test Data: Test Data Builder

Based on Builder Pattern

[Solution to the telescoping constructor anti-pattern]

Reduces the number of constructors, by processing initialization parameters step by step

```
Product product = new ProductBuilder()
                    .withId(1L)
                    .withName("Bicycle")
                    .withDescription("2 Wheels")
                    .build();
```

Telescoping Constructors

Difficult to identify parameters of the same type
Often there isn't a constructor that fits your need
So, either add a new constructor or use a null parameter.

```
public Product(Long id, String name) {
        this(id,name,””, 0.0,null);
    }

public Product(Long id, String name, String Description) {
        this(id,name,description, 0.0,null);
    }

public Product(Long id, String name, String Description,Float price) {
        this(id,name,description, price,null);
    }

 public Product(Long id, String name, String Description,Float price,Category category) {
        this.setId(id);
        this.setName(name);

            …
    }
```

ProductService is under Test

```
public class ProductServiceImpl implements ProductService {

@Autowired
ProductRepository productRepository;

   public List<Product> getAll() {
        return productRepository.getAll();
}

public void save(Product product) {
   productRepository.save(product);
   return ;
}

   public Product findOne(Long id) {
        return productRepository.findOne( id);
}
```

This is the behavior to Mock!
See ProductTestSA/ProductServiceTest Demo

ProductRepository Interface

```
public interface ProductRepository   {

  // Returns a List of all Products
public List<Product> getAll();
// Save a Product
public void save(Product product);
// Find a Product by id
 public Product findOne(Long id);


}
```

Here is our Interface
We want to Mock its behavior

Spring Framework Testing Support

The adoption of the test-driven-development (TDD) approach to software development is advocated by the Spring team:

By application of the IoC principle to unit testing

& support for integration testing

Built-in Mock Libraries:

org.springframework.mock.env
org.springframework.mock.jndi
org.springframework.mock.web
org.springframework.mock.web.portlet

Spring Test Reference

Spring MVC Testing Support

First class support for testing Spring MVC code

Built on top of the Servlet API mock objects:

org.springframework.mock.web

comprehensive set of Servlet API mock objects, which are useful for testing web contexts, controllers, and filters.

THEREFORE: does *not* use a running Servlet container.

Support for :

Standalone mode - instantiate and test individual controllers

Application Context mode– test in full configuration environment

Spring MVC Unit Testing

Stand Alone Mode – Similar to a unit test. It tests one controller at a time, the controller can be injected with mock dependencies manually.

BOTH modes are "officially" integration tests

Since they use the Dispatcher Servlet…

Application Context Mode - loads the actual Spring MVC configuration resulting in a more complete integration test.

Stand Alone Mode

On a scale from classic unit test[1] to full integration test[10]

3 [IMHO]

On the other hand unit testing without the Dispatcher Servlet is somewhat meaningless

Stand Alone Mode - Allows for "Unit Testing":

A range of Controller Specific Annotations & Functionality:

Request mapping, data binding, type conversion, validation…

Testing @InitBinder, @ModelAttribute, and @ExceptionHandler methods

View “Unit Testing”

Selenium

Automates web application – browser interactions for testing purposes

Selenium WebDriver

Browser-based regression automation suites and tests

Major browser support- Chrome [**ChromeDriver**],FireFox[**FireFoxDriver**], etc.

Language specific bindings [Java,C#,PHP…]

Selenium IDE

record-and-playback of interactions with the browser

HTMLUnit

Headless browser

Models HTML documents and provides an API that allows you to invoke pages, fill out forms, click links, etc... just like you do in your "normal" browser.

Why WebDriver & HtmlUnit ?

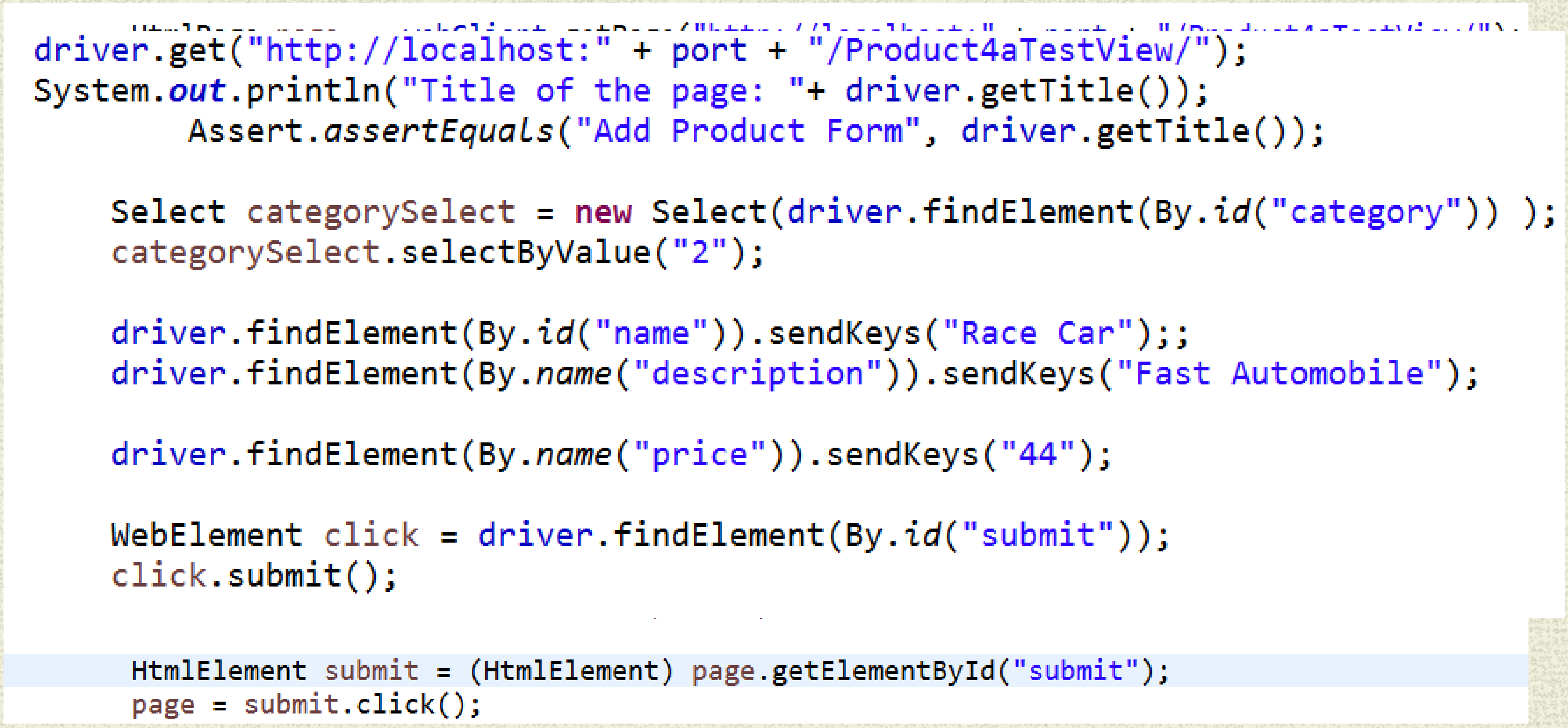
We can integrate Selenium with HTMLUnit [**HTMLUnitDriver**]

Selenium with HTMLUnit

- Selenium WebDriver provides a very elegant API and allows us to easily organize our code.

Selenium WebDriver

HtmlUnit Stand Alone



```
HtmlPage page = webClient.getPage("http://localhost:" + port + "/Product4aTestView/");
```

```
driver.get("http://localhost:" + port + "/Product4aTestView/");
System.out.println("Title of the page: "+ driver.getTitle());
        Assert.assertEquals("Add Product Form", driver.getTitle());

    Select categorySelect = new Select(driver.findElement(By.id("category")) );
    categorySelect.selectByValue("2");

    driver.findElement(By.id("name")).sendKeys("Race Car");;
    driver.findElement(By.name("description")).sendKeys("Fast Automobile");

    driver.findElement(By.name("price")).sendKeys("44");

    WebElement click = driver.findElement(By.id("submit"));
    click.submit();
```

```
     HtmlElement submit = (HtmlElement) page.getElementById("submit");
     page = submit.click();
```

JSP .versus. Thymeleaf View Testing

JSP

Requires Servlet container so we cannot MOCK it.

It becomes a more “heavyweight” integration test…

Furthermore

To start Tomcat, we must use Spring Boot [embedded Tomcat]

Otherwise manually configuring Tomcat becomes complex

```
@SpringBootTest( ...,webEnvironment=WebEnvironment.RANDOM_PORT)
driver.get("http://localhost:" + port + "/ProductTestView/");
```

Thymeleaf

- Template engine has no servlet constraints

It allows for Mocking – more of a “Unit Test”

```
@SpringBootTest( ...,webEnvironment=WebEnvironment.MOCK)
driver.get("http://localhost/ProductTestView/");
```

*********** If URL host == “localhost” then MOCK ***********

Even Better Test Organization
Page Object Pattern

- Design pattern to create a “Page” class for each web page
- **Problem:** Many tests use same page WebElements
- If a WebElement changes ALL tests also change
- **Solution:**
- Page class contains all the WebElements of the web page
- Page class contains methods to operate on the WebElements.
-

Changes are in only one place

UI changes do not require changes to tests

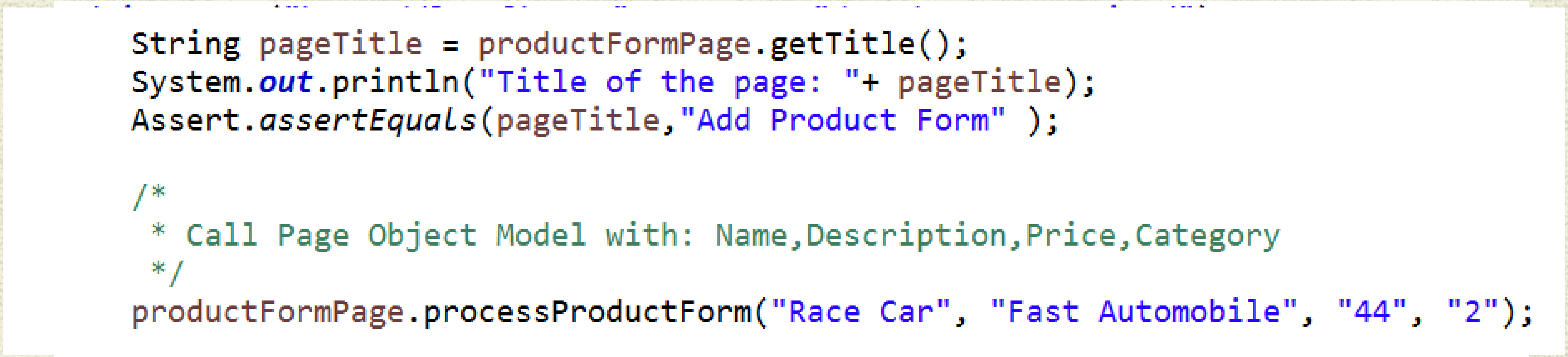
Only need to change the page objects

- **See ProductTestViewThyme/ProductFormSeleniumTestPOM.java**

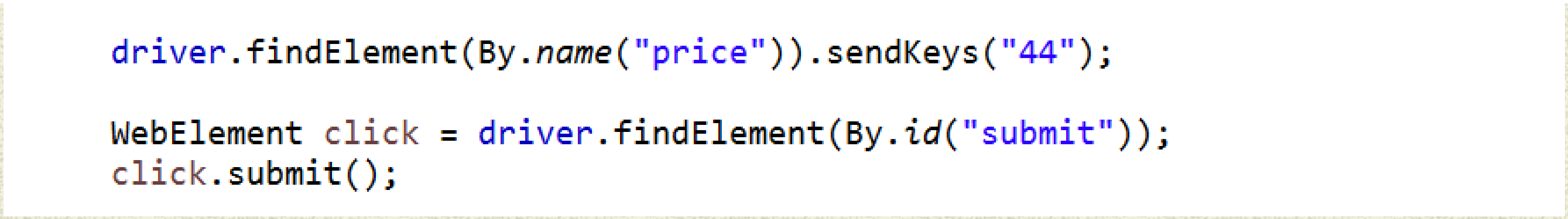
Selenium With Page Object Pattern

- Page Object enhances test maintenance and reduces code duplication.

SelenPatternHTMLObjectWebModelDriver

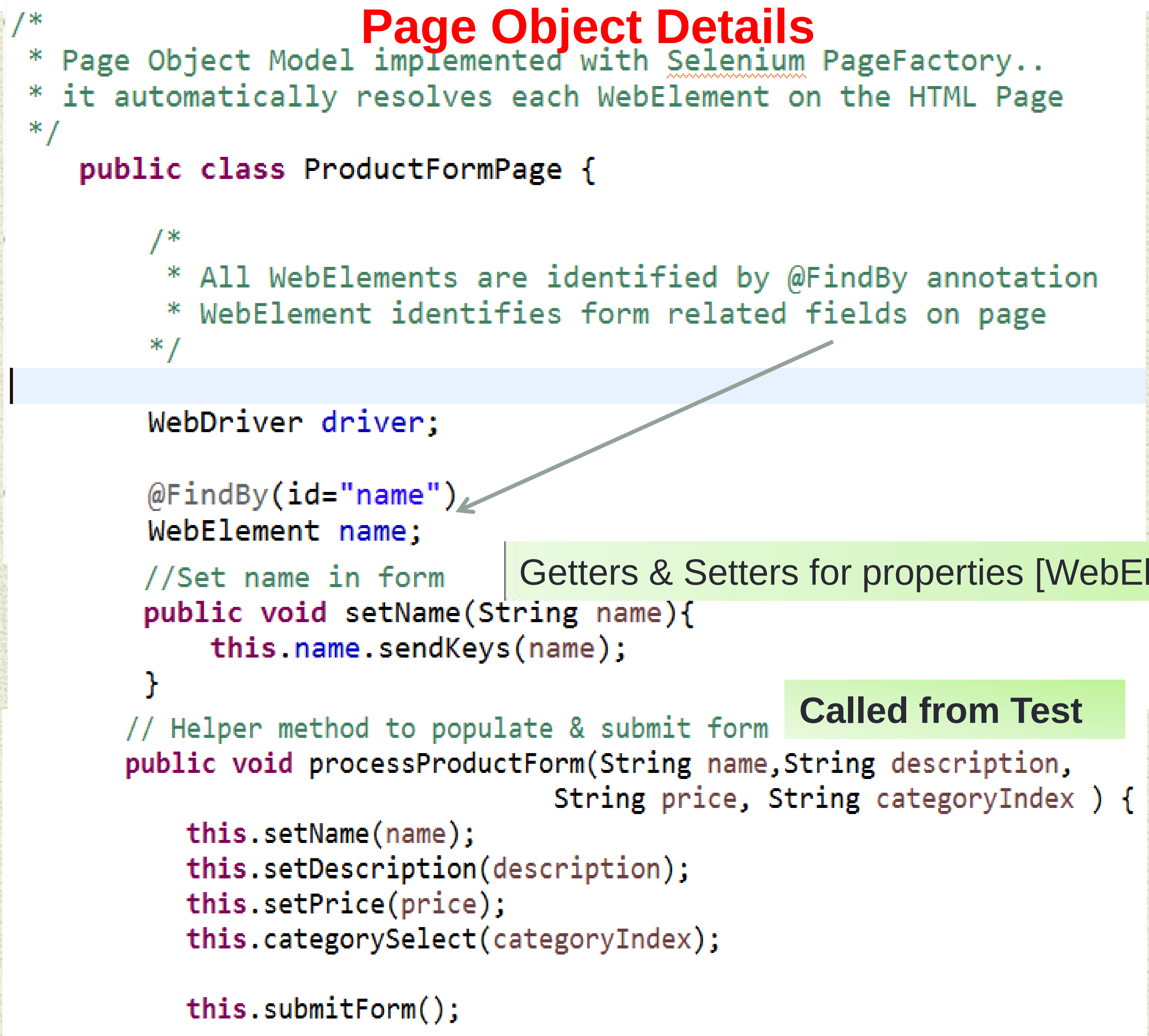


```
    String pageTitle = productFormPage.getTitle();
    System.out.println("Title of the page: "+ pageTitle);
    Assert.assertEquals(pageTitle,"Add Product Form" );

    /*
     * Call Page Object Model with: Name,Description,Price,Category
     */
    productFormPage.processProductForm("Race Car", "Fast Automobile", "44", "2");
```



```
      driver.findElement(By.name("price")).sendKeys("44");

      WebElement click = driver.findElement(By.id("submit"));
      click.submit();
```

Page Object Details



```
/*
 * Page Object Model implemented with Selenium PageFactory..
 * it automatically resolves each WebElement on the HTML Page
 */
    public class ProductFormPage {

        /*
         * All WebElements are identified by @FindBy annotation
         * WebElement identifies form related fields on page
        */

        WebDriver driver;

        @FindBy(id="name")
        WebElement name;

        //Set name in form
        public void setName(String name){
            this.name.sendKeys(name);
        }
       // Helper method to populate & submit form
       public void processProductForm(String name,String description,
                                      String price, String categoryIndex ) {
           this.setName(name);
           this.setDescription(description);
           this.setPrice(price);
           this.categorySelect(categoryIndex);

           this.submitForm();
```

Getters & Setters for properties [WebElelments]

Called from Test

Main Point

Testing is a fundamental and integral aspect of any and every software development effort. It ensures error free business applications.

Likewise, life in accordance with the structuring Laws of Nature make every action problem free and successful.