Test Automation Lecture 15 –

Data-Driven Testing & Behavior-Driven Development

IT Learning &
Outsourcing Center

Lector: Milen Strahinski
Skype: strahinski
E-mail: milen.strahinski@pragmatic.bg
Facebook: http://www.facebook.com/LamerMan
LinkedIn: http://www.linkedin.com/pub/milen-strahinski/a/553/615

www.pragmatic.bg

Summary-overall

- Getting familiar with Data-Driven Testing
- Creating a data-driven test using JUnit
- Creating a data-driven test using TestNG
- Reading test data from a CSV file using JUnit
- Reading test data from Database using JUnit
- Reading test data from an Excel file using JUnit and Apache POI
- Using Cucumber-JVM and Selenium WebDriver in Java for BDD

Data-Driven – Introduction (part 1)

- Testing can be very repetitive, not only because we must run the same test over and over again, but also because many of the tests are only slightly different. For example, we might want to run the same test with different test inputs or test conditions and verify that the actual output varies accordingly. Each of these tests would consist of the exact same steps, however, what differs is the test data.

- We can use the data-driven approach to achieve this. The data-driven testing approach is a widely used methodology in software test automation.

Data-Driven Introduction (part 2)

- We will use the BMI calculator application as an example to understand the data-driven testing approach. The Body Mass Index (BMI) is a measurement of body fat based on height and weight, which applies to both men and women. BMI can be used to indicate whether you are overweight, obese, underweight, or normal. The following tables show the various BMI categories:

Category	BMI range
Underweight	Less than 18.5
Normal weight	18.5 to 24.9
Overweight	25 to 29.9
Obesity	Greater than or equal to 30

Data-Driven Introduction (part 3)

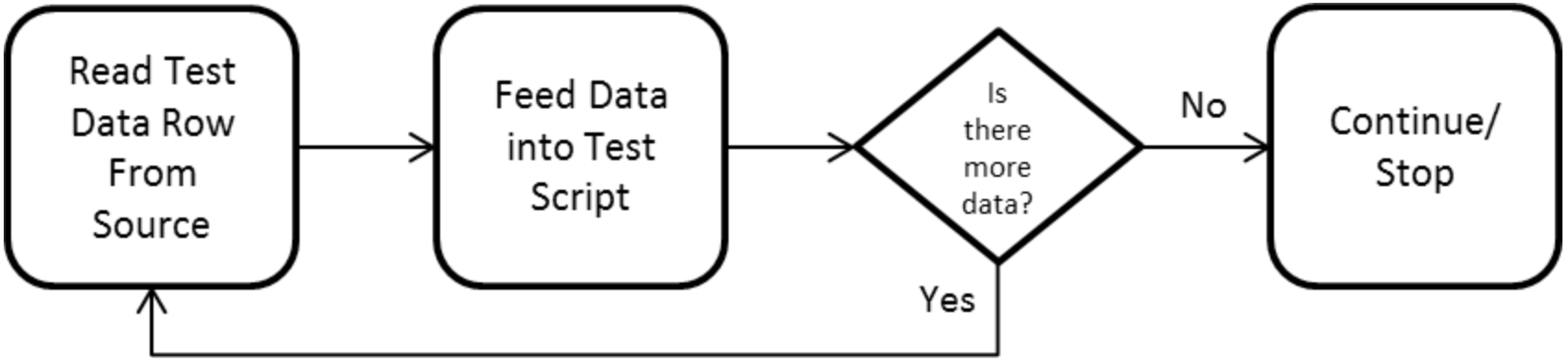
- To test whether the BMI calculator application indicates BMI categories correctly, instead of having a separate test script for each category, we can have one script that will enter the height and weight by referring to a set of values and checking the expected values. We can use the following combinations of test conditions to test the BMI calculator application:

Height(centimeters)	Weight(kilograms)	BMI	Category
160	45	17.6	Underweight
168	70	24.8	Normal
181	89	27.2	Overweight
178	100	31.6	Obesity

- In the simplest form, the tester supplies inputs from a row in the table and expected outputs, which occur in the same row.

Data-driven approach – workflow

- In the data-driven-approach, we can maintain the test data in form tables in a variety of formats, such as CSV files, Excel spreadsheets, and databases.
- We implement test scripts after reading input and output values from data files, row by row, then passing the values to the main test code. Then, the test code navigates through the application, executing the steps needed for the test case using the variables loaded with data values.
- Data-driven tests are great for applications involving calculations for testing ranges of values, boundary values, and corner cases.



Benefits of data-driven testing (part 1)

- The benefits of data-driven testing are as follows:
  - With data-driven tests, we can get greater test coverage while minimizing the amount of test code we need to write and maintain.
  - Data-driven testing makes creating and running a lot of test conditions very easy.
  - Data-driven tests require no programming skills. Even testers with limited knowledge of automation can quickly create tables on their own.
  - Test data can be designed and created before the application is ready for testing.
  - Data tables can also be used in manual testing.

Benefits of data-driven testing (part 2)

- Overall, the data-driven approach is the key for ease of use while building large-scale test automation.

- Selenium WebDriver, being a pure browser automation API, does not provide built-in features to support data-driven testing. However, we can add support for data-driven testing using various options in Selenium WebDriver. In this lecture we will create some basic data-driven tests in JUnit and TestNG.

Creating a data-driven test using JUnit (part 1)

- We can create data-driven Selenium WebDriver tests using the JUnit 4 parameterization feature. This can be done by using the JUnit parameterized class runner.

- In this lecture, we will create a simple JUnit test case to test our BMI calculator application. We will specify the test data within our JUnit test case class. We will use various JUnit annotations to create a data-driven test.

Creating a data-driven test using JUnit (part 2) – RULES!

JUnit 4 has introduced a new feature Parameterized tests. Parameterized tests allow developer to run the same test over and over again using different values. There are five steps, that you need to follow to create Parameterized tests.

- Annotate test class with @RunWith(Parameterized.class)
- Create a public static method annotated with @Parameters that returns a Collection of Objects (as Array) as test data set.
- Create a public constructor that takes in what is equivalent to one "row" of test data.
- Create an instance variable for each "column" of test data.
- Create your tests case(s) using the instance variables as the source of the test data.

Creating a data-driven test using JUnit (part 3)

- Lets explore the SimpleDDT.java class in the code examples.

Creating a data-driven test using TestNG (part 1)

- TestNG is another widely used testing framework with Selenium WebDriver. It is very similar to JUnit. TestNG has rich features for testing, such as parameterization, parallel test execution, and so on.

- TestNG provides the @DataProvider feature to create data-driven tests. In this recipe, we will use the @DataProvider feature to create a simple test. Creating data-driven tests in TestNG is fairly easy, when compared with JUnit.

Creating a data-driven test using TestNG (part 2)

- Unlike the JUnit, where parameterization is done on a class level, TestNG supports parameterization at test level.

- In TestNG, we do not need a constructor and instance variables for the test case class to pass the parameter values. TestNG does the mapping automatically.

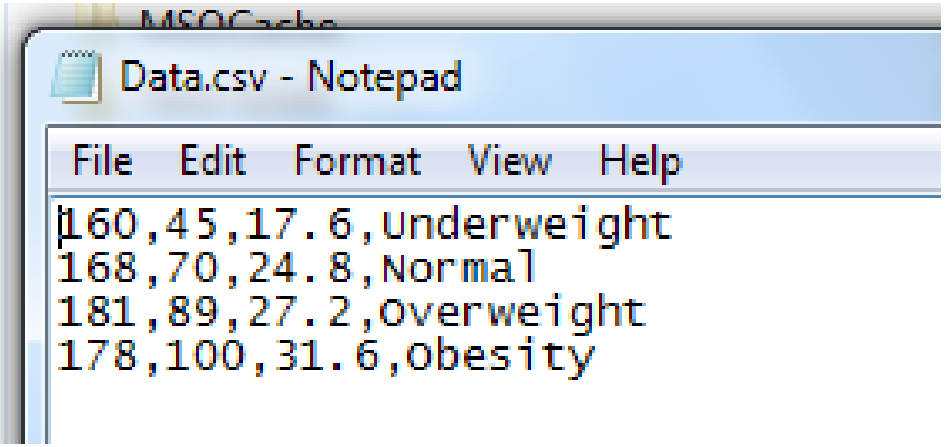
Creating a data-driven test using TestNG (part 3)

- Lets explore the TestNGDDT.java class in the code examples.

Reading test data from a CSV file using JUnit (part 1)

- We saw a simple data-driven test using JUnit and TestNG. The test data was hardcoded in test script code. This could become difficult to maintain. It is recommended that we store the test data separately from the test scripts.

- Often we use data from the production environment for testing. This data can be exported in CSV format. We can read CSV files using Java IO and utility classes and can pass the data from these files to the test code.



Reading test data from a CSV file using JUnit (part 2)

- Lets explore the CsvTestData.java class in the code examples.

- Changing delimiters - Sometimes, CSV files may have a separate delimiter than the comma. We can change the getTestData() method to handle these delimiters. For example, a tab character is used to separate records. For this, we can split the record, using the \t character, in the following way:

```
String fields[] = record.split("\t");
```

Reading test data from an Excel file using JUnit and Apache POI (part 1)

- To maintain test cases and test data, Microsoft Excel is the favorite tool used by testers. Compared to the CSV file format, Excel gives numerous features and a structured way to store data. A tester can create and maintain tables of test data in an Excel spreadsheet easily.

	A	B	C	D	E
1	Height	Weight	Bmi	Category	ErrorMessage
2	160	45	17.6	Underweight	<Blank>
3	168	70	24.8	Normal	<Blank>
4	181	89	27.2	Overweight	<Blank>
5	178	100	31.6	Obesity	<Blank>
6	<Blank>	80	<Blank>	<Blank>	Please enter Height
7	181	<Blank>	<Blank>	<Blank>	Please enter Weight

- In this recipe, we will use an Excel spreadsheet as your data source. We will use the Apache POI API, developed by the Apache Foundation, to manipulate the Excel spreadsheet. This recipe also implements some negative test handling.

Reading test data from an Excel file using JUnit and Apache POI (part 2)

- Set up a new project and add JUnit4 to project's build path
- Download and add the following Apache POI JAR files to the project's build path, from http://poi.apache.org/:
  - poi-3.9.jar
  - poi-ooxml-3.9.jar
  - poi-ooxml-schemas-3.9.jar
  - dom4j-1.6.1.jar
  - stax-api-1.0.1.jar
  - xmlbeans-2.3.0.jar
- Create an Excel spreadsheet with the required data
- We will also need a SpreadsheetData helper class to read Excel spreadsheets. This will be available in the code examples. This class supports both the old .xls and newer .xlsx formats.

Reading test data from an Excel file using JUnit and Apache POI (part 3)

- Lets explore the ExcelTestData.java class and its helper SpreadsheetData.java in the code examples.

JDBC test data preconditions for Microsoft Access

- 2007 Office System Driver: Data Connectivity Components
http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=23734

- JDK 7 is required for the JDBC-ODBC Bridge, otherwise the official one should be used which can be found somewhere under Microsoft's website.

- 32 bit JDK
http://www.oracle.com/technetwork/java/javase/downloads/index.html?ssSourceSiteId=otnjp

- Eclipse -> Window -> Preferences -> Installed JRE -> Remove all other JREs and add the just installed 32 bit JDK

Reading test data from a database using JUnit and JDBC (part 1)

- In the earlier recipes, we used CSV and Excel spreadsheets to maintain the test data and read this test data in JUnit.

- The test data can also be read from a database. This works similar to the previous recipes, however, we will create a helper method using JDBC to read the test data from a Microsoft Access database. This recipe can be used with any database.

Reading test data from a database using JUnit and JDBC (part 2)

- Lets explore the DbTestData.java classin the code examples.

Behavior Driven Development (BDD) – Introduction (part 1)

- Behavior-driven Development (BDD) is an agile software development practice that enhances the paradigm of Test Driven Development (TDD) and acceptance tests, and encourages the collaboration between developers, quality assurance, domain experts, and stakeholders. Behavior-driven Development was introduced by Dan North in the year 2003 in his seminal article available at http://dannorth.net/introducing-bdd/.

- Behavior-driven Development focuses on obtaining a clear understanding of desired application behavior through discussion with stakeholders using an ubiquitous language as described at http://behaviour-driven.org/.

- It extends TDD by writing test cases in a natural language that non-programmers can read. Users describe features and scenarios to test these features in plain text files using Gherkin language in Given, When, and Then structure. You can find out more about Gherkin language at http://en.wikipedia.org/wiki/Behavior-driven_development and https://github.com/cucumber/cucumber/wiki/Gherkin.

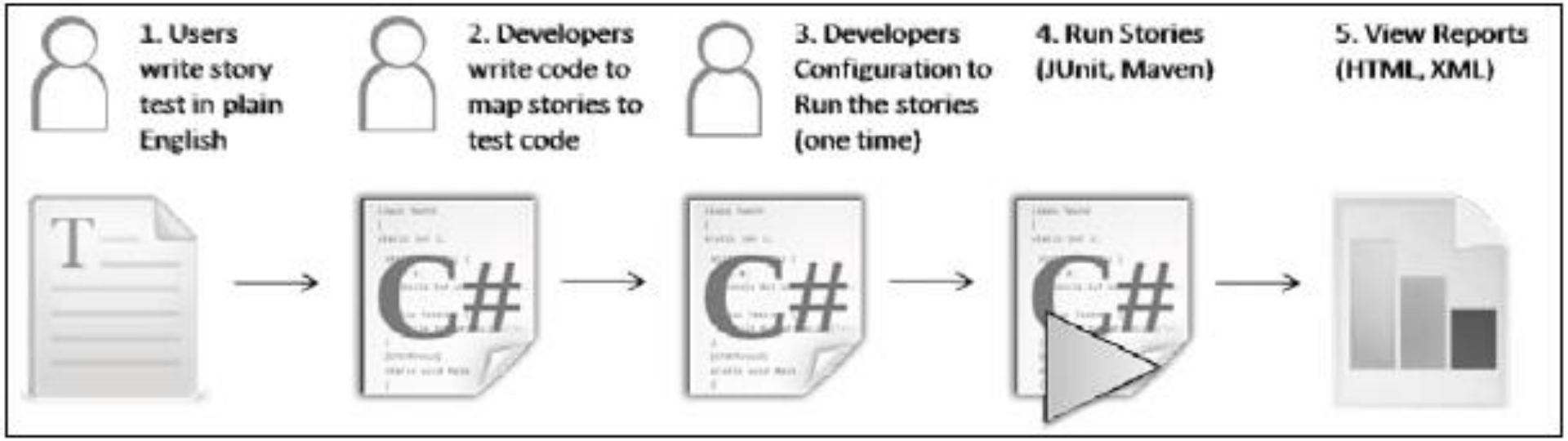
Behavior Driven Development (BDD) – Introduction (part 2)

- The Given, When, and Then structures in the Gherkin language are described as follows:
  - *Given*: This represents the initial context (precondition)
  - *When*: This represents the user performing a key action (actor + action)
  - *Then*: This ensures some kind of outcome (observable result)

- Behavior-driven Development is also known as **Acceptance Test Driven Development (ATDD)** or **Story Testing**.

Behavior Driven Development (BDD) – Introduction (part 3)

- In Behavior-driven Development, the process starts with users of the system and development team discussing features, user stories, and scenarios. These are documented in feature or story files using the Gherkin language. Developers then use the red-green-refactor cycle to run these features using the BDD framework, then write step definition files mapping the steps from scenarios to the automation code and re-running until all the acceptance criteria are met:



Using Cucumber-JVM and Selenium WebDriver in Java for BDD (part 1)

- BDD/ATDD is becoming widely accepted practice in agile software development, and Cucumber-JVM is a mainstream tool used to implement this practice in Java. Cucumber-JVM is based on Cucumber framework, widely used in Ruby on Rails world.

- Cucumber-JVM allows developers, QA, and non-technical or business participants to write features and scenarios in a plain text file using Gherkin language with minimal restrictions about grammar in a typical Given, When, and Then structure.

- This feature file is then supported by a step definition file, which implements automated steps to execute the scenarios written in a feature file. Apart from testing APIs with Cucumber-JVM, we can also test UI level tests by combining Selenium WebDriver.

- In this recipe, we will use Cucumber-JVM and Selenium WebDriver for implementing tests for the fund transfer feature from an online banking application.

Using Cucumber-JVM and Selenium WebDriver in Java for BDD (part 2)

- We will use Maven to build our dependencies and learn how this is done
- We will also explore in details the BDDFundTransfer-Cucumber-JVM.rar in the code examples

Questions?