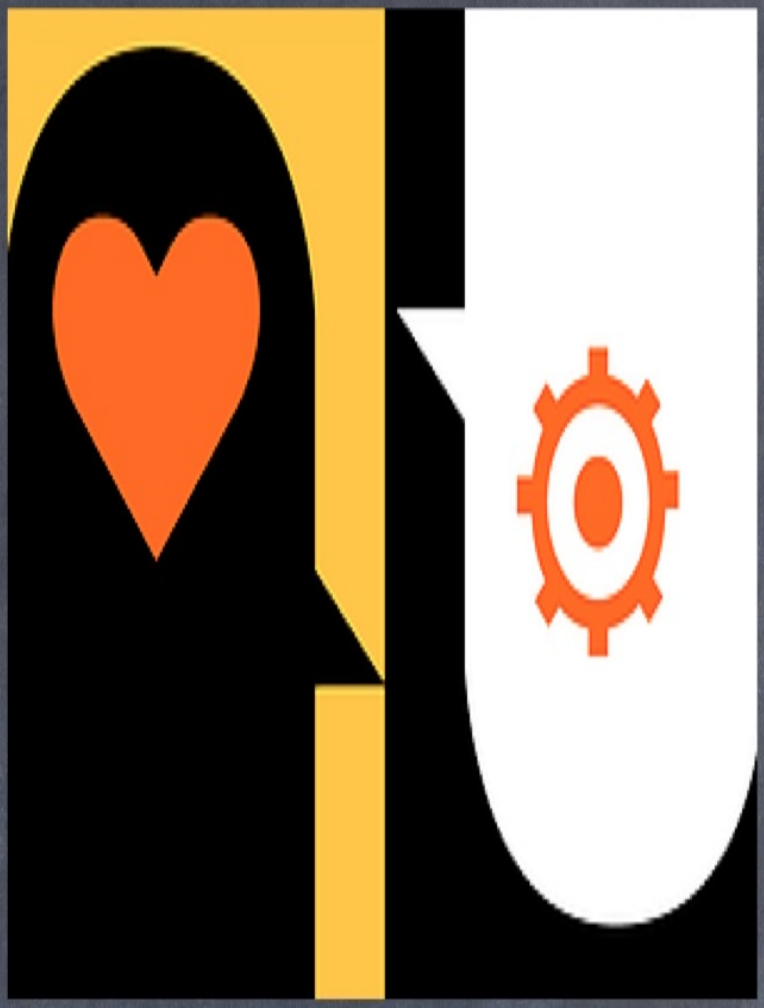


Building Test Framework using Cucumber-JVM

Table of Contents

Cucumber-jvm

Cucumber-jvm is an open source BDD tool that lets user express the behavior of system under test in plain English. It is a great way to have an executable documentation of your system.

Why This Book?

The intent of this book is to demonstrate building a functional test framework using cucumber-jvm. It will help you learn and implement concepts from scratch, at the same time evolving a scalable solution.

What you will learn from this book?

Expect to have a very good grasp of the following once you go through this book

- Basics of behavior driven development.
- Evolving a test framework
- Understanding test runners
- Cucumber feature files and glue code
- Background and hooks in cucumber

For the purpose of the example framework we are using an in-house application (a basic accounting app).

Want us to help developing a custom BDD framework for you?

At TestVagrant, we help Startups with building robust test automation solution be it Mobile, Web or Services. The core idea behind TestVagrant is to bring an

- affordable,
- niche
- rapid solution to startups.

Please visit TestVagrant website to request for free POC.

Happy Reading !!!

BDD - The Basics

Welcome on the learning journey. We will start with a basic know how of BDD. The following articles will talk about the origin of BDD and why are people across the spectrum (business or IT) are so excited about it.

The ideas talked about in this book are from several project experiences that i have had. The target of this book is to get you comfortable with building a test framework from scratch.

Let me know if you were able to follow the articles and achieve the goal. And definitely, let me know in case you were not. That will drive the next versions of this book.

What?

BDD stands for **Behavior Driven Development**. It is an agile software development practice that has evolved from TDD (Test Driven Develolment), and is a brainchild of Dan North, an agile luminary.

The premise of BDD is to bring technology and business togther. As the name suggests, the practice recommends defining the behavior upfront and using the same to drive the product development.

For an example, think of writing a simple addition program. If you were to practice BDD, the team should get together to define the expected behavior such as:

- should perform addition of two positive number
- should perform addition of a positive and a negative number
- should perform addition of two negative number
- should perform addition for decimal numbers upto 2 digits
- and likewise

This brings a lot of clarity into what the team has to develop and where are the boundaries.

Why?

As discussed in the previous article, it has evolved from TDD. So there was something amiss in TDD that led to this evolution.

For those who don't know about TDD, it is a development practice that is more technology focussed in nature. It states that.

- As a developer you should always write a test first for the unit you are going to code for. This would obviously fail in the begining as there is no implementation
- Then implement the unit, as in write the code
- And then verify that the implememtation get the test to pass.

Consider the same addition example, if we were to follow TDD,

- we would have written a test for the addition method first, say a test method that checks the result of adding two positive numbers.
- Only then you write the implementation of the addition method so that the tests pass.

But TDD gives no clarity of how much to test.i.e. should we test for negative numbers, should we test for decimal etc. And hence there is a clear chance of missing an expected behavior of the method while implementing it.

It is not driven through a specification or the so called desired behavior. And because of this, the boundaries are not clear.

This very reason set up the premise for BDD. It advocates that the business value is specified upfornt in terms of the desired behavior of the unit, and that is what gets implemented.

When the team sits together to define the expected behavior, it talks beyond just the need for an addition method. The team agrees on all the behavior that the business expects from the addition method.

So the most important reason on why BDD is a better practice is, in it's core it brings business and technology together. The team now knows how much to test in one go and where are the boundaries.

How?

There are two parts to BDD adoption for teams.

- Mindset
- Tooling support

The team definitely needs to buy-in to the practices and benefits of this development methodology.

And then there are tools that help you facilitate BDD adoption. These tools basically propose a structure for writing specifications that brings in the needed clarity into business requirements.

A very commonly used structure is that of the **user story** in agile teams.

The user story requires you to define every piece of requirement in terms of :

- who is the primary stakeholder for a particular requirement
- what effect does the user wants from this requirement
- what business value would the user get if the aforesaid requirement is acheived.

Consider the addition example to understand the above.

```
As a user of the calculator program
I want to have an addition method supported in the program
So that I can add two numbers
```

Further, BDD expects the desired behavior to be written in a Given, When, Then format.

- **Given** describes the pre-requisites for the expectation
- **When** describes the actual steps to carry on the effect of the expectation
- **Then** describes the verification that the expectations were met.

e.g. Consider the following expectations

Should allow addition of two positive numbers

```
Given the calculator has an addition program
When i choose to add 2 numbers
And i add 2 and 2
Then the result is 4
```

Now there are testing tools that help to automate and execute scenarios like above. So these specifications can actually serve as living documentation. If a requirement changes, the test has to change for it to execute successfully.

This book is going to talk about one such tool, Cucumber and the whole test framework ecosystem around it.

Getting Started

Articles in this chapter will help you understand the whole ecosystem around any test framework plus help you set up to get started on the journey.

Understanding the Ecosystem

Let's start with understanding all the different parts of any test framework.

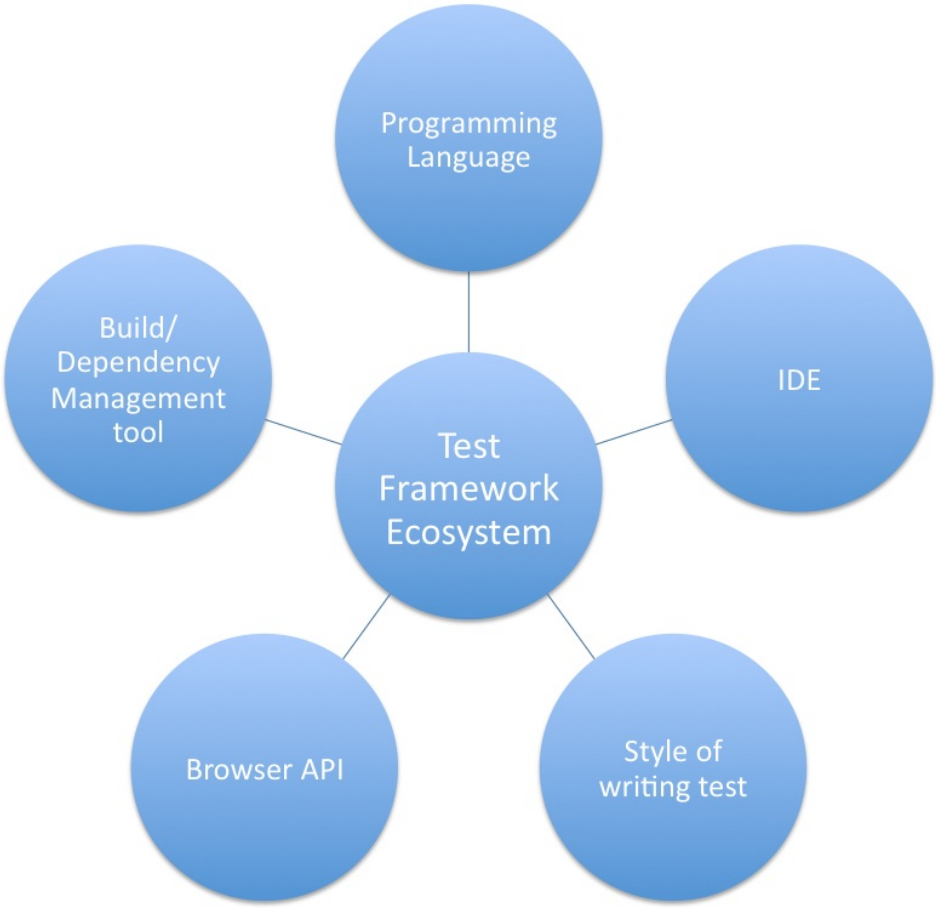
The most important thing is to choose a **programming language** that we will be using for writing for the test code.

The next thing you need is an **IDE** (integrated development environment). This is where we will write all our test code in the chosen language.

Another thing that we will be using with the IDE is a **cucumber-jvm plugin**. This will add rich features for syntax highlighting, autocompletion and navigation in the plain text based feature file.

And then we have a **browesr driving tool**. Remember cucumber-jvm only gives you a way to write test scenarios, it doesn't give you api's for browser interaction.

And a **build + dependency management tool** that would help us manage dependent jars that are needed for the test project.



Programming Language

The first and the foremost important thing is to choose a programming language for writing the tests.

I would recommend to always go with the same stack that is being used for development of the application. i.e. use JAVA for writing tests if application is JAVA based and likewise.

For the purpose of this book we will be using JAVA as the chosen language to write all our test code.

IDE

IDE stands for integrated development environment. As the name suggests, this helps in getting some sort of a structure to our project.

Also, the IDE generally has plug-ins for tools that we would want to use as part of our project. Like a plug-in for cucumber-jvm in our case.

For the purpose of this book, we have chosen to use the community edition of IntelliJ. The other possible options is Eclipse.

People comfortable in eclipse should be able to use the same and readily apply the concepts dealt with in the book henceforth.

Style of Writing Tests

This is one of the most important aspect to decide when choosing a test framework stack.

This is how you express yourself in the test scenario. This is what everyone reads to understands what getting tested and how the system behaves.

Since the core of this book is a BDD tool, we will be using the BDD style of writing test scenario. This emphasizes on a Given, When, Then format, where

- **Given** defines the pre-requisite of any test scenario
- **When** is used to list down all the necessary steps needed for the scenario workflow
- **Then** takes care of asserting whether the actual output of the workflow matches with our expectation.

We will plenty of such examples further in this book.

The other possible options in this space are the normal unit testing style of writing tests using a JUnit or TestNG test frameworks.

Browser API's

This is the core of the whole UI based test framework. This is what takes care of doing all the real action using the various browser components.

You need the api's to click a button, enter text into a text box, select from a list of options and several other browser level actions.

For the purpose of this book, we have chosen to use Selenium.

Build and Dependency Management Tool

This is another very important piece. While choosing any test stack to write our tests, we are actually talking about multiple tools to be used. Like in our case, we will be using

- Selenium for browser api's
- Cucumber-jvm for writing test scenarios
- JUnit for doing assertions etc

So basically, our framework is dependent on a bunch of smaller projects or jar's in our case. This is where a **dependency management tool** comes into picture.

We could have chosen to download and add all the required jars into some lib folder but that can easily get confusing. This is where a tool like **maven** can be of great help. It can pull all the dependent jars needed for our project on it's own and we wouldn't need to download and add them separately.

Also, Maven doubles up as a **build tool** and allows you to run tests from command line. This is of great help when we plan to integrate our test to a continuous integration process.

Setup Machine

Now that we understand the ecosystem, lets set up the machine to have all the required components in place.

Download And Install

Download and install the following:

Intellij Community Edition - this is the IDE that we will use to write our feature files and all the code behind.

JAVA - Get the latest jdk on your machine. At the time of writing this book, jdk 1.8 is the latest.

Selenium - Download the selenium-java jar. We will be using selenium api's to drive the browser actions.

Maven - Download the latest maven jar. We will use it for dependency management and at a later point to demonstrate running the test through command line.

Get the Cucumber-jvm Plugin

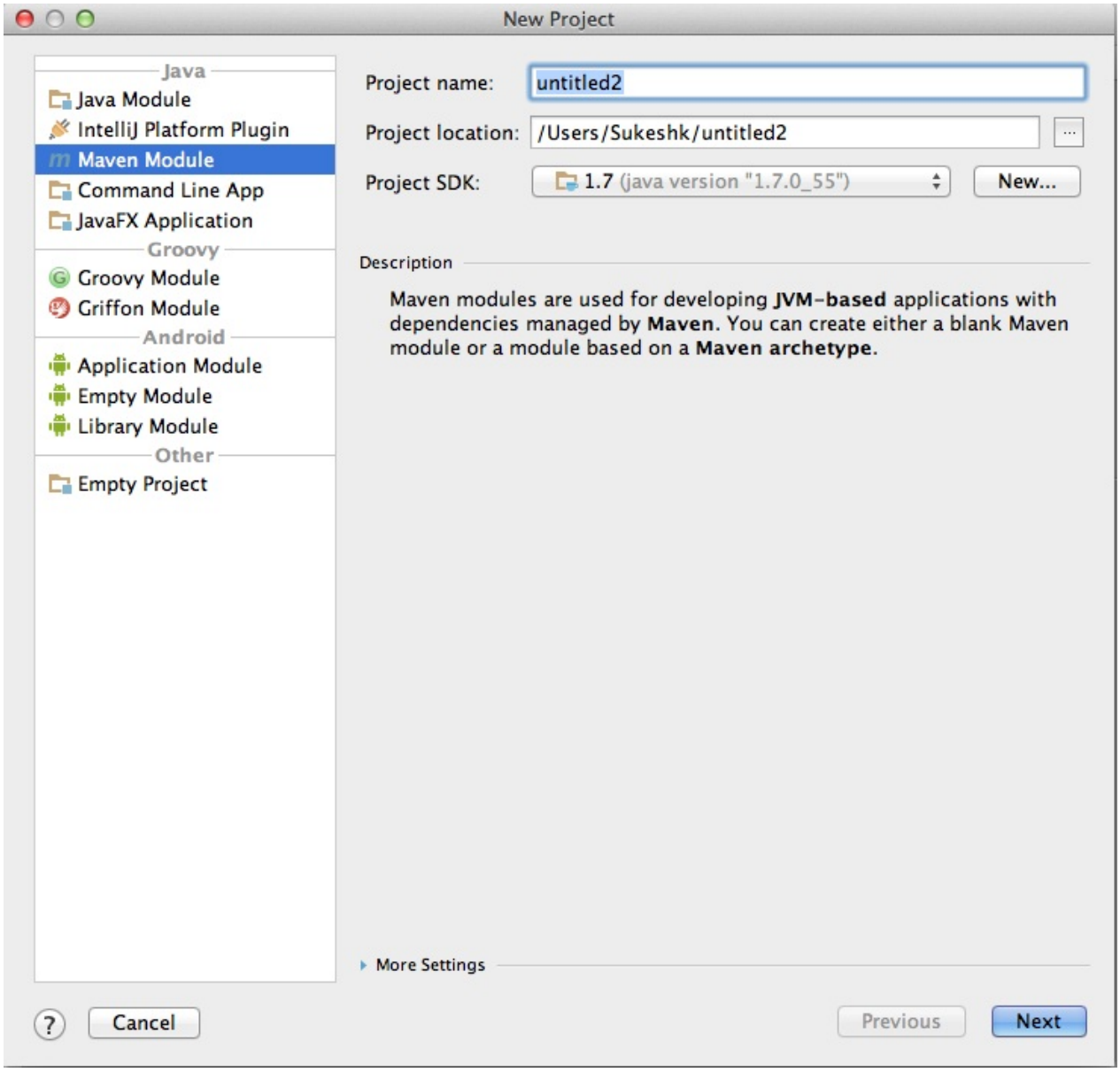
- Open Intellij
- Go to Preferences -> Plugins
- Choose to install Cucumber for JAVA plugin.
- You might need to restart the IDE.

Create The Test Project

So our machine is ready with all the setup required in place. Next we will create the example test project that would be used to demonstrate the concepts.

Create a New Maven Project

Start Intellij Choose to create a new project. Go to File -> New Project. Choose to create a Maven Module.



Enter a project name. Select the java sdk to use. Select the jdk that you have downloaded.

Use the wizard to complete creation of the project. Once created, you will have a projetc created and a pom.xml file for the same.

Add project dependencies

Open the pom.xml file.

Copy the following inside the dependencies tag.

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>1.1.5</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-jvm</artifactId>
    <version>1.1.5</version>
    <type>pom</type>
</dependency>

<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-junit</artifactId>
    <version>1.1.5</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>2.41.0</version>
</dependency>
```

If you notice, we are cucumber-jvm version 1.1.5 and selenium 2.41.0. These are the latest available at the time of writing the book.

Choose to pull the latest jar versions available.

Once you do this, all the required jars for your project would be imported and available for you while writing the test code.

Now the environment is all set to start writing the test code.

Write Your First Test

Now starts the exciting bit. You are all set to rock on. Let's write some code.

Decide On A Test Scenario

Cucumber proposes to write scenario in the Given/When/Then format.

Decide on a example test scneario from the example accounts application that we can try to automate.

Let's go with adding a new client as the test scenario.

Now as mentioned, in BDD terms the scenario would look like the following.

```
    Scenario: Sign up a new user
     Given the user is on landing page
     When she chooses to sign up
     And she provides the first name as Sukesh
     And she provides the last name as Kumar
     And she provides the email as validemail@aq.com
     And she provides the password as password
     And she provides the confirm password again as password
     And she signs-up
     Then she should be logged in to the application
```

Now we will try to convert the above into an executable specification using cucumber-jvm.

Write The First Feature File

When the project was created as a maven module, intellij would have also created a folder for all the source code to be kept named as "src" and src should ideally have two sub-folders - "main" and "test"

Conventionally, all the test code should go to the "test" package. Both the sub-folder would have a java package inside them.

Let's create a new package for all our feature files, inside the test->java package. Name it "features".

Now, choose to add a new file inside the feature package, we can call it NewClientWorkflow.feature.

Add the following into your feature file:

```
Feature:
  As a user
  I want to be able to add new clients in the system
  So that i can add accounting data for that client


Scenario: Sign up a new user
    Given the user is on landing page
    When she chooses to sign up
    And she provides the first name as Sukesh
    And she provides the last name as Kumar
    And she provides the email as validemail@aq.com
    And she provides the password as password
    And she provides the confirm password again as password
    And she signs-up
    Then she should be logged in to the application
```

It should look like the following in your code.

NewClientWorkFlow.feature

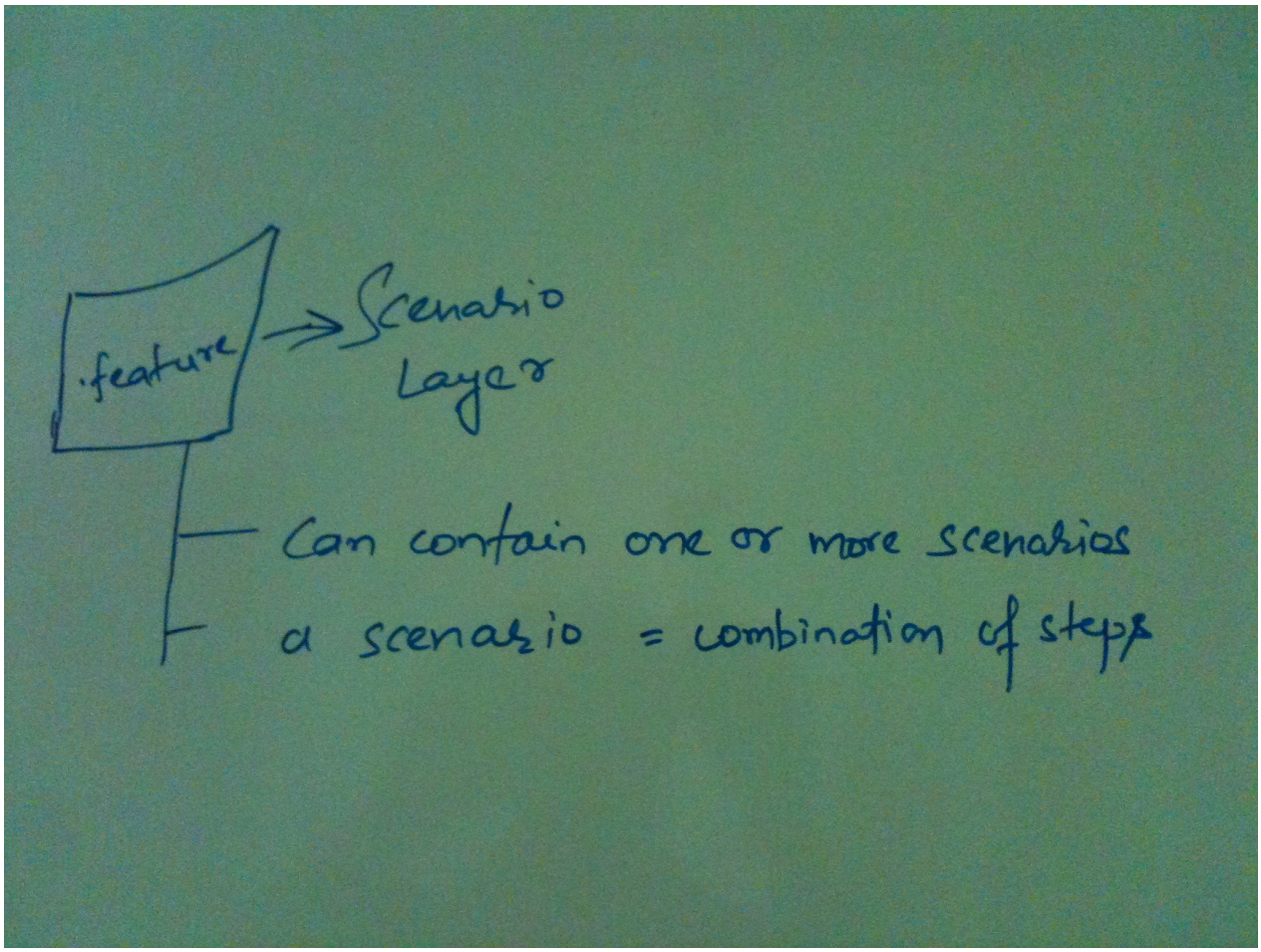
```
Feature:
  As a user
  I want to be able to add new clients in the system
  So that i can add accounting data for that client


  Scenario: Sign up a new user
    Given the user is on landing page
    When she chooses to sign up
    And she provides the first name as Sukesh
    And she provides the last name as Kumar
    And she provides the email as validemail@aq.com
    And she provides the password as password
    And she provides the confirm password again as password
    And she signs-up
    Then she should be logged in to the application
```

Decoding the feature file

Also now is the time to familiarize yourself with some cucumber terminologies. If you see the feature file content, you can notice the following keyword:

- **Feature**: Cucumber mandates to write a brief feature description of the scenarios you want to write in a feature file.
- **Scenario**: Before writing any scenario it also mandates the user to write a brief scenario description.
- **Given/When/Then/And** - Every step in your scenario would be starting with one of them.
- There are **more such keywords** but let's talk about them as they get introduced later in the book.

Now our feature file is ready but there is no code in the background to support these steps, hence the job is only half done. Let's write the glue code for the steps in our scenario.



Writing the Glue Code

Every step in the scneario needs to be matched with a glue code for the same.

Now is the time to create another package inside test->java. Name it "steps". This is where we will keep all the glue code.

Create a new java class inside steps package. Call it "ClientSteps".

Here is the glue code for the scenario we wrote:

public class ExampleSteps {

```
WebDriver driver = new FirefoxDriver();

@Given("^the user is on landing page$")
public void setup() throws Throwable {
    driver.get("http://accountsdemo.herokuapp.com");
    driver.manage().window().maximize();
}


@When("^she chooses to sign up$")
public void she_chooses_to_sign_up() throws Throwable {
    driver.findElement(By.linkText("Sign up")).click();
}


@And("^she provides the first name as ([^\"]*)$")
public void she_provides_the_first_name_as(String firstName) throws Throwable {
     driver.findElement(By.id("user_first_name")).sendKeys(firstName);
}


@And("^she provides the last name as ([^\"]*)$")
public void she_provides_the_last_name_as(String lastName) throws Throwable {
    driver.findElement(By.id("user_last_name")).sendKeys(lastName);
}

@And("^she provides the email as ([^\"]*)$")
public void she_provides_the_email_as(String email) throws Throwable {
    driver.findElement(By.id("user_email")).sendKeys(email);
}

@And("^she provides the password as ([^\"]*)$")
public void she_provides_the_password_as(String password) throws Throwable {
    driver.findElement(By.id("user_password")).sendKeys(password);
}
```

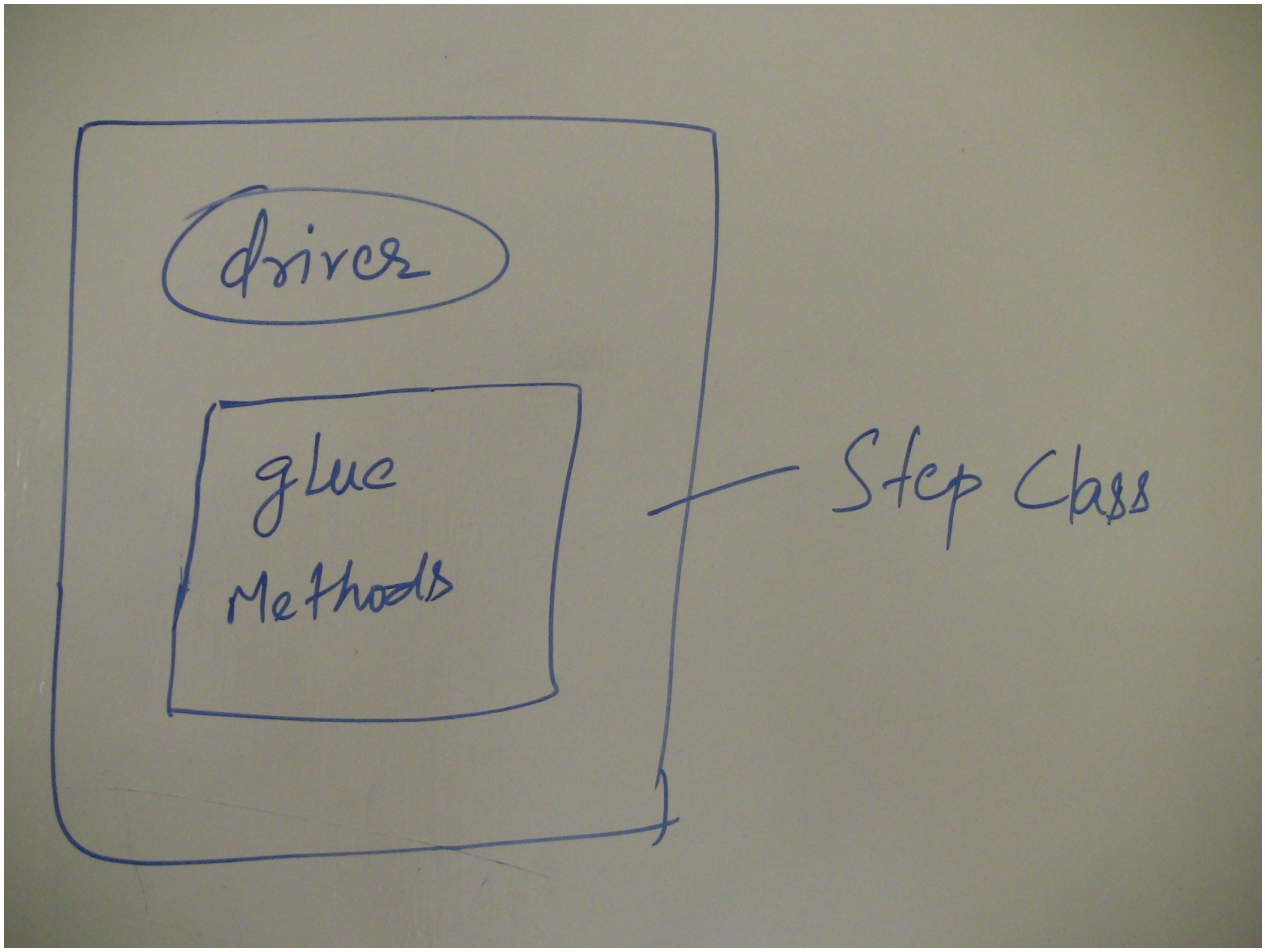
```
    @And("^she provides the confirm password again as ([^\"]*)$")
    public void she_provides_the_confirm_password_again_as(String confirmPassword) throws
    Throwable {
        driver.findElement(By.id("user_password_confirmation")).sendKeys(confirmPassword);
    }


    @And("^she signs-up$")
    public void she_signs_up() throws Throwable {
        driver.findElement(By.name("commit")).click();
    }

    @Then("^she should be logged in to the application$")
    public void she_should_be_logged_in_to_the_application() throws Throwable {
        boolean signOutLinkDisplayed = driver.findElement(By.cssSelector("a[href='/users/s
    ign_out']")).isDisplayed();
        Assert.assertTrue(signOutLinkDisplayed);
    }
```

}

Decoding the step class

Let try to understand the step class code we just wrote. If you look closely, you will observe the following:

- You will find one step method each for every step that we wrote in our test scenario.
- Every step method is annotated with a english like statement. That should match your step statement. This is how cucumber understands which method to execute while runnign a test scenario.
- Also the annotations would have some regular expressions. These are used to pass parameters in the step statement.
- Steps in the scenario are passing data to the step method in the step statements and the step methods accepts them using regex expressions.



Run The Test - See it fail !

Now that you have the test ready, i.e. scenario has been created and the glue code also exists, we should expect it to run.

Well it will run but not pass. Let's try it.

Go to the Feature file. Do a right click -> Run Feature What do you see?

It should fail saying step definitions not found.

Well you could argue that the step definitions are there in the step class we wrote.

Here is the interesting thing. We know that the code exisits in such and such class for my scenarios But jvm that is trying to run the tests doesn't know and it needs to be told that the step files exists at a certain location and only then it can pick up and run your tests.

Now is the time to introduce different ways of running cucumber-jvm features.

Time for the next chapter.

Understanding Test Runners

There are multiple ways and runners to use when it comes to cucumber feature files.

We would try to understand how to run it from the IDE first and then from a command line at a later point.

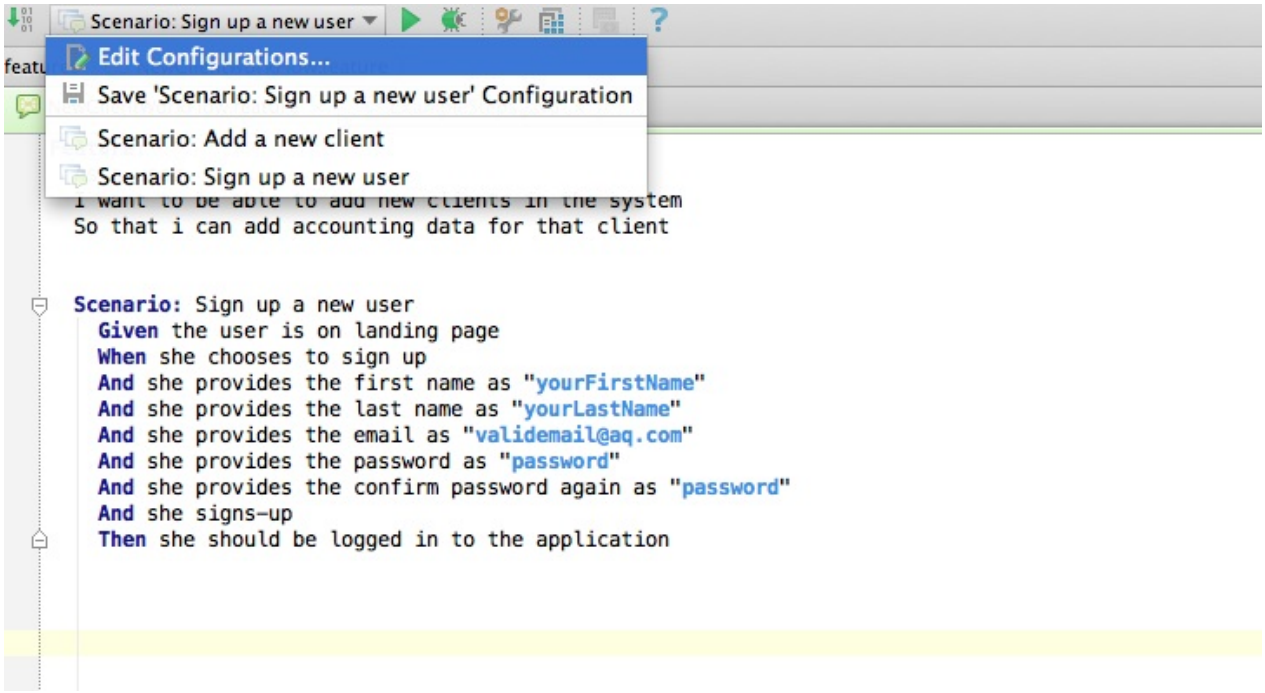
Even from the IDE, there are a couple of ways to run these feature files.

1. Running it by setting the glue code path in Run Configuration setting.
2. Running it through a specified runner class that can accomodate setting around the feature files and glue code location and much more.

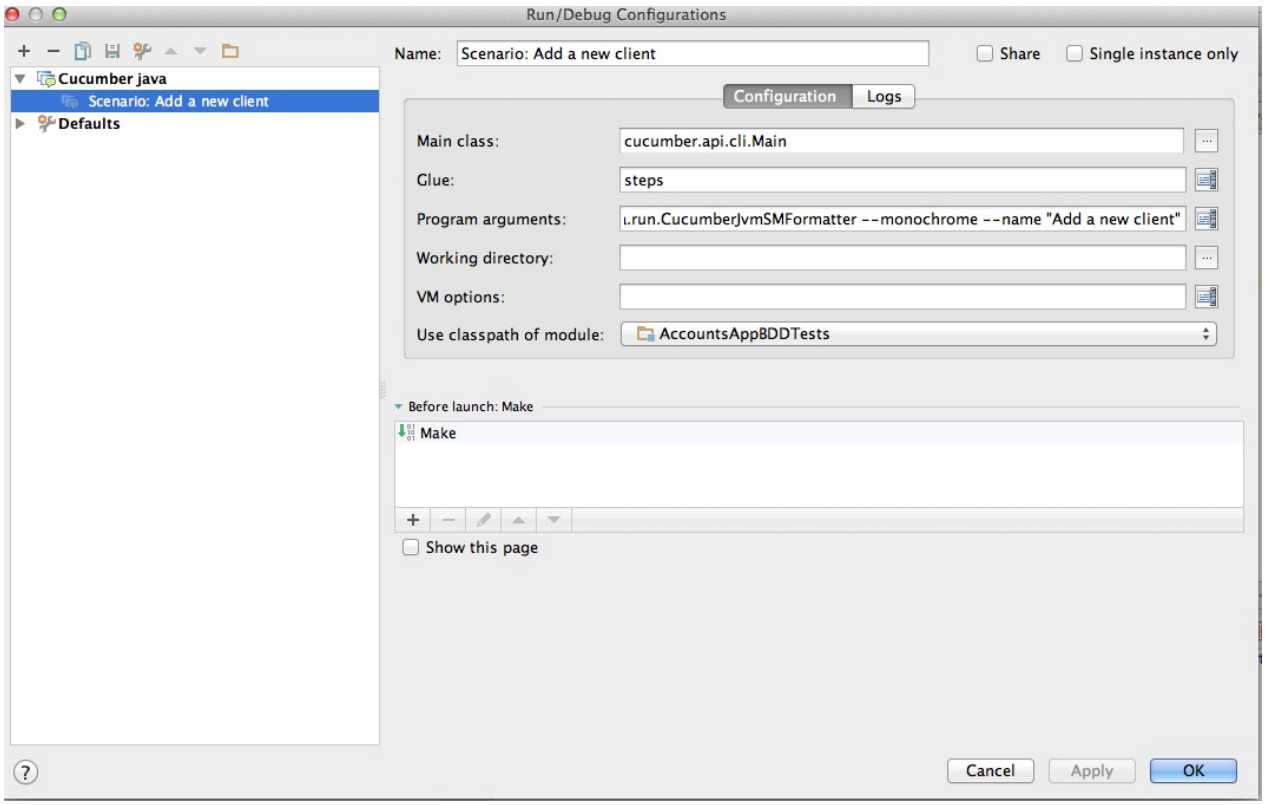
Let's explore both these options one by one.

Configure IntelliJ to Run Cucumber features

When you run the tests through the feature file, you can notice that it create a run configuration for this file. See the picture below.



Choose to Edit Configurations. In the configurations window, specify the path to the steps package in the Glue parameter. That should be it.


Run/Debug Configurations
Cucumber java
Scenario: Add a new client
Defaults
Name:
Scenario: Add a new client
Share
Single instance only
Configuration
Logs
Main class:
cucumber.api.cli.Main
Glue:
steps
Program arguments:
ı.run.CucumberJvmSMFormatter --monochrome --name "Add a new client"
Working directory:
VM options:
Use classpath of module:
AccountsAppBDDTests
Before launch: Make
Make
Show this page
Cancel
Apply
OK

Run The Test - See it Pass

Now that you have specified the location for glue code, jvm now understands that while running the feature file i have to pick up the step methods from inside the stated package.

- Go back to feature file. *Right Click -> Run fetaure* Voila, it runs. Well it passes or not could still be seen. :)

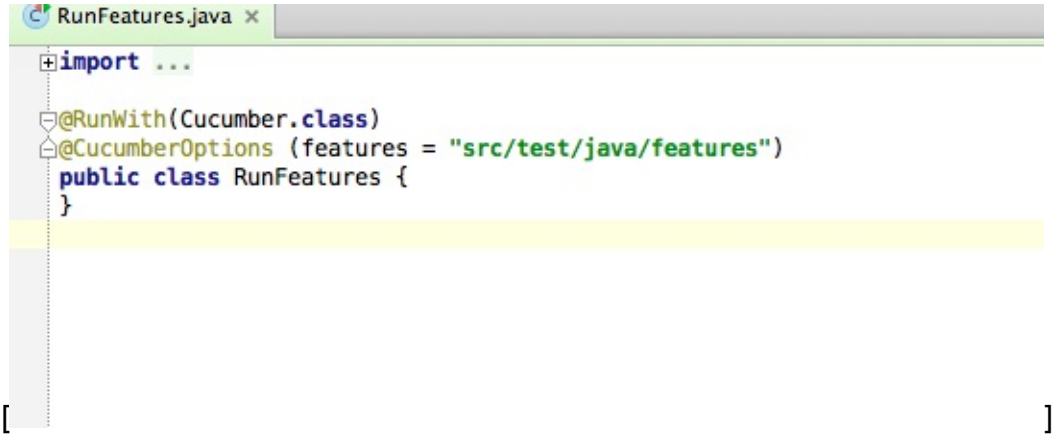
Runner Class

This is another way of configuring the IDE to run feature files.

- Create a new class called "RunFeatures.java".
- This class doesn't need to have any code.
- It just need annotations to understand that cucumber features would be run through him and you can specify feature files to be picked up plus the steps package location.
- There are bunch of other parameters that it can take, to be discussed later.

Here is a sample code for the same.

```
@RunWith(Cucumber.class)
@CucumberOptions (features = "src/test/java/features/")
public class RunFeatures {
}
```

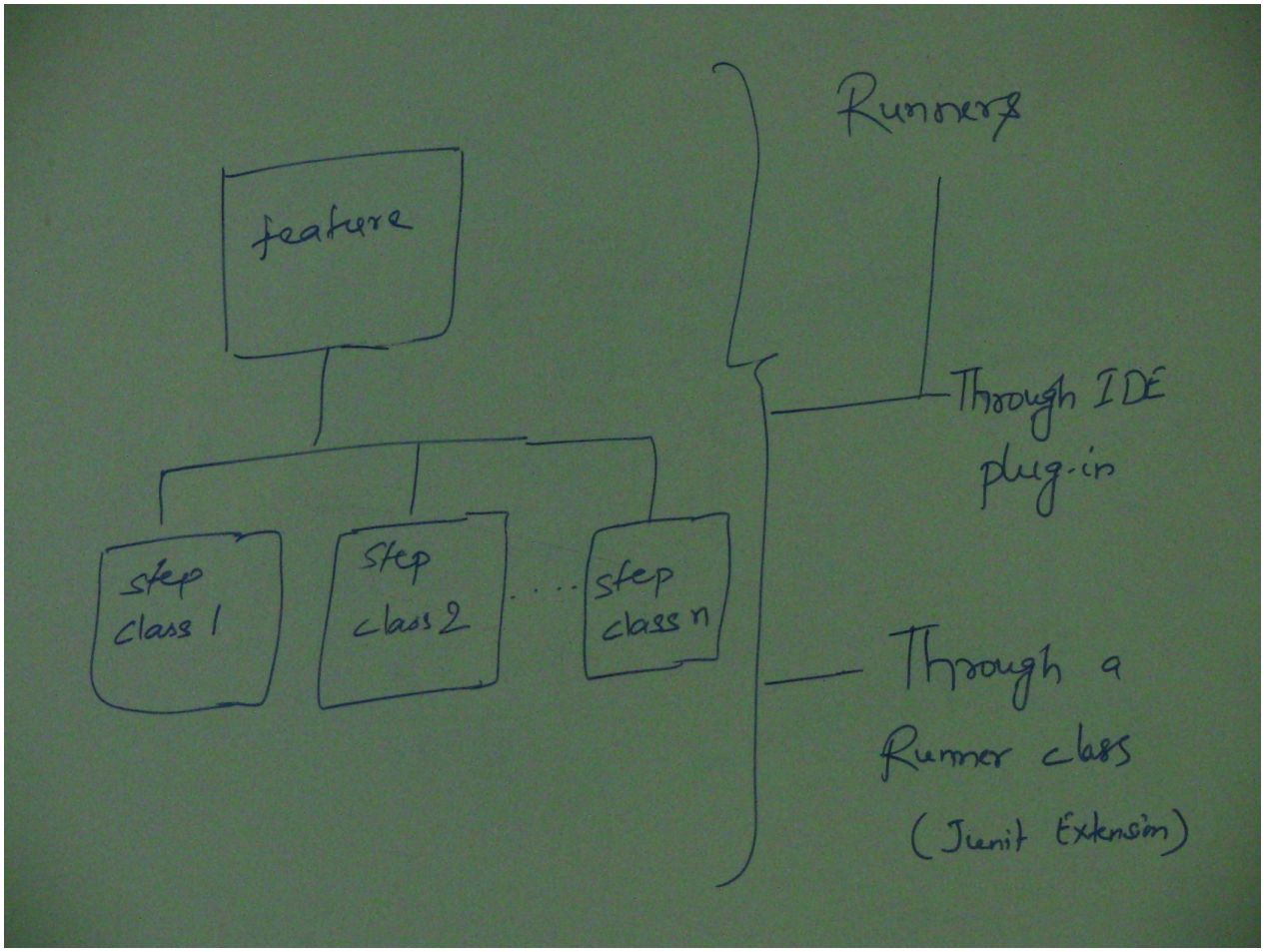


[ ]

Now if you just do a right click and Run "RunFeatures", that should run the tests.

How Awesome!

Runner options - Consolidated



Consolidate - Journey So far

This is what we learnt so far.

- Getting the machine ready for writing tests.
- Creating a basic feature file and the glue code for the same.
- Understood some basic Feature file keywords.
- Running the tests through IDE.

Now that our test runs and it works in theory, let look at the code so far in terms of an efficient test framework.

Here are some of the problems i see, the obvious once:

- All glue code is in one step class. We need to organize them better.
- All the driver code seems to be lying in the same class with webelement identifiers hard coded.
- Although we are talking to three different pages in the scenario, the code is all at one place i.e there are no page objects.

In the following articles we would fix these problems one by one and slowly evolve the framework.

Underlying Code - Page Object Pattern

Time to introduce some framework concepts. If you notice, we interacted with three different pages in our scenario.

- **Landing Page** - from where we chose to sign up
- **SignUp Page** - where we entered details of the new user and committed
- **Home Page** - of the application you see as a signed-in user

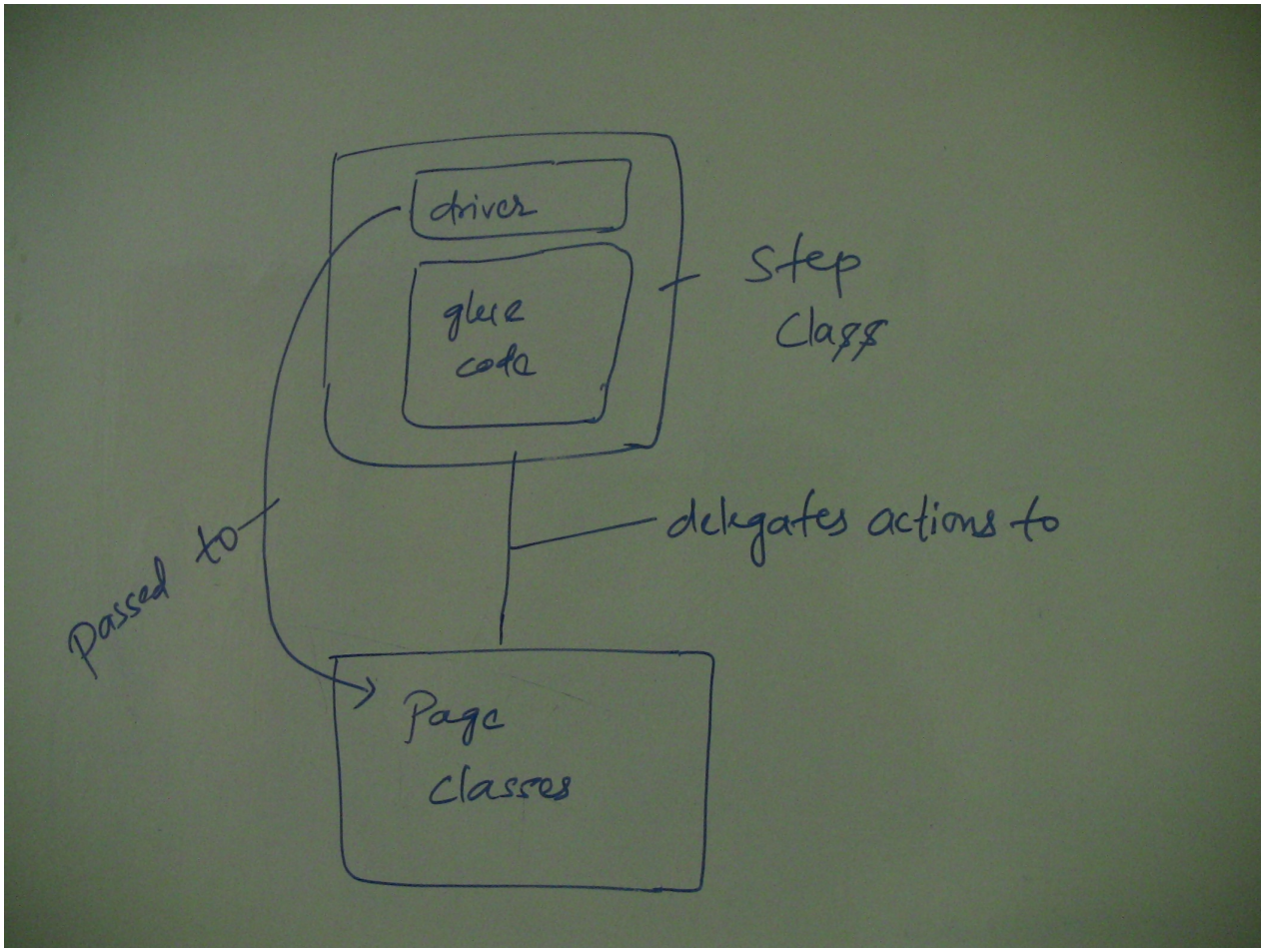
Page object pattern in test automation is a concept that basically advocates the idea of creating one java class per page of the application.

The idea is to encapsulate all the controls that are available on the page and also all the actions that can be performed on a specific page.

This way you can also re-use the code written on a page for different test scenarios we would be creating.

Having said that, in our case we should be creating 3 different page classes. And based on above understanding

- **Landing page** - should encapsulate the sign up button and the sign up method.
- **SignUp Page** - should encapsulate all the user details field inputs and the actions for the same.
- **Home Page** - should encapuslate the logged-in user home page controls and the related actions.


driver
glue code
Step Class
delegates actions to
Passed to
Page classes

Page Classes

Now that we understand the page object pattern, let's create the page for our example scenario.

First let us create a package named pages inside src->test->java. i.e. at the same level as features and step packages.

Once the package is ready, we will create three page classed as discussed one each for Landing page, SignUp page and Home page.

Also, all the code that we wrote in example steps can be moved to these newly created pages.

Let us do it one by one.

LandingPage

- It should encapsulate the sign up link.
- It should encapsulate the action to click on the sign up link.

Here is how it should look.

```
public class LandingPage {

WebDriver driver;

public LandingPage(WebDriver driver) {
    this.driver = driver;
}


public void she_chooses_to_sign_up() throws Throwable {
    driver.findElement(By.linkText("Sign up")).click();
}
}
```

Notice the constructor for this class. It takes a web driver instance. This is needed because we are writing code to click a link using the driver in this class.

And similarily, all the other page classes would also need the driver as they would be having methods to work with controls on the pages like enter some text, select from a list, click a button etc.

On similar lines,

SignupPage

- should encapsulate all the user detail input controls like firstName, lastName etc
- should encapsulate all the methods that would allow entering data into these controls

So SignupPage class should look like this.

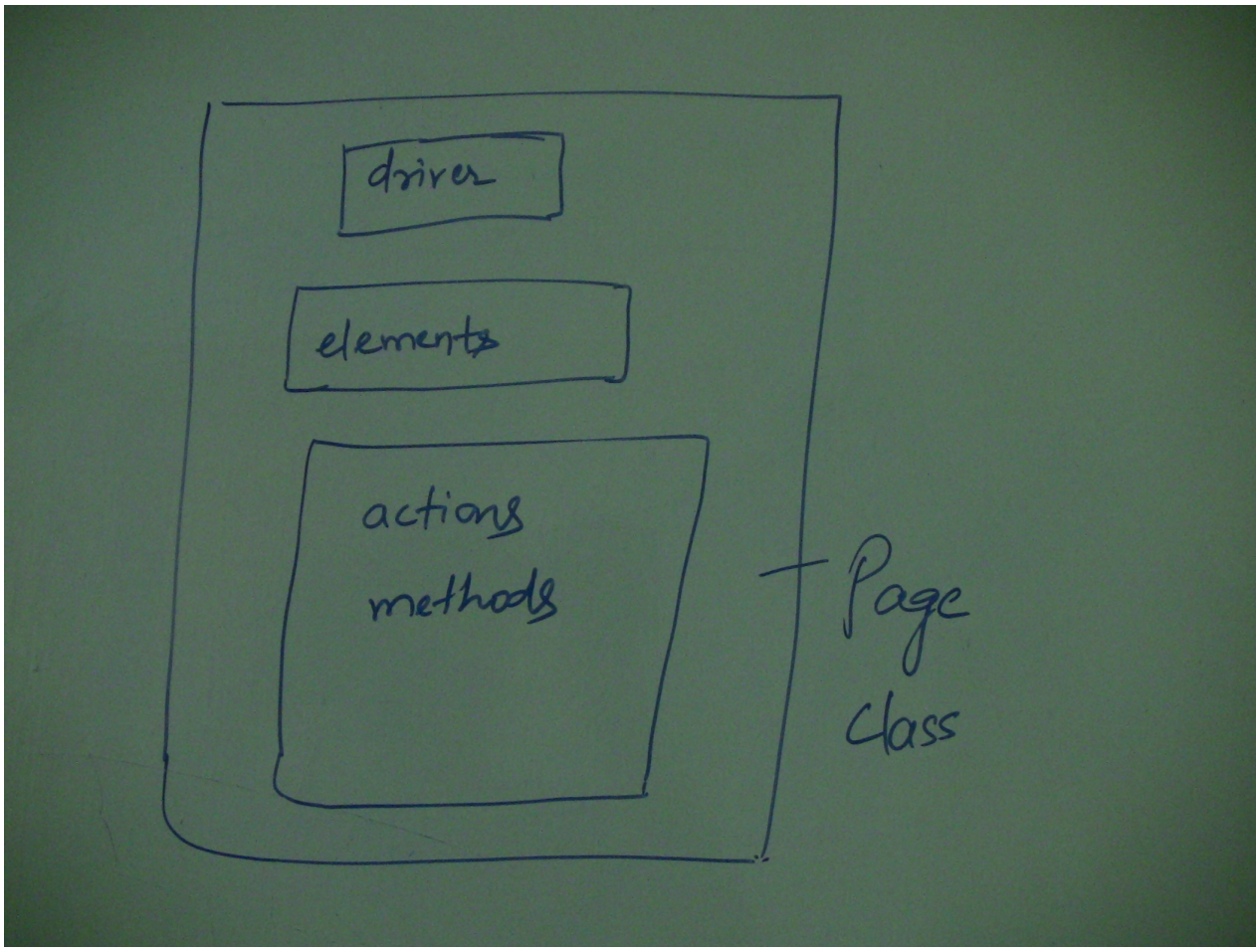
```
public class SignupPage {

WebDriver driver;

public SignupPage(WebDriver driver) {
    this.driver = driver;
}

public void she_provides_the_first_name_as(String firstName) throws Throwable {
    driver.findElement(By.id("user_first_name")).sendKeys(firstName);
}


public void she_provides_the_last_name_as(String lastName) throws Throwable {
    driver.findElement(By.id("user_last_name")).sendKeys(lastName);
}

public void she_provides_the_email_as(String email) throws Throwable {
    driver.findElement(By.id("user_email")).sendKeys(email);
}

public void she_provides_the_password_as(String password) throws Throwable {
    driver.findElement(By.id("user_password")).sendKeys(password);
}

public void she_provides_the_confirm_password_again_as(String confirmPassword) throws
Throwable {
    driver.findElement(By.id("user_password_confirmation")).sendKeys(confirmPassword);
}


public void she_signs_up() throws Throwable {
    driver.findElement(By.name("commit")).click();
}
}
```

HomePage

- should encapsulate the logout link
- should encapsulate the methods that tells us whether the signout link is displayed or not.

Here is how this class would look.

```
public class HomePage {

WebDriver driver;

public HomePage(WebDriver driver) {
    this.driver = driver;
}

public boolean isSignOutLinkDisplayed() throws Throwable {
   return driver.findElement(By.cssSelector("a[href='/users/sign_out']")).isDisplayed(
);
}
}
```



Calling page objects from steps

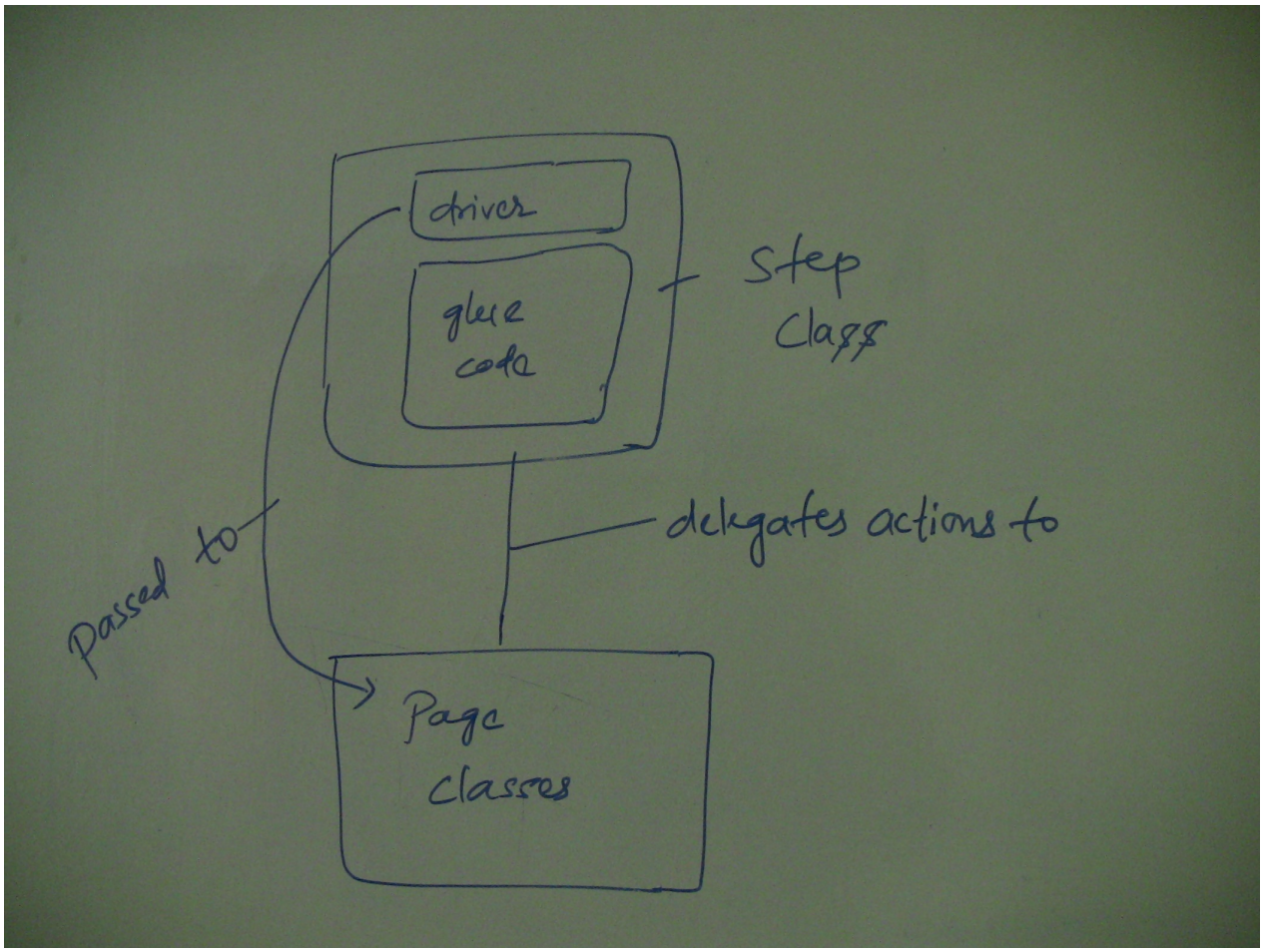
Now that we have create page objects and moved the actual code of doing text enter, click buttons etc on to the pages, the step classes should be just calling the methods from these pages.

What that means is your ExampleSteps class should change to more of a delegator class to pages and look like the following

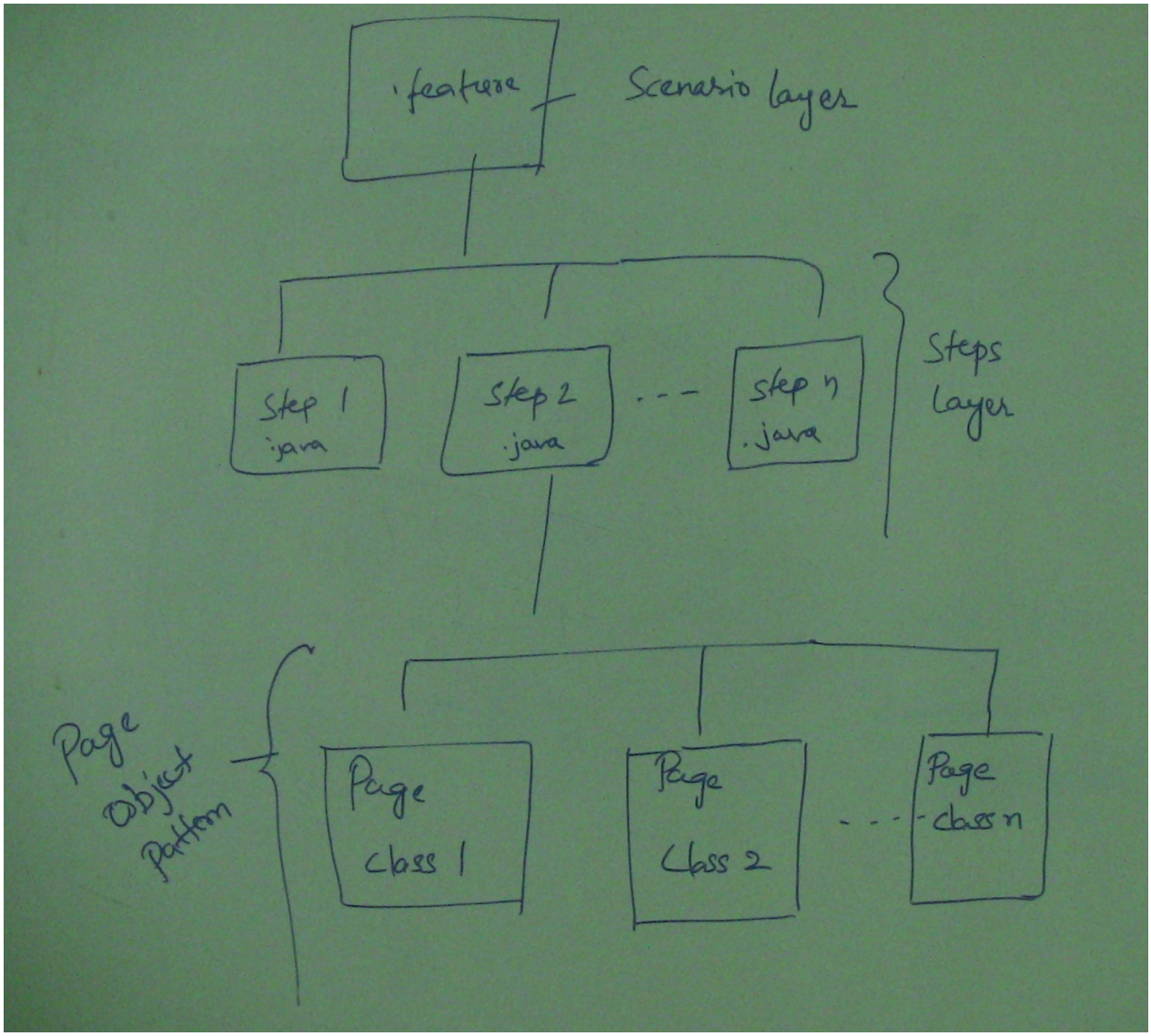
```
public class ExampleSteps {

WebDriver driver = new FirefoxDriver();

@Given("^the user is on landing page$")
public void setup() throws Throwable {
    driver.get("http://accountsdemo.herokuapp.com");
    driver.manage().window().maximize();
}


@When("^she chooses to sign up$")
public void she_chooses_to_sign_up() throws Throwable {
    new LandingPage(driver).she_chooses_to_sign_up();
}


@And("^she provides the first name as ([^\"]*)$")
public void she_provides_the_first_name_as(String firstName) throws Throwable {
    new SignupPage(driver).she_provides_the_first_name_as(firstName);
}


@And("^she provides the last name as ([^\"]*)$")
public void she_provides_the_last_name_as(String lastName) throws Throwable {
    new SignupPage(driver).she_provides_the_last_name_as(lastName);
}

@And("^she provides the email as ([^\"]*)$")
public void she_provides_the_email_as(String email) throws Throwable {
    new SignupPage(driver).she_provides_the_email_as(email);
}

@And("^she provides the password as ([^\"]*)$")
public void she_provides_the_password_as(String password) throws Throwable {
    new SignupPage(driver).she_provides_the_password_as(password);
}

@And("^she provides the confirm password again as ([^\"]*)$")
```

```
public void she_provides_the_confirm_password_again_as(String confirmPassword) throws
Throwable {
    new SignupPage(driver).she_provides_the_confirm_password_again_as(confirmPassword)
;
}


@And("^she signs-up$")
public void she_signs_up() throws Throwable {
    new SignupPage(driver).she_signs_up();
}

@Then("^she should be logged in to the application$")
public void she_should_be_logged_in_to_the_application() throws Throwable {
    Assert.assertTrue(new HomePage(driver).isSignOutLinkDisplayed());
}
}
```

Now, notice how we create the driver in this class. And everytime we delegate a step to the page class method, we create a new page instance with the same driver.



Framework This Far



Understanding the Code - Page Factory Pattern

Now that we have seen, how the code is flowing through from steps in a scenario to methods in the pages, lets try to understand different aspects of the page class a little better.

Articles in this chapter revolve around how to efficiently handle the element identification strategy in your test framework.

Element Identifiers - Some basics

An element identifier is needed to get a handle of the specific element on which we want to perform certain action.

It is a way to identify an element. The various options that selenium allows you to use for identifying an element are:

- By id attribute
- By name attribute
- By css class name attribute
- By css locator of the element
- By xpath locator of the element

If you look at the page class we wrote, we have used one of these startegies to locate elements that the step had to act upon on.

One thing to note in the current implementation though is, the fact that these element identification startegies are tightly coupled with the method in which it is called.

For example "first name" is found by id in the method where we are entering first name. Now suppose there are other methods in the class that also want to use the first name element. These methods will again try to locate the element with similar or a different location strategy, which is not really needed.

Further this would add to unnecessary duplication and a maintainence overhead. In case the id attribute for first name changes, we will have to update the same at multiple places now.

Abstract Element Locators

The obvious way out to get rid of duplication is to abstract the id's from the methods. An example of that using the signup page class would look like this.

```
public class SignupPage {

WebDriver driver;

By firstNameTextBoxLocator = By.id("user_first_name");
By lastNameTextBoxLocator = By.id("user_last_name");
By emailTextBoxLocator = By.id("user_email");
By passwordTextBoxLocator = By.id("user_password");
By confirmPasswordTextBoxLocator = By.id("user_password_confirmation");
By signupButtonLocator = By.name("commit");


public SignupPage(WebDriver driver) {
    this.driver = driver;
}

public void she_provides_the_first_name_as(String firstName) throws Throwable {
    driver.findElement(firstNameTextBoxLocator).sendKeys(firstName);
}


public void she_provides_the_last_name_as(String lastName) throws Throwable {
    driver.findElement(lastNameTextBoxLocator).sendKeys(lastName);
}

public void she_provides_the_email_as(String email) throws Throwable {
    driver.findElement(emailTextBoxLocator).sendKeys(email);
}

public void she_provides_the_password_as(String password) throws Throwable {
    driver.findElement(passwordTextBoxLocator).sendKeys(password);
}

public void she_provides_the_confirm_password_again_as(String confirmPassword) throws
Throwable {
    driver.findElement(confirmPasswordTextBoxLocator).sendKeys(confirmPassword);
}


public void she_signs_up() throws Throwable {
    driver.findElement(signupButtonLocator).click();
}
}
```

Observe that we have pulled out the locator strategies as variables. Now if we can call the same locator as many places we need and refer to it by the variable name.

Also, if the locator changes we just need to change it at one common place.

Page Factory Pattern

In the previous example, though we have abstracted the locators, we still keep repeating the part of driver.findElement while working with elements.

The PageFactory pattern allows you to get rid of that as well.This helps a lot in making the code look more clean and readable.

Let's look at the code first and then we talk about it later.

```
public class SignupPage {

WebDriver driver;


@FindBy(id = "user_first_name")
private WebElement firstNameTextBox;

@FindBy(id = "user_last_name")
private WebElement lastNameTextBox;

@FindBy(id = "user_email")
private WebElement emailTextBox;

@FindBy(id = "user_password")
private WebElement passwordTextBox;

@FindBy(id = "user_password_confirmation")
private WebElement confirmPasswordTextBox;

@FindBy(name = "commit")
private WebElement signupButton;


public SignupPage(WebDriver driver) {
    this.driver = driver;
    PageFactory.initElements(driver, this);
}

public void she_provides_the_first_name_as(String firstName) throws Throwable {
    firstNameTextBox.sendKeys(firstName);
}


public void she_provides_the_last_name_as(String lastName) throws Throwable {
    lastNameTextBox.sendKeys(lastName);
}
```

```
public void she_provides_the_email_as(String email) throws Throwable {
    emailTextBox.sendKeys(email);
}

public void she_provides_the_password_as(String password) throws Throwable {
    passwordTextBox.sendKeys(password);
}

public void she_provides_the_confirm_password_again_as(String confirmPassword) throws
Throwable {
    confirmPasswordTextBox.sendKeys(confirmPassword);
}


public void she_signs_up() throws Throwable {
    signupButton.click();
}

}
```

Now that you have seen the code lets understand the chnaged pieces on by one.

- **@FindBy** annotation - The annotation can take the locator strategy as paramters and when the code tries to use the webelement in the method, this is used to find the element.
- **WebElements as variables** - Another thing you notice is that we are directly defining the Webelements as variables. This is what "driver.findElement" call was returning. This helps to directly access the element and perform actions on it.

We can do the same for the other 2 page classes.

Evolving The Framework - Structuring Step Classes

Now that we have a fair bit of understanding around pages and the locator strategies, let us look back at the step classes.

The primary job that we have for the step classes so far is to:

- Provide the glue code for the english like scenario so that it finds something to execute.
- Delegate the actual action to the pages which encapsulate the actual method to interact with the browser.

Given the fact that we will have a fair number of test scenarios in any given applications and hence quite a lot of steps across the applications, we also need to think of a strategy to manage the various steps in an elegant fashion.

Creating the DriverFactory

For now i will just move the driver creation code to a separate class (DriverFactory, for now i am keeping it in the same steps package) and extend the ExampleSteps class from the DriverFactory class.

So basically, our new classed would like the following

DriverFactory

```
public class DriverFactory  {

protected WebDriver driver = new FirefoxDriver();
}
```

Creating page level step classes

Now is the time to break the ExampleSteps class into the following three classes and keep them all in steps package.

LandingPageSteps

```
public class LandingPageSteps extends DriverFactory {


    @When("^she chooses to sign up$")
    public void she_chooses_to_sign_up() throws Throwable {
        new LandingPage(driver).she_chooses_to_sign_up();
    }


}
```

SignupPageSteps

```
public class SignUpPageSteps extends DriverFactory {


@And("^she provides the first name as ([^\"]*)$")
public void she_provides_the_first_name_as(String firstName) throws Throwable {
    new SignupPage(driver).she_provides_the_first_name_as(firstName);
}


@And("^she provides the last name as ([^\"]*)$")
public void she_provides_the_last_name_as(String lastName) throws Throwable {
    new SignupPage(driver).she_provides_the_last_name_as(lastName);
}

@And("^she provides the email as ([^\"]*)$")
public void she_provides_the_email_as(String email) throws Throwable {
    new SignupPage(driver).she_provides_the_email_as(email);
}

@And("^she provides the password as ([^\"]*)$")
public void she_provides_the_password_as(String password) throws Throwable {
    new SignupPage(driver).she_provides_the_password_as(password);
}

@And("^she provides the confirm password again as ([^\"]*)$")
public void she_provides_the_confirm_password_again_as(String confirmPassword) throws
Throwable {
    new SignupPage(driver).she_provides_the_confirm_password_again_as(confirmPassword)
;
}


@And("^she signs-up$")
public void she_signs_up() throws Throwable {
    new SignupPage(driver).she_signs_up();
}
}
```

HomePageSteps

```
public class HomePageSteps extends DriverFactory {

    @Then("^she should be logged in to the application$")
    public void she_should_be_logged_in_to_the_application() throws Throwable {
        Assert.assertTrue(new HomePage(driver).isSignOutLinkDisplayed());
    }
}
```

The only thing remaining in the ExampleStep class is now the piece of code that launches the website. Let's rename this class to StartingSteps.

StartingSteps

```
public class StartingSteps extends DriverFactory {


    @Given("^the user is on landing page$")
    public void setup() throws Throwable {
        driver.get("http://accountsdemo.herokuapp.com");
        driver.manage().window().maximize();
    }


}
```

Now we have our step classes organized and the driver abstracted. But are we ready with everything. Will this code run.

Try it? What do we see?

The test runs and opens 2 browser instances and the test fails. Well looks like our DriverFactory is doing a little more than we asked for.

To break it down into exact java terms.

- All our step classes are now extending DriverFactory class.
- Everytime the scenario reaches to a new step class it follows inheritance, and the Driverfactory is asking it to create a new Driver every time.
- In our case, it creates one for StartingSteps and then again for LandingPageSteps.

What we need here is that even if all the step classes are extending the driver factory, they should not be creating a new driver instance every time. Rather they should be creating a single instance and reusing the same every time.

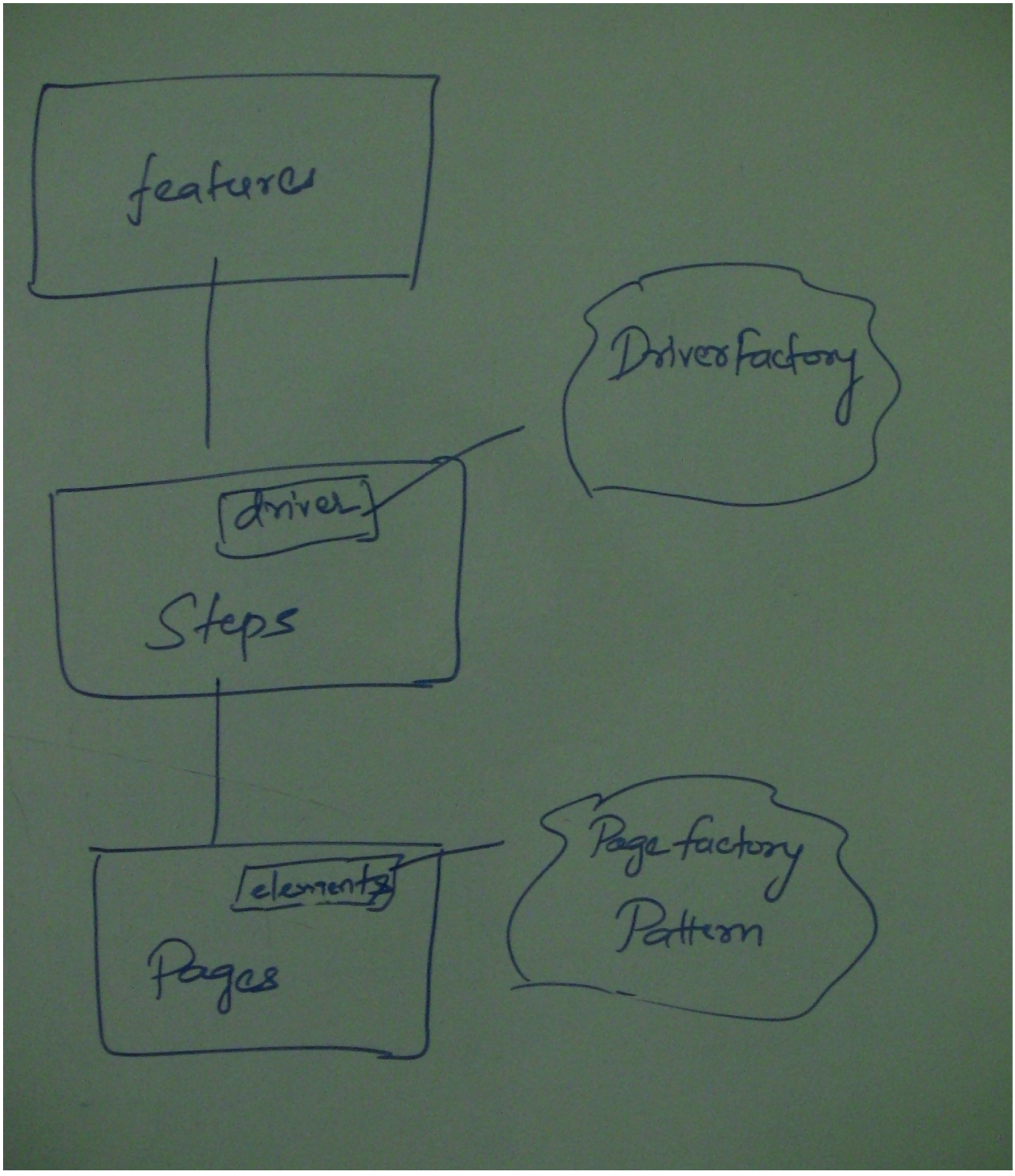
The easiest way to achieve this is to make the driver object static.

So the DriverFactory class would now look like the following

```
public class DriverFactory  {

    protected static WebDriver driver = new FirefoxDriver();
}
```

If you run the test now, all should hold good for now.

Consolidation - 2nd Milestone



Cucumber - More Details

The following articles will take you through some of the basic and mostly used cucumber features.

This is where the real beauty is, and you will shortly understand why it wins hands down when it comes to test frameworks.

Cucumber Tagging

Tags are a feature in cucumber that allows you to categorize tests. That way you can organize your feature and scenarios better.

And once we have tests belonging to a certain category we can choose to run one or more combinations of these tags with our test runners.

Scenario tagging

Consider the following code to understand how scenarios can be tagged.

```
Feature:
  As a user
  I want to be able to add new clients in the system
  So that i can add accounting data for that client

  Background:
    Given the user is on landing page
    When she chooses to sign up

  @Signup-Simple
  Scenario: Sign up a new user
    And she provides the first name as Sukesh
    And she provides the last name as Kumar
    And she provides the email as validemail@aq.com
    And she provides the password as password
    And she provides the confirm password again as password
    And she signs-up
    Then she should be logged in to the application

  @Signup-DataDriven
  Scenario Outline: Data driving new user sign-up
    And she provides the first name as <firstName>
    And she provides the last name as <lastName>
    And she provides the email as <email>
    And she provides the password as <password>
    And she provides the confirm password again as <password>
    And she signs-up
    Then she should be logged in to the application
  Examples:
    | firstName | lastName | email             | password |
    | Sukesh    | Kumar    | validemail@aq.com | password |
```

If you notice the syntax the tag name has to be prefixed by @. Also the tag needs to be placed right on top of the scenario keyword.

Feature tagging

Not only scenarios, the feature file itself can be tagged. Consider the following code to understand how feature files are tagged.

```
@Signup @TaggedAgain
Feature:
  As a user
  I want to be able to add new clients in the system
  So that i can add accounting data for that client

  Background:
    Given the user is on landing page
    When she chooses to sign up

  @Signup-Simple
  Scenario: Sign up a new user
    And she provides the first name as Sukesh
    And she provides the last name as Kumar
    And she provides the email as validemail@aq.com
    And she provides the password as password
    And she provides the confirm password again as password
    And she signs-up
    Then she should be logged in to the application

  @Signup-DataDriven
  Scenario Outline: Data driving new user sign-up
    And she provides the first name as <firstName>
    And she provides the last name as <lastName>
    And she provides the email as <email>
    And she provides the password as <password>
    And she provides the confirm password again as <password>
    And she signs-up
    Then she should be logged in to the application
  Examples:
    | firstName | lastName | email             | password |
    | Sukesh    | Kumar    | validemail@aq.com | password |
```

The tagging syntax here is the same as for the scenarios. Also notice that the feature file has 2 different tags. Any feature file or scenario can be tagged more than one time and all the tags should be separated by space.

Tag Inheritance

A feature tag is inherited by all the scenarios of the feature file.

In the above mentioned feature file, if we choose to run all tests which are tagged @Signup, both the scenarios would run.

Obviously, the same should happen if we choose to run all tests that are tagged @TaggedAgain.

Running cucumber tests based on tags

The tags can be used when specifying what tests to run through any of the running mechanism.

I am showing here how it is done using the Runner class we had written earlier.

```
@RunWith(Cucumber.class)
@CucumberOptions (features = "src/test/java/features/", tags = "@Signup-DataDriven")
public class RunFeatures {
}
```

So, tags is just another parameter in the cucumber options annotation. We can also pass multiple tags as values separated by commas if we need so.

Also, we will see other ways to do it through the build task in upcoming chapters.

Data Driven Testing Using Cucumber

Time to introduce a new Cucumber keyword.

Scenario Outline - This is used when you want to run the same scenario for 2 or more different set of test data. e.g. In our scenario, if you want to register another user you can data drive the same scenario twice.

Examples - All scenario outlines have to be followed with the Examples section. This contains the data that has to be passed on to the scenario.

Let see how your scenario would look like if you were to data drive the sign up new user scenario.

```
Scenario Outline: Data driving new user sign-up
  Given the user is on landing page
  When she chooses to sign up
  And she provides the first name as <firstName>
  And she provides the last name as <lastName>
  And she provides the email as <email>
  And she provides the password as <password>
  And she provides the confirm password again as <password>
  And she signs-up
  Then she should be logged in to the application
Examples:
  | firstName | lastName | email             | password |
  | Sukesh    | Kumar    | validemail@aq.com | password |
```

We can pass as many data rows in the example table as we want and the same scenario would run once for every data row.

Also notice that the table header names are same as the variable names passed in the steps.

Copy this scenario into your feture file and run the same.

Cucumber - Background and Hooks

With the overall skeleton of the framework set, lets dive deeper into cucumber as the testing framework.

Cucumber by default provides a lot of in-built functionality to handle and manage your test suit better.

Articles in this chapter revolve around understanding cucumber way of organizing tests and how to set up your test run in an efficient manner.

Understanding Background in Cucumber

Background in cucumber is a concept that allows you to specify steps that are pre-requisite to all the scenarios in a given feature file.

Precisely doing what a setup method does in your junit or testNG. For example, in both the scnearios we have written so far the user needs to be on the landing page to start the sign-up process. We can treat the step to start the sign-up process from the landing page as a Background step in the feature file and then write multiple sign-up scenario (positive and negative) in the feature file.

Let us see some code that would make it more clear. This is how our feature file would look like with Background step.

```
Feature:
  As a user
  I want to be able to add new clients in the system
  So that i can add accounting data for that client

  Background:
    Given the user is on landing page
    When she chooses to sign up

  Scenario: Sign up a new user
    And she provides the first name as Sukesh
    And she provides the last name as Kumar
    And she provides the email as validemail@aq.com
    And she provides the password as password
    And she provides the confirm password again as password
    And she signs-up
    Then she should be logged in to the application


  Scenario Outline: Data driving new user sign-up
    And she provides the first name as <firstName>
    And she provides the last name as <lastName>
    And she provides the email as <email>
    And she provides the password as <password>
    And she provides the confirm password again as <password>
    And she signs-up
    Then she should be logged in to the application
  Examples:
    | firstName | lastName | email             | password |
    | Sukesh    | Kumar    | validemail@aq.com | password |
```

Notice how background is defined in the feature file. Now all the scenarios in this class would run the Background steps before running any scenario.

Understanding Hooks

Somewhat similar to background in terms of feature it provides but the scope is a little different. Let's see how?

Scenario Hooks

Scenario hooks

As the name suggests, they would allow us to hook some piece of code before and after running any scenario in our framework.

They are somewhat similar to Background but the scope of a Background is limited only to a specific feature file. Whereas, the hooks are applicable to all scenarios across all the feature files.

They can be used in 2 ways:

- @Before
- @After

@Before when specified runs before executing any scenario.

@After when specified runs right after executing any scenario irrespecive of the scenario's pass/fail status.

Consider the following code to understand the hooks.

```
public class StartingSteps extends DriverFactory {


    @Given("^the user is on landing page$")
    public void setup() throws Throwable {
        driver.get("http://accountsdemo.herokuapp.com");
        driver.manage().window().maximize();
    }

    @Before
    public void beforeScenario(){
        System.out.println("this will run before the actual scenario");
    }

    @After
    public void afterScenario(){
        System.out.println("this will run after scneario is finished, even if it faile
d");
    }

}
```

Now run the test again to see the output from the hooks.

Tagged Hooks

Tagged Hooks are much like the scenario hooks but the only difference is that they are executed before and after the specified tag.

So basically, they can also be run in the following two ways:

- Before ('tagName')
- After ('tagName')

This can be used when we need some kind of a feature level setup and teardown, assuming that all the scneario are tagged with the same feature name.

In our example scenario, consider the following code to understand the concept.

```
public class StartingSteps extends DriverFactory {


    @Given("^the user is on landing page$")
    public void setup() throws Throwable {
        driver.get("http://accountsdemo.herokuapp.com");
        driver.manage().window().maximize();
    }

    @Before
    public void beforeScenario(){
        System.out.println("this will run before the actual scenario");
    }

    @After
    public void afterScenario(){
        System.out.println("this will run after scneario is finished, even if it faile
d");
    }


    @Before("Signup-DataDriven")
    public void signupSetup(){
        System.out.println("This should run everytime before any of the @Signup-DataDr
iven tagged scenario is going to run");
    }

    @After("Signup-DataDriven")
    public void signupTeardown(){
        System.out.println("This should run everytime after any of the @Signup-DataDri
ven tagged scenario has run");
    }

}
```

Cucumber - DataTables

So far we have been passing one arguement in every step of our scenario.

There could be times when we need to pass more than one arguements from a step. This is were datatables are very handy.

Let's try a different way of writing the same scenario that we have using a datable example.

Consider the following scenario.

```
Scenario: Sign-up a new user with datatable example
  Given the user is on landing page
  When she chooses to sign up
  And she provides the her details as follows:
    | firstName | lastName | email             | password |
    | Sukesh    | Kumar    | validemail@aq.com | password |
  And she signs-up
  Then she should be logged in to the application
```

So this scenario would result in the same behavior like the other two we have written so far. The only difference here is the way we are passing the data values.

Notice that the first 2 lines were moved to Background in the earlier chapter. You will have to remove the same from this scenario when your try to run it, otherwise cucumber will try to execute it twice and that may throw some unwanted behavior.

Also, understand that this is not a data driven scenario. It willl run only once and the datatable's scope is limited to the step where it is passed.

Parsing DataTables - Type Transformation

Datatables when passed as arguements from a cucumber step can be parsed in different ways. In this section we will understand how they can be parsed to a java object.

Also, this is the most recommended practice for parsing datatables.

Cucumber can understand a table as a list of certain type. e.g. in the current context what we are trying to pass from the step is user details (name, email, password). So the datatable can be transformed into a list of userdetail objects. The underlying assumption here is the fact that userdetail is a separate class with attributes like firstName, lastName, email, password.

Consider the following code to understand the step implementation. We can put this in the SignUpSteps class.

```
@And("^she provides the her details as follows:$")
    public void she_provides_the_her_details_as_follows(List<UserDetails> users) throw
s Throwable {
        UserDetails userToBeSignedUp = users.get(0);

        new SignupPage(driver).she_provides_the_first_name_as(userToBeSignedUp.firstNa
me);
        new SignupPage(driver).she_provides_the_last_name_as(userToBeSignedUp.lastName
);
        new SignupPage(driver).she_provides_the_email_as(userToBeSignedUp.email);
        new SignupPage(driver).she_provides_the_password_as(userToBeSignedUp.password)
;
        new SignupPage(driver).she_provides_the_confirm_password_again_as(userToBeSign
edUp.password);

    }
```

Notice the fact that it needs the UserDetails class as well. For now let's create an inner class with that name in the same SignupStepsClass.

```
private class UserDetails {
        public String firstName;
        public String lastName;
        public String email;
        public String password;
    }
```

This is a plain old java class that will facilitate the transformation of the data table to an object of this class. Also mark the fact that the variables in this class and the column header in the step should match.

Likewise, whenever we need to pass a large set of related data in our steps, we can create similar domain object classes and use the same to do a data table transformation.

Now that we have understood the concept, let's move the class UserDetails.java to a separate package. We will create one called domain inside src and move the class inside that domain package.

Test Framework - Driver Abstratction

Time to relook at the DriverFactory class and get introduced to another interesting piece in the whole framework ecosystem, i.e. the driver abstraction.

Driver Usage - so far

How are we using it thus far?

- We first created a driver instance in the ExmapleSteps class.
- Then at the point where we separated step classes, we moved the driver creation code to DriverFactory.
- Next we made the driver instance static so that all the step classes could use the same instance.
- Also, we have been passing the same driver instance to all the page classes.

Lets understand who all needs this driver.

- StartingSteps class needs it, to launch the application using driver.get("applicationUrl") method call.
- All the pages need it, to do page level action on the browser using this driver, like click, sendkeys etc.

Problems with the current approach.

Consider the case when we want to run all the tests in a feature file at once. Ideally we would want to start a new browser instance as part of every test, and then close the browser after every test.

But is that possible, given the current code which only instantiates a static driver instance just once? NO.

Let's assume we try to kill the browser after first test. This driver instance is an static instance. So although the browser will get closed, the instance being static will not be completely dead.

Now when the next test tries to use the driver again to launch a new browser instance, the execution will bomb. Reason being the fact that the static instance is not completely dead and this test will try to use the same instance.

The following articles will talk about how to abstract the driver and instantiate it in the right way so that the usage of the driver object is clean and clear.

DriverFactory - Create and Destroy

Consider the following code.

```
public class DriverFactory {

    protected static WebDriver driver;


    public DriverFactory() {
        initialize();
    }

    public void initialize() {
        if (driver == null)
            createNewDriverInstance();
    }

    private void createNewDriverInstance() {
        driver = new FirefoxDriver();
    }

    public WebDriver getDriver() {
        return driver;
    }

    public void destroyDriver() {
        driver.quit();
        driver = null;
    }
}
```

Now we can use the destroyDriver() method to kill the driver in the right way. So if all the tests want to start their own driver instance and kill the same at the end, this code would keep it clean.

So how do we use this code in the right way, let's see.

Driver Instance - With hooks

The best place to implement this code is the @Before and @After scenario hooks given that we want to start a new driver instace before the scenario starts and kill the browser after the scenario is completed.

Consider the following code for StartingSteps class to demonstrate this concept.

```
public class StartingSteps {

    private WebDriver driver;


    @Before
    public void beforeScenario() {
        driver = new DriverFactory().getDriver();
        System.out.println("this will run before the actual scenario");
    }

    @After
    public void afterScenario() {
        new DriverFactory().destroyDriver();
        System.out.println("this will run after scneario is finished, even if it faile
d");
    }


    @Given("^the user is on landing page$")
    public void setup() throws Throwable {
        driver.get("http://accountsdemo.herokuapp.com");
        driver.manage().window().maximize();
    }


    @Before("@Signup-DataDriven")
    public void signupSetup() {
        System.out.println("This should run everytime before any of the @Signup-DataDr
iven tagged scenario is going to run");
    }

    @After("@Signup-DataDriven")
    public void signupTeardown() {
        System.out.println("This should run everytime after any of the @Signup-DataDri
ven tagged scenario has run");
    }


}
```

Driver Abstraction - Move it to a separate package

We should move the DriverFactory class to a separate package now as it doesn't belong to the steps package.

We will create a separate util package and move this class in therevfor now.

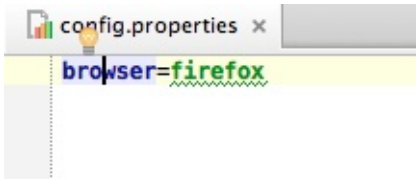
Evolving The Framework - Properties file

Another abstraction that plays a very crucial role in controlling your test run strategies. Stuff like which browser to use for test run, which environment to run test against and many such details can be easily abstracted using properties file.

Property file - Type of Driver abstraction

So a property file would be basically a key value pair. We need one file that would contain all properties that we need (type of driver for now) and we would also need some code to read the values from this property file when needed.

Let us create a properties file, right inside the src folder. We will name it config.properties and add a browser property inside it. It should look like the following.



Properties file - The reader class

Now we we need a PropertyReader class that would help us read the property values.

Consider the following code for reading the properties. Add this class under the util package.

```
public class PropertyReader {

    Properties properties = new Properties();
    InputStream inputStream = null;

    public PropertyReader() {
        loadProperties();
    }

    private void loadProperties() {
        try {
            inputStream = new FileInputStream("src/config.properties");
            properties.load(inputStream);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public String readProperty(String key) {
        return properties.getProperty(key);
    }
}
```

Important points to note in the property reader class.

- Whenever a new instance of this class is created, all the properties would be loaded.
- readProperty() method would be used by implementers to read a property value.

Using the PropertyReader

Now that we have moved the browser key into the property file, lets use the same to create a driver instance in the DriverFactory class.

Consider the following code for the same.

```
public class DriverFactory {

    protected static WebDriver driver;


    public DriverFactory() {
        initialize();
    }

    public void initialize() {
        if (driver == null)
            createNewDriverInstance();
    }

    private void createNewDriverInstance() {
        String browser = new PropertyReader().readProperty("browser");
        if (browser.equals("firefox")) {
            driver = new FirefoxDriver();
        } else {
            System.out.println("can't read browser type");
        }
    }

    public WebDriver getDriver() {
        return driver;
    }

    public void destroyDriver() {
        driver.quit();
        driver = null;
    }
}
```

So we can switch the driver creation based on what value we pass from the "browser" property key.

Build Tools - Using maven to run tests

Maven works both as build tool and a dependency management tool.

What that means is you can use maven to do build tasks like compiling the test code, running the functional tests etc and at the same time use it to manage all the dependent jars that are needed for our project.

If you look back into the 1st chapter where we created the current maven project, we had started with a pom.xml which had all the dependencies defined. That helped us to get all the dependent jars for cucumber and all other related jars.

Now we will use the same pom file to create a build task that will run our cucumber tests.

Earlier, in the begining we learned about 2 different ways to run tests.

- using the cucumber-cli, through edit configuration
- using the cucumber-junit runner, through the runner class.

Both of these were used to run the tests from the IDE.

The following chapters will try doing the same from command line using a maven build task.

The JUnit way

Maven can intelligently find any JUnit test in your source code and run the same.

You just need to run a single command from inside your project directory.

```
mvn test
```

In our case, it should find the RunFeatures class and trigger the tests as configured in the CucumberOptions annotation.

But it won't. Here is the catch.

Maven expects the Junit test class to end with the word test. So we need to rename the RunFeatures class to something that end with a 'Test' word. Let's name it RunCukesTest.

So now it should look like the following:

```
@RunWith(Cucumber.class)
@CucumberOptions (features = "src/test/java/features/")
public class RunCukesTest {
}
```

If we now run the 'mvn test' command from commandline, it will run all the feature tests.

Further, CucumberOptions only talks about what features to run. Let's add some formatting options for the results of the test run.

Cucumber gives you a couple of options for result formatting.

- HTML
- Json

We will add both for our tests.

Consider the following code for result formatting.

```
@RunWith(Cucumber.class)
@CucumberOptions(features = "src/test/java/features/",
        format = {"pretty", "html:target/cucumber", "json:target/cucumber-report.json"
})
public class RunCukesTest {
}
```

Running the same tests would now produce an html result and also a json output. You can see the same in the target folder inside your project.

Formatting options for test output

Cucumber gives you a couple of options for result formatting.

- HTML
- JSON

We will add both for our tests.

Consider the following code for result formatting.

```
@RunWith(Cucumber.class)
@CucumberOptions(features = "src/test/java/features/",
        format = {"pretty", "html:target/cucumber", "json:target/cucumber-report.json"
})
public class RunCukesTest {
}
```

Running the same tests would now produce an html result and also a json output. You can see the same in the target folder inside your project.