Liferay Inc.

BDD - Good Practices Guides

1. BDD - Test Driven Development

BDD is an extension of TDD (Test Driven Development). The idea of TDD is that you start writing a unit test that fails, then you code until it succeed and repeat that cycle until the work is done.

BDD propose to apply this same workflow on another level: you start writing the acceptance tests with your clients, then you develop the features that will make those tests succeed. BDD defines a Domain Specific Language based on Natural Language to write these acceptance tests, in a way that non-technical collaborators can easily understand them. The development team will then code the features and the necessary “glue” that will translate the acceptance test in executable code.

Cucumber is a framework/tool that helps creating tests in this way. We can use Cucumber without BDD, as a tool to make functional tests very well readable.

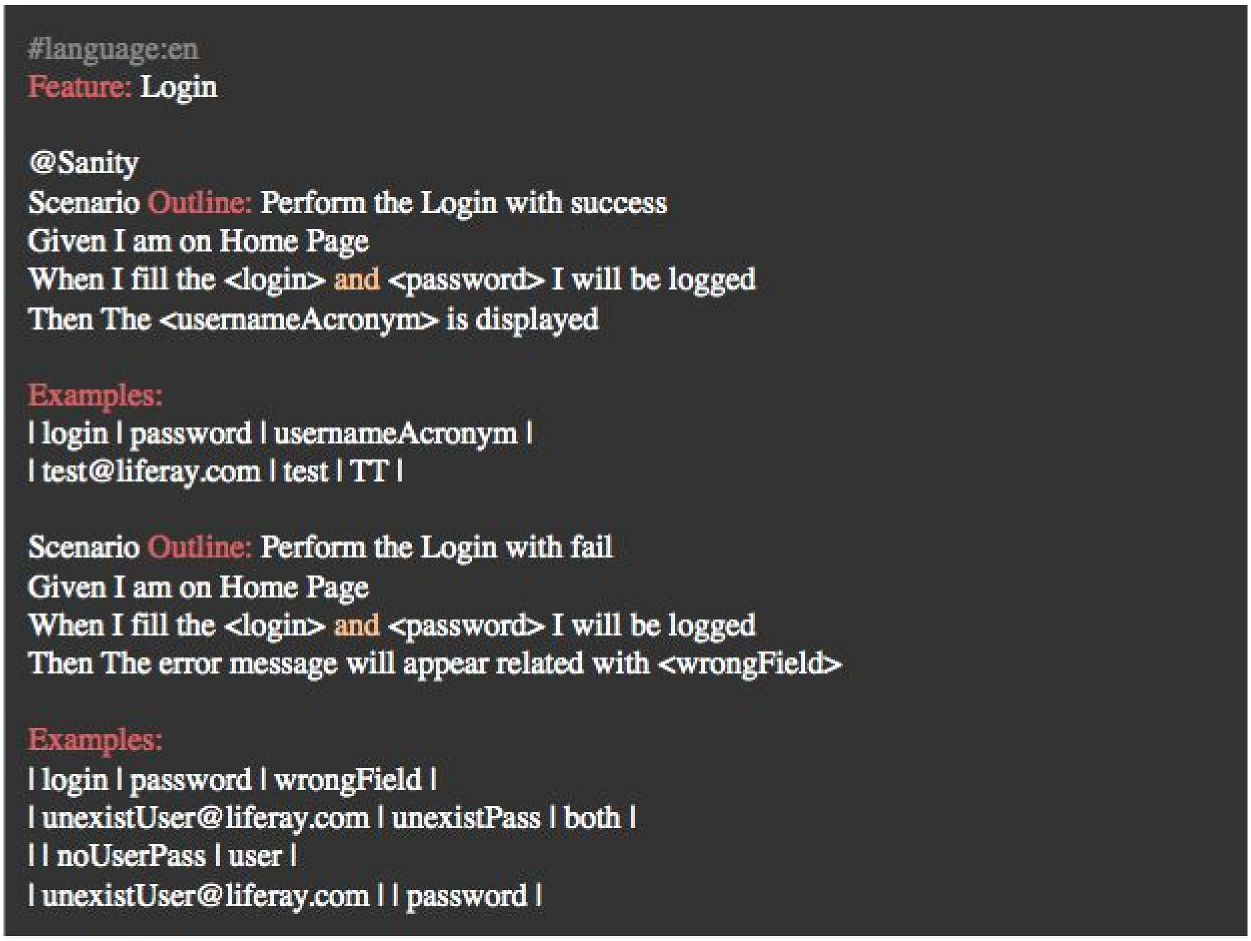
2. How it works

2.1. Essentials to have in place before implementing BDD

- Requirements should be converted into user stories that can define concrete examples.
- Each example should be a valid user scenario, rather than a mere test case.
- An understanding of the ‘role-feature-reason’ matrix and the ‘given-when-then’ formula.
- An awareness of the need to write ‘the specification of the behavior of a class’ rather than ‘the unit test of a class’.

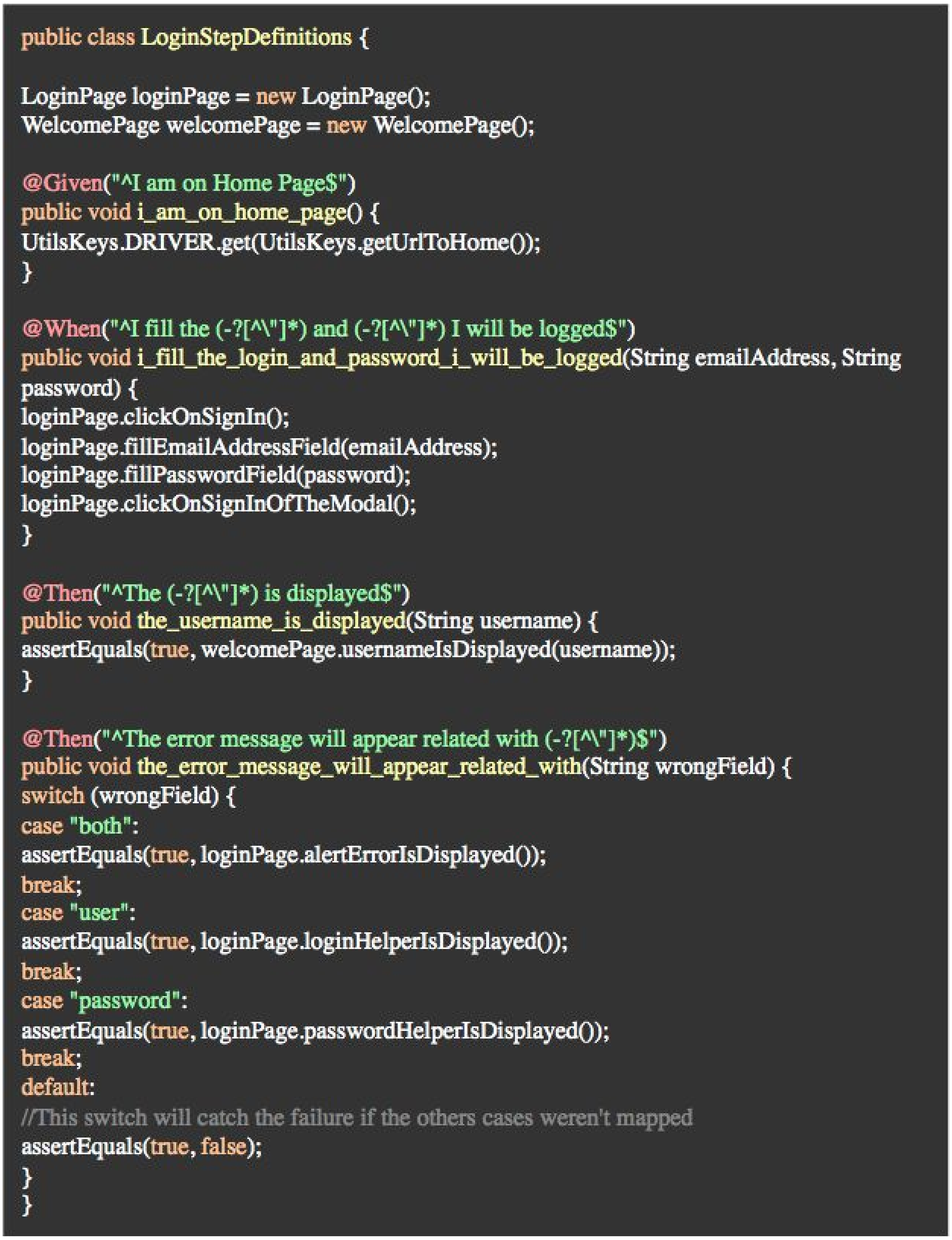
2.2. Code Structure

In your code you will have two kinds of files, the first is the ".feature" file and the second is the ".java" files. This files have a different functions on the code. The ".feature" will received the scenarios related with the feature that team are developing, example of one ".feature" below:



```
#language:en
Feature: Login

@Sanity
Scenario Outline: Perform the Login with success
Given I am on Home Page
When I fill the <login> and <password> I will be logged
Then The <usernameAcronym> is displayed

Examples:
| login | password | usernameAcronym |
| test@liferay.com | test | TT |

Scenario Outline: Perform the Login with fail
Given I am on Home Page
When I fill the <login> and <password> I will be logged
Then The error message will appear related with <wrongField>

Examples:
| login | password | wrongField |
| unexistUser@liferay.com | unexistPass | both |
| | noUserPass | user |
| unexistUser@liferay.com | | password |
```

The ".java" files will have the annotations of the file above to make the translation of the "cucumber language" to "selenium" or "java language". Example of one ".java" below:



```
public class LoginStepDefinitions {

LoginPage loginPage = new LoginPage();
WelcomePage welcomePage = new WelcomePage();

@Given("^I am on Home Page$")
public void i_am_on_home_page() {
UtilsKeys.DRIVER.get(UtilsKeys.getUrlToHome());
}

@When("^I fill the (-?[^\"]*) and (-?[^\"]*) I will be logged$")
public void i_fill_the_login_and_password_i_will_be_logged(String emailAddress, String
password) {
loginPage.clickOnSignIn();
loginPage.fillEmailAddressField(emailAddress);
loginPage.fillPasswordField(password);
loginPage.clickOnSignInOfTheModal();
}

@Then("^The (-?[^\"]*) is displayed$")
public void the_username_is_displayed(String username) {
assertEquals(true, welcomePage.usernameIsDisplayed(username));
}

@Then("^The error message will appear related with (-?[^\"]*)$")
public void the_error_message_will_appear_related_with(String wrongField) {
switch (wrongField) {
case "both":
assertEquals(true, loginPage.alertErrorIsDisplayed());
break;
case "user":
assertEquals(true, loginPage.loginHelperIsDisplayed());
break;
case "password":
assertEquals(true, loginPage.passwordHelperIsDisplayed());
break;
default:
//This switch will catch the failure if the others cases weren't mapped
assertEquals(true, false);
}
}
```

Using this approach any member of the team can contribute with the quality of the code, because anyone can write the ".feature" file, and other technical member of the team can create the annotations related with the action of the respective annotation.

References

- It was used a template project made by GS-BR. Template Project
- Know more on: https://cucumber.io

2.3. Page Object Pattern Structure

This approach is one easy technique to keep your code more understandable and easier to make the maintenance. You should create two initial structures. The **src/test/java/** must have four folders/packages:

- **com.liferay..test.functional.pages:** This path contains all classes mapped between functions per pages, for example, the Login Page, should have all elements and methods that the test will need to perform the actions on the respective page, the same logic is applicable on Welcome Page.

- **com.liferay..test.functional.steps:** This path have all the steps definitions, in other words, is the path who have the annotations mentioned on topic above. • com.liferay..test.functional.utils: This path have all the things that can be util to the code, for example the class "CommonMethods".

- **com.liferay..test.functional:** This path have only the "run" class.

- The **src/test/resources/** must have only the ".feature" files.

In this kind of organization will be easier to developer or tester, developing the code and looking for the future, realize the maintenance.

2.4. F.I.R.S.T Principles for Writing Good Unit Tests

The idea of F.I.R.S.T Principles is that can help make your tests shine and ensure that they pay off more than they cost.

Acronym FIRST stand for below test features:

- [F]ast
- [I]solated
- [R]epeatable
- [S]elf-validating
- [T]imely

Please, check this LINK more information.

2.5. LFRGS Selenium Commons Setup (How to prepare your environment)

Gradle’s tasks should be executed to perform your setups. All informations and steps you can find on README.md from your project (.../modules/test), such as:

- which tasks you need to run (to create a functional tests properties, create a environment configurations, download and automatically setting webDrivers: Firefox Driver and Chrome Driver Headless, etc);

2.6. **Common Methods**

2.6.1. FRW Selenium Commons

About the shared classes in "Util Class" section, you can know this information, on this link: Existing Classes on frw-selenium-commons

2.7. Why would you use ID attributes

- Clean automation code. Your automation code will be more readable if you get the attribute's locator by ID instead of getting it by XPath or CSS selectors.

- In some cases, it's inevitable to find an element by ID, but if you create a XPath with the ID element, this XPath will be more robust.
- Fastest way to locate elements on page because selenium gets it down tóexecuting document.getElementById().
- Fragility in UI tests is hard to manage, so other issues might also need to be resolved before you see the benefits of adding IDs. This point can achieve everybody's confidence in the team about the automated tests.

- Best discussions and articles about this subject can be found here:
    - Good practices for thinking in ID and Class name by Google.
        - Google's HTML code has ID's with some_id and also with some-id. I believe that the two one is correct. It must be chosen one way to follow and make the code readable.
    - Is adding ids to everything standard practice when using selenium
    - Which is the best and fastest way to find the element using webdriver by xpath

3. Auxiliary tool during test automation process

3.1. Firebug with FirePath - a great help when you need to find XPath expressions, he creates one for you by inspecting the element, work only on Firefox version 50.