



Test Automation

Leonardo Araujo Santos
Felipe Carvalho

29/05/2017

 **Behavior Driven Development - BDD**

 **JBehave**

 **REST API Testing**

 **Front-End Testing**



Behavior Driven Development



A Little of Context ...

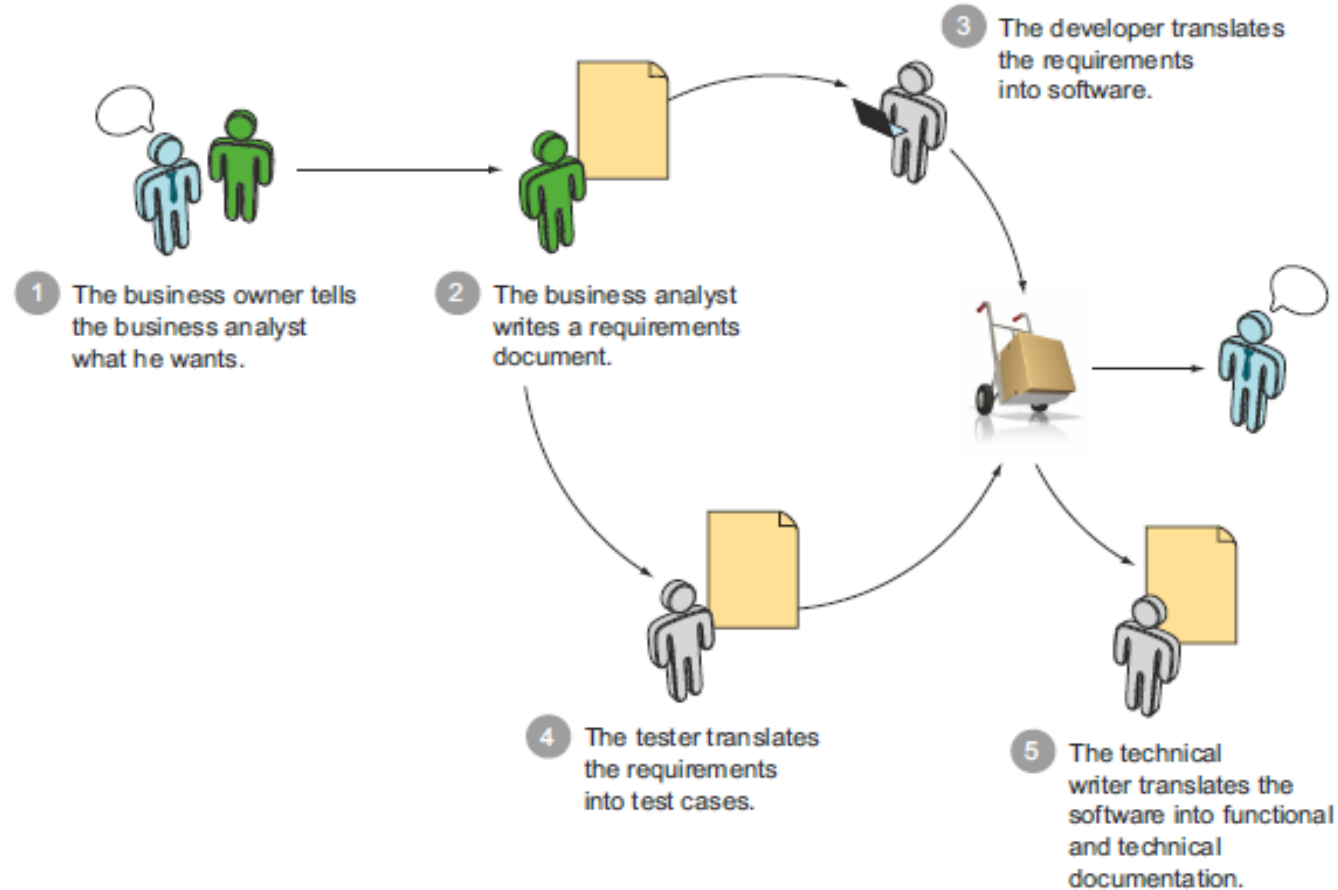


Requirements: Classic Methodologies vs Agile Methodologies

Classic Methodologies

- Requirement Analysis as a discipline
 - Aims to know and understand the customer needs
 - Activities: Discover, Analyse and document the requirements
 - Comprehensive Documentation
 - Techniques: Interviews with the stakeholders, requirement workshops, lists, prototypes, business use cases

Classic Methodologies



Issues

- Users do not know exactly what they really want
- Attempt to adapt a new system to a common standard or model instead of building something that meets the **customer needs**
- The **communication** tends to be slow
- The customer is involved usually at the beginning and at the end of the development process
- Business users and Technical staff have a **different vocabulary**

Agile: Software that matters!

“Customer satisfaction through early and continuous software delivery”

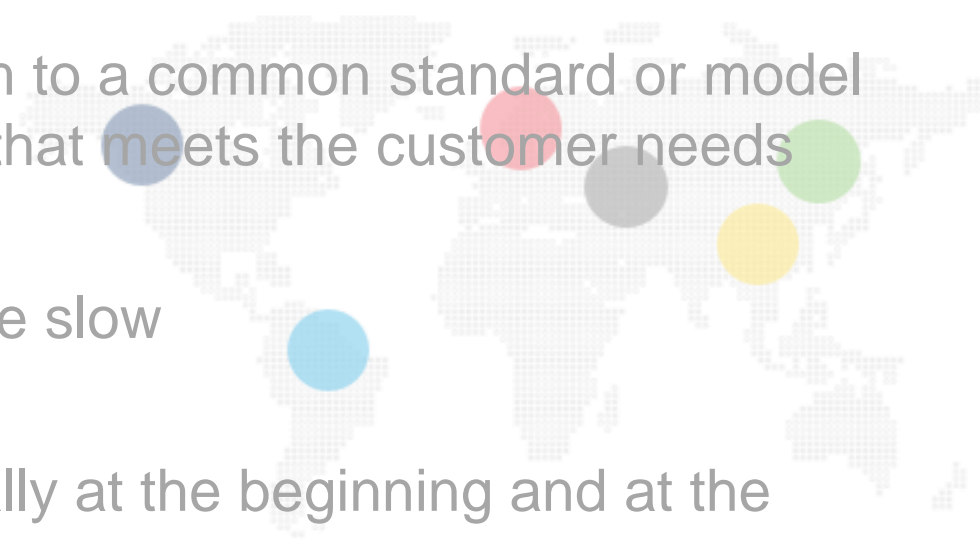
(Agile Manifesto, 2001)



Agile Methodologies

- Requirements
 - Meetings for scope definition (high-level approach)
 - User Stories
 - Requirement changes are normal and acceptable
 - Describe just the enough to develop the requirement and check often with the customer if it is right
 - Constant feedback from the customers
 - Deliver value to the customers, deliver software that really matters.

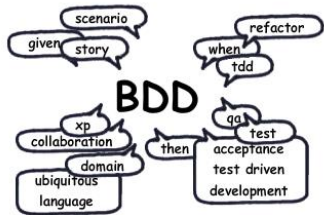
Issues

- Attempt to adapt a new system to a common standard or model instead of building something that meets the customer needs
 - The communication tends to be slow
 - The customer is involved usually at the beginning and at the end of the development process
 - Business users and Technical staff have a **different vocabulary**
- 

Behavior Driven Development

“You can't write good software if you don't understand what it's supposed to do.”

John Ferguson Smart - BDD In Action



So what is BDD?

- BDD stands for *Behavior Driven Development*
- It is a set of agile practices and techniques that aim the enhancement of the **collaboration** and **communication** in the software development teams
- Focus on the language and interactions during the software development process

What BDD is not ...

- BDD is not a software development methodology
- It is not a substitute for Scrum, XP, Kanban, RUP or any methodology / framework you are using for developing your software

History

- Created in 2003 by Dan North, consultant specialist in agile methodologies (ThoughtWorks)
- It came up from observations and ideas from agile practices such as Scrum, Lean, TDD and DDD
- Initially, it was a response to TDD technique (which focus on the technical aspect of the software)

BDD Benefits

- Encourage the team to develop, use and share a **common understanding** of how and why the application should work
- Define features that focus your development efforts on underlying business goals (software that matters!)

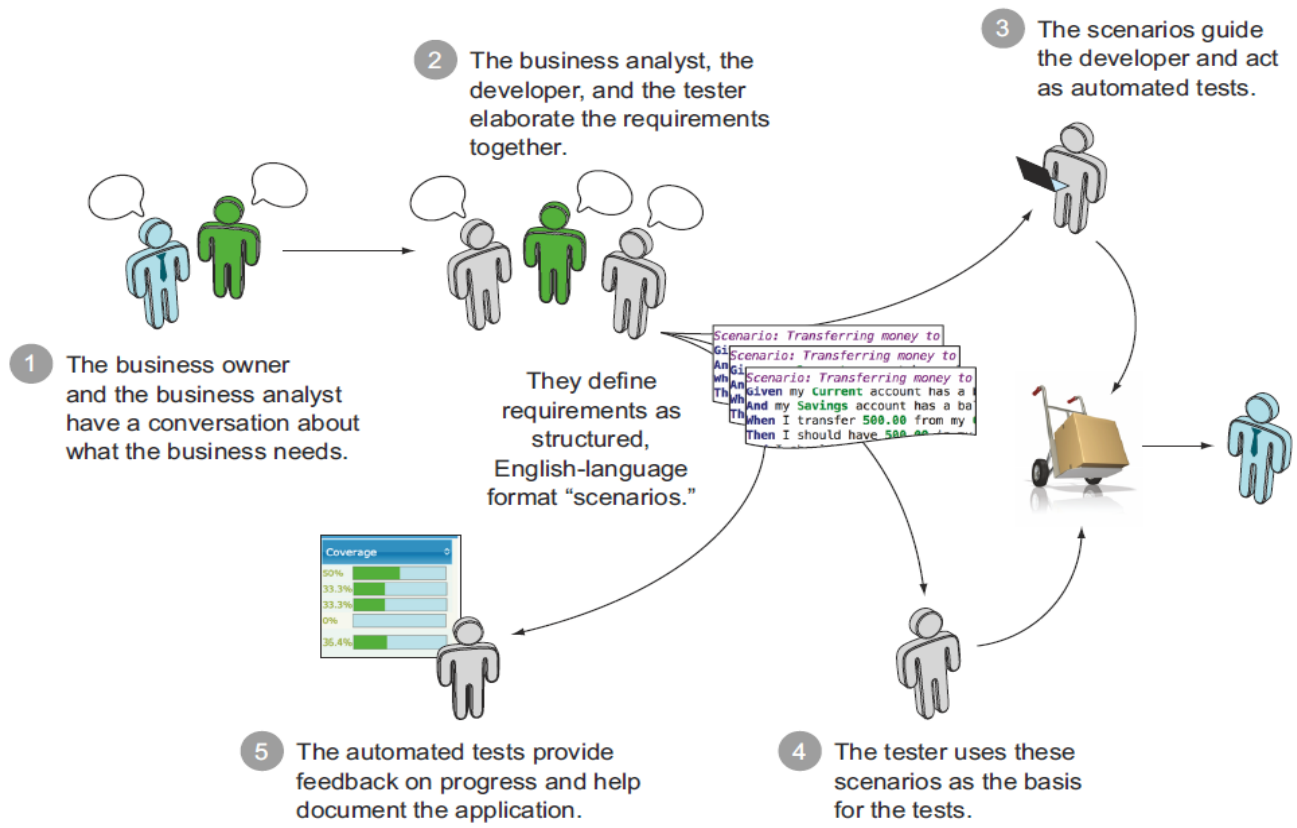
BDD Benefits

- Enhancement of the language and the interactions during the development process
- It allows the developers to understand better the reasons the code should be created and not just the technical details
- It reduces the constant translations between the business language and technical language, everyone should be in the same page (users, developers, testers, PO, Managers)

BDD Techniques

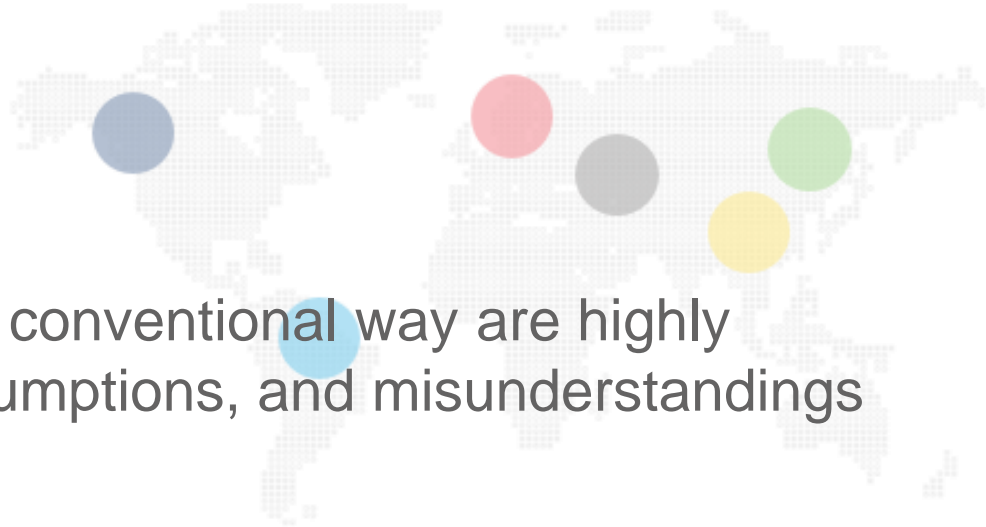
- To achieve this, BDD introduces some techniques that should be integrated with the development process:
 - Specification using scenarios/examples
 - Usage of a Ubiquitous Language
 - Usage of “should” and “ensureThat” to describe behavior
 - Automated Examples for fast feedback (acceptance tests)
 - Outside-In Development

BDD Workflow



Specification Using Examples

Requirements written in a conventional way are highly susceptible to ambiguities, assumptions, and misunderstandings



Specification Using Examples

- Write acceptance criterias (AC) for the User Stories using scenarios and examples to describe the expected feature behavior
- Establishes in a clear way what should be developed and tested
- Extremely effective way to **communicate requirements** in a precise, clear and unambiguous manner

Specification Using Examples

Story: ATM Withdraw

Narrative:

In order to get money from my banking account

As a customer

I want to perform a withdraw

Scenario: The withdraw value should be debited from the customer's current balance

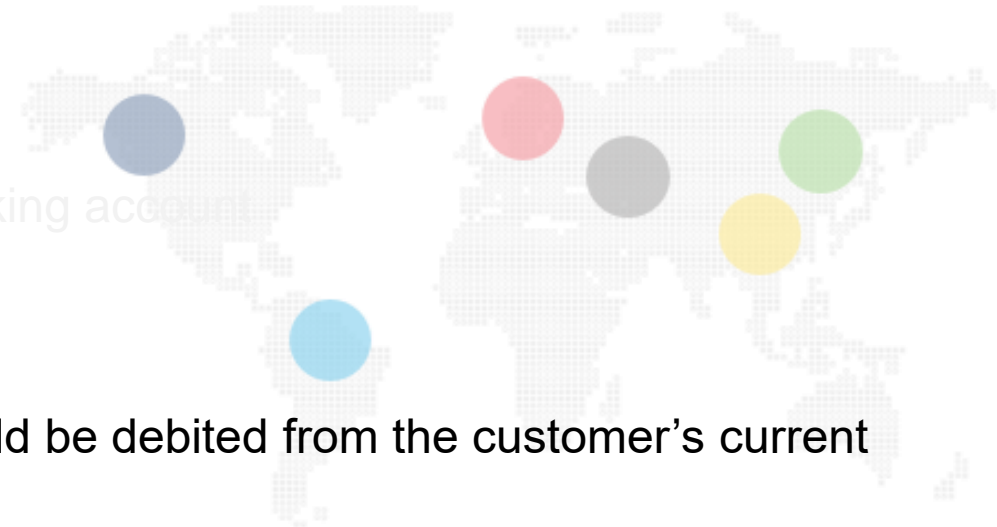
Given an active customer with the id 9999

And his account has a balance of \$2000.00

When the customer withdraw \$500.00

Then the withdraw should be completed successfully

And the new balance should be \$1500.00



Specification Using Examples

- Keywords

Story (Feature)

As a
I want
so that

Scenario (Acceptance Criteria)

Given
When
Then



Specification Using Examples

- Its strength is that it forces you to identify the value of delivering a story when you first define it. When there is no real business value for a story, it often comes down to something like "... I want [some feature] so that [I just do, ok?]." This can make it easier to descope some of the more esoteric requirements. (Dan North)
- A story's behaviour is simply its acceptance criteria – if the system fulfills all the acceptance criteria, it's behaving correctly; if it doesn't, it isn't.

The “Should” word

- It encourages to question whether the behavior described is appropriate, or whether they can be moved or removed entirely

Then the withdraw **should** be completed successfully
And the new balance should be \$1500.00

Ubiquitous Language

- It aims to define a shared domain language that can be understood either by the customers and developers, so they can talk in the same language
- Eric Evans published his bestselling book Domain-Driven Design that describes the concept of modeling a system using a ubiquitous language based on the business domain, so that the business vocabulary permeates right into the codebase.
- So Dan North realized that BDD provides an Ubiquitous Language for the analysis process itself

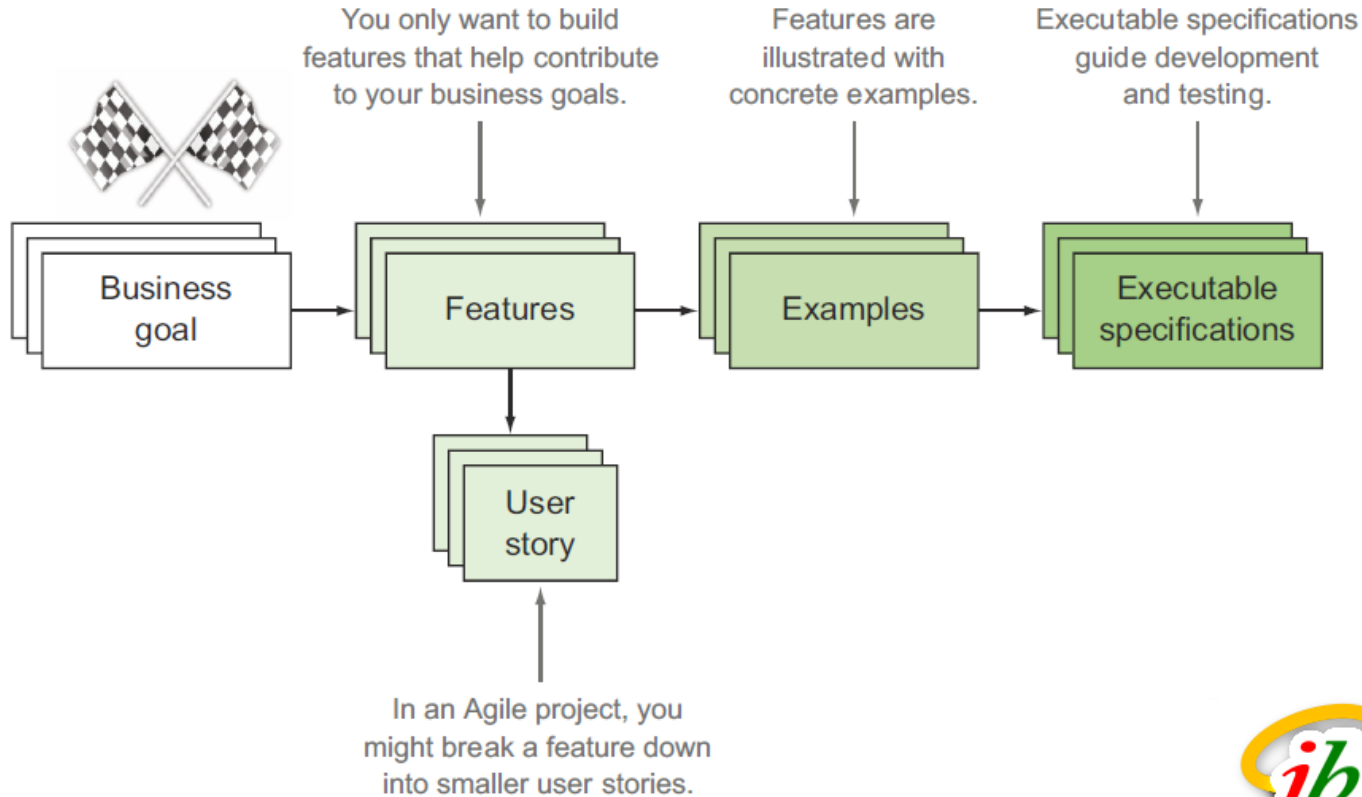
Outside-In Development

- BDD is guided by the business values. These values are noticed by the users through the user interface (usually, but not always the GUI)
- All features should be developed from the interface (UI) perspective and then evolved to the other application modules (services, domain, database)
- Once having the UI in place, all the other application modules can be developed as its clients

Automated Examples

- In BDD the acceptance criteria can be easily automated from the stories and scenarios
- Provides Live Documentation, regression tests and fast feedback on the quality (CI)
- Tools like Jbehave, Cucumber, Serenity BDD, Rspec (Ruby), Jasmine (JavaScript) can be used to easily automate them and allow the specification to be executed

Automated Examples



JBehave



What is JBehave?

- JBehave is an *open-source* **JAVA** based **Framework** for **Behavior Driven Development** (BDD). It intends to make the BDD more accessible and intuitive, shifting the vocabulary from test-based to behavior-based.

What is JBehave?

Writing Stories 1. Write story

Plain text

Scenario: A trader is alerted of status

Given a stock and a threshold of 15.0
When stock is traded at 5.0
Then the alert status should be OFF
When stock is traded at 16.0
Then the alert status should be ON

2. Map steps to Java

POJO

```
public class TraderSteps {  
    Mapping Stories TradingService service; // Injected  
    private Stock stock; // Created  
  
    @Given("a stock and a threshold of $threshold")  
    public void aStock(double threshold) {  
        stock = service.newStock("STK", threshold);  
    }  
    @When("the stock is traded at price $price")  
    public void theStockIsTraded(double price) {  
        stock.tradeAt(price);  
    }  
    @Then("the alert status is $status")  
    public void theAlertStatusIs(String status) {  
        assertThat(stock.getStatus().name(), equalTo(status));  
    }  
}
```

3. Configure Stories

Only once

```
public class TraderStories extends JUnitStories {  
    public Configuration configuration() {  
        return new MapStoryConfiguration();  
        useStoryLoader(new LoadFromClasspath(this.getClass()));  
        useStoryReporterBuilder(new StoryReporterBuilder()  
            .withCodeLocationCodeLocationFromClass(this.getClass()));  
        withFormats(CONSOLE, TXT, HTML, XML);  
    }  
  
    public Configuring Stories candidateSteps() {  
        return new AnnotatedStepsFactory(configuration(),  
            new TraderSteps(new TradingService()));  
    }  
  
    protected List<String> storyPaths() {  
        return new StoryFinder().findPaths(CodeLocationFromClass(this.getClass()),  
            "**/*-story");  
    }  
}
```

4. Run Stories

With any of



5. View Reports

HTML

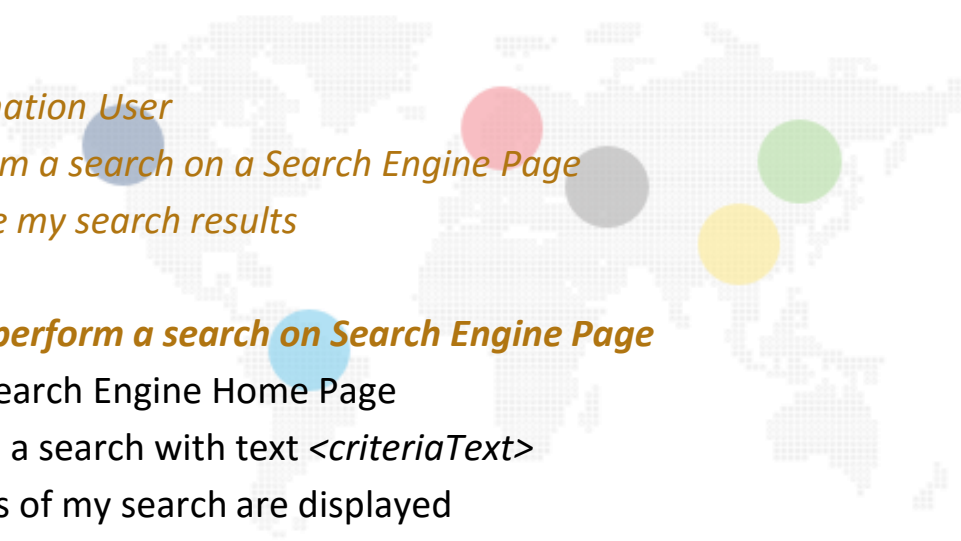
Scenario: A trader is alerted of status

Given a stock and a threshold of 15.0
When stock is traded at 5.0
Then the alert status is OFF
When stock is traded at 16.0
Then the alert status is ON

Writing Stories Using JBehave

- Based on BDD, each **story** should be broken up in scenarios.
- A **story** is a collection of **scenarios**, and these **scenarios** consist of a collection of **steps**.
- Stories should be written using the business language, and avoid as much as possible to use technical details.

Story File Structure



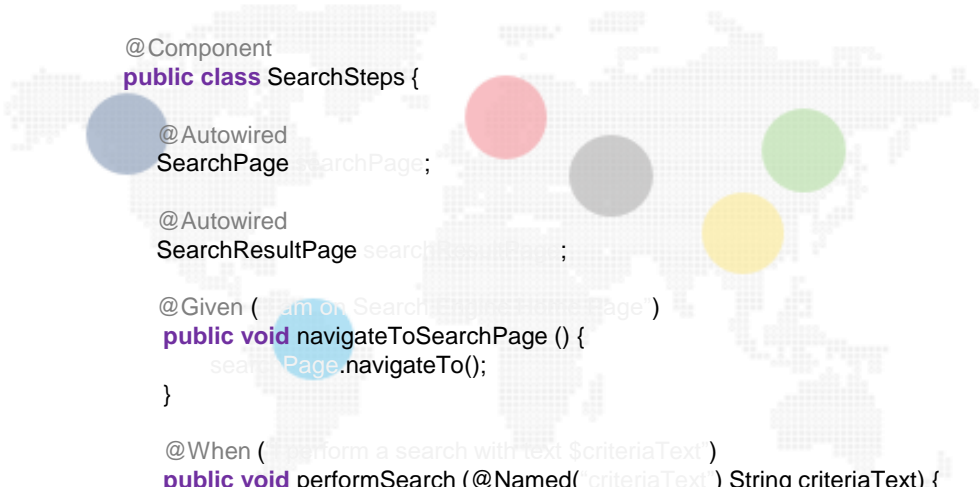
Narrative	<div data-bbox="656 185 830 229">Narrative:</div> <div data-bbox="656 240 1101 283"><i>As a Test Automation User</i></div> <div data-bbox="656 294 1506 338"><i>I want to perform a search on a Search Engine Page</i></div> <div data-bbox="656 349 1217 393"><i>So that I can see my search results</i></div>
Scenario	<div data-bbox="656 469 1586 513">Scenario: <i>I can perform a search on Search Engine Page</i></div> <div data-bbox="656 524 1313 567">Given I am on Search Engine Home Page</div> <div data-bbox="656 578 1449 622">When I perform a search with text <i><criteriaText></i></div> <div data-bbox="656 633 1371 677">Then The results of my search are displayed</div> <div data-bbox="656 687 830 731">Examples:</div> <div data-bbox="656 742 966 786"> <i>criteriaText</i> </div> <div data-bbox="656 797 966 840"> <i>Test Automation</i> </div>

Story File Structure

- **Narrative:** Gives a brief description of the story, what operation(s) is/are going to be tested.
- **Scenario:**
 - **Given:** Pre-conditions expected for the scenario execution.
 - **When:** Interactions on the application, may be a user interaction or even a system internal interaction, normally is when function or features are executed through the system.
 - **Then:** Test validations. Here you will verify if the application have worked as expected.

Step Definition

- JBehave centers around the matching of textual **steps** with **Java methods** contained in steps instances.
- Each annotated method in the steps instances corresponds to a **StepCandidate**, which is responsible for the matching and for the creation of an executable step.



```
@Component
public class SearchSteps {

    @Autowired
    SearchPage searchPage;

    @Autowired
    SearchResultPage searchResultPage;

    @Given ( "I am on Search Page" )
    public void navigateToSearchPage () {
        searchPage.navigateTo();
    }

    @When ( "I perform a search with text $criteriaText" )
    public void performSearch ( @Named( "criteriaText" ) String criteriaText ) {
        searchPage.typeSearchCriteria( criteriaText );
        searchPage.performSearch();
    }

    @Then ( "The results of my search are displayed" )
    public void checkSearchResults () {
        Assert.assertTrue( searchResultPage.getResults() > 0 );
    }

}
```

Step Parameters

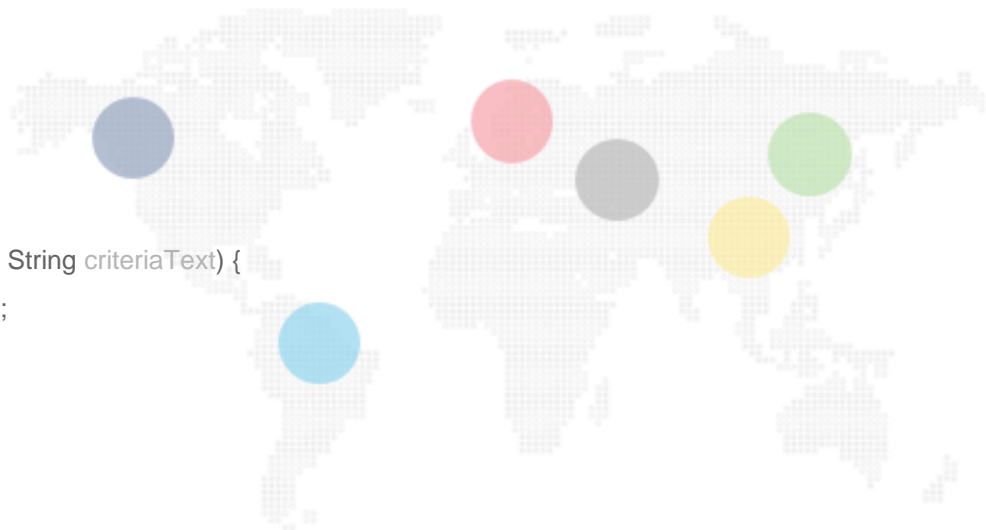
- Standalone parameter:

- Java Step definition

```
@When("I perform a search with text $criteriaText")
public void performSearch(@Named("criteriaText") String criteriaText) {
    searchPage.typeSearchCriteria(criteriaText);
    searchPage.performSearch();
}
```

- Story file step call

When I perform a search with text *Test Automation*



Step Parameters

- Examples Table
 - Java Step definition

```
@When("I perform a search with text <criteriaText>")
public void performSearch(@Named("criteriaText") String criteriaText) {
    searchPage.getSearchComponent().typeSearchCriteria(criteriaText);
    searchPage.getSearchComponent().performSearch();
}
```

- Story file step call

When I perform a search with text *<searchCriteria>*

Examples:

```
|searchCriteria    |
|Test Automation   |
|Software Engineer|
```



REST API Testing



REST API Testing

- JBehave just provide us the BDD/DSL semantic
- What do you need?
 - REST Client (Apache HttpClient, RestTemplate, etc)
 - JSON/XML Parser (Jackson, Gson, JAX-B)
 - For a better OO Design some Helper classes:
 - Test Service Classes
 - DTO/ Domain Objects
 - Context
 - Lifecycle

Test Service Objects

- Plain Java objects that help to decouple service calls from steps or other test related classes. It isolates the service logic from other test classes.

Test Service Objects

```
@Component
public class AccountTestService {

    @Value("${sbservices.endpoint}")
    private String SB_API_ENDPOINT;

    @Autowired
    private TestExecutionContext context;

    @Autowired
    private RestTemplate restTemplate;

    private static final String ACCOUNT_CREATE_URL = "accounts/?ownerCpf={ownerCpf}";

    public ResponseDTO<Account> createAccount(String ownerCpf) {
        ...

        ResponseEntity<ResponseDTO<Account>> response = restTemplate.exchange(
            uri, HttpMethod.POST, httpEntity, typeRef, params);

        responseDTO = response.getBody();

        if (responseDTO.getData() != null) {
            Account account = responseDTO.getData();
            context.registerCreatedAccount(account);
        }

        context.setResponseDTO(responseDTO);
        return responseDTO;
    }
}
```

DTO/Domain Objects

- A data transfer object is an object that carries data between processes.
- A DTO object aggregates the data that would have been transferred by the several calls,

```
public class Account {  
  
    private String ownerCpf;  
    private BigDecimal balance;  
  
    public void Account (String ownerCpf) {  
        this.ownerCpf = ownerCpf;  
        this.balance = BigDecimal.ZERO.setScale(2, RoundingMode.HALF_EVEN);  
    }  
  
    public String getOwnerCpf () {  
        return cpfOwner;  
    }  
  
    public void setBalance (BigDecimal value)  
        balance = balance.add(value);  
    }  
  
    public BigDecimal getBalance (BigDecimal value)  
        return balance ;  
    }  
}
```

Context

- Plain Java objects that help to store data that need to be shared between steps or steps classes.
- They store information during the test runtime and normally are cleaned after scenario/story.

```
@Component
public class TestExecutionContext {

    private ResponseDTO<?> lastResponse;
    private List<Account> createdAccounts = new ArrayList<>();

    public void registerCreatedAccount(Account account) {
        createdAccounts.add(account);
    }

    public void removeCreatedAccount(Account account) {
        createdAccounts.remove(account);
    }

    public List<Account> getCreatedAccounts() {
        return createdAccounts;
    }

    public ResponseDTO<?> getLastResponse() {
        return this.lastResponse;
    }

    public void setResponseDTO(ResponseDTO<?> responseDTO) {
        this.lastResponse = responseDTO;
    }
}
```

Jbehave Lifecycle

- @BeforeScenario
- @AfterScenario
- @BeforeStory
- @AfterStory
- @BeforeStories
- @AfterStories

```
public class TestLifecycle {

    @BeforeScenario(uponType = ScenarioType.ANY)
    public void setupScenario() {
        System.out.println("# BEFORE SCENARIO #");
    }

    @AfterScenario(uponType = ScenarioType.ANY, uponOutcome = Outcome.ANY)
    public void tearDownScenario() {
        System.out.println("# AFTER SCENARIO #");
    }

    @BeforeStory
    public void setupStory() {
        System.out.println("# BEFORE STORY #");
    }

    @AfterStory
    public void tearDownStory() {
        System.out.println("# AFTER STORY #");
    }

    @BeforeStories
    public void setupStories() {
        System.out.println("# BEFORE STORIES #");
    }

    @AfterStories
    public void tearDownStories() {
        System.out.println("# AFTER STORIES #");
    }

}
```

Jbehave Lifecycle

- ScenarioType:
 - **Normal:** Scenarios that don't have Examples Table.
 - **Example:** Scenarios that have Examples Table.
 - **Any:** Any of above scenario types.
- Outcome:
 - **Success:** Executed if the scenario has successfully executed.
 - **Failure:** Executed if the scenario has failed.
 - **Any:** Any of above outcomes.

Exercise 1

- Create a scenario in Account Transactions Story to test the bank's transfer service.
- The service URI is:
<https://sbsservice.herokuapp.com/api/accounts/{owner}/>
- The HTTP Method is PUT and takes two parameters:
 - targetCpf – Destination Account
 - value - Ammount to be transfered
- You should use the existent structure as much as possible

Exercise 1 - Tips

- You should have two existent accounts.
- One of the accounts should have enough balance to the transferring.
- The Transfer method does not exist yet in AccountTestService class, so you have to implement it.
- You should assert the balance of the both accounts after the transferring.
- After each scenario execution the created accounts are cleaned.

Exercise 2

- Write your own lifecycle class that logs a message in console indicating what is going on (e.g “Setup before story ...”).
 - Remember to register your lifecycle class as a bean in Spring application context

Exercise 3

- Create a new story to test the brand new Banks credit service.
- New service URL:
<https://sbsservice.herokuapp.com/api/accounts/credit/{ownerCpf}/loan?value={ammount}>
 - ownerCpf: CPF associated to account that will receive the money.
 - loan: Requested loan ammount.

Exercise 3

- This service allows the customer to get a loan following some credit rules. Then to get a loan the customer have to:
 - Have a minimum balance \$2000 in his account.
 - The customer can just get a loan of 30% of his actual balance.
 - The customer can only get the loan once.

Exercise 3 - Hints

- Write a new story to test this service since is related to a new credit service.
- Test Scenarios:
 - Customer is able to get a loan.
 - Customer does not have the minimum balance to get a loan.
 - Customer tries to get a loan that exceeds the allowed 30% of his actual balance ammout.
 - Customer already has a pending loan.

Exercise 3 - Tips

- The validation return codes and messages that you will need in the assertions can be find in the Enum **ResponseMessage.java**.
- Don't forget that the accounts created are cleaned after the scenario. In the **ScenarioLifecycle** class, take a look in the method annotated with `@AfterScenario`.

Front-End Testing



Front-end Testing

- For web front-end testing purposes, we will have to use a tool that can automate the user actions through the browser interface: Selenium

Selenium


- Tool to automate browser.
- Designed for testing purposes.
- Can be used to automate even web-based administration tasks.
- Has support for many browsers.



Selenium WebDriver API

- Collection of methods to interact with a browser.
- Drives a browser natively as a real user.
- Makes direct calls to the browser using each browser's native support for automation.
- Implementations:
 - ChromeDriver
 - FirefoxDriver
 - InternetExplorerDriver
 - ...

Selenium WebDriver API

- Using WebDriver you can:
 - Navigate
 - Search for HTML/DOM elements in the page using some criteria (css class, id, name, xpath, etc)
 - Get HTML elements attributes/properties
 - Trigger elements actions (type, click, double click, select, etc)
 - Manage the browser (delete cookies, set a proxy, etc)
- 

Selenium WebDriver API

- Example:

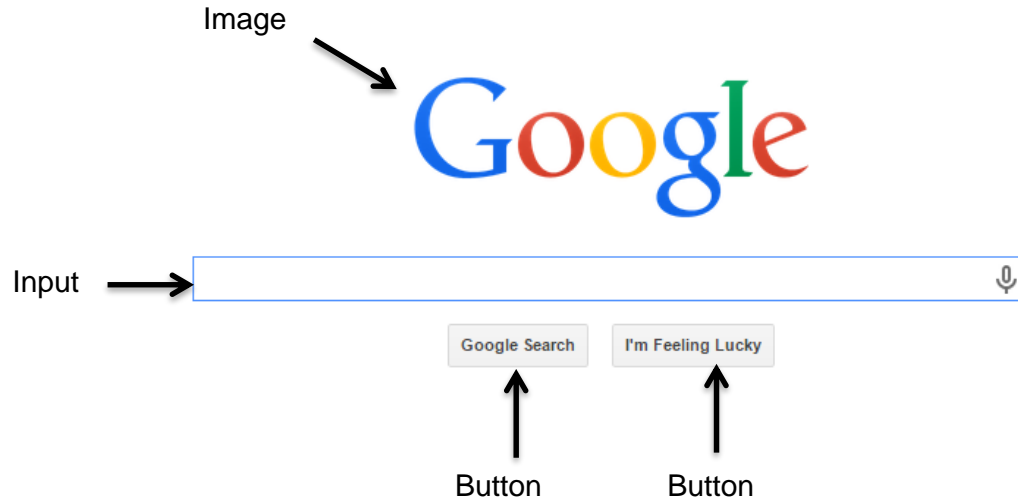
```
WebElement input = driver.findElement(By.tagName("input"));
input.sendKeys("Sample Test");
```

```
WebElement button = driver.findElement(By.className("gwt-button"));
if ( button.isDisplayed() && button.isEnabled() ) {
    button.click();
}
```

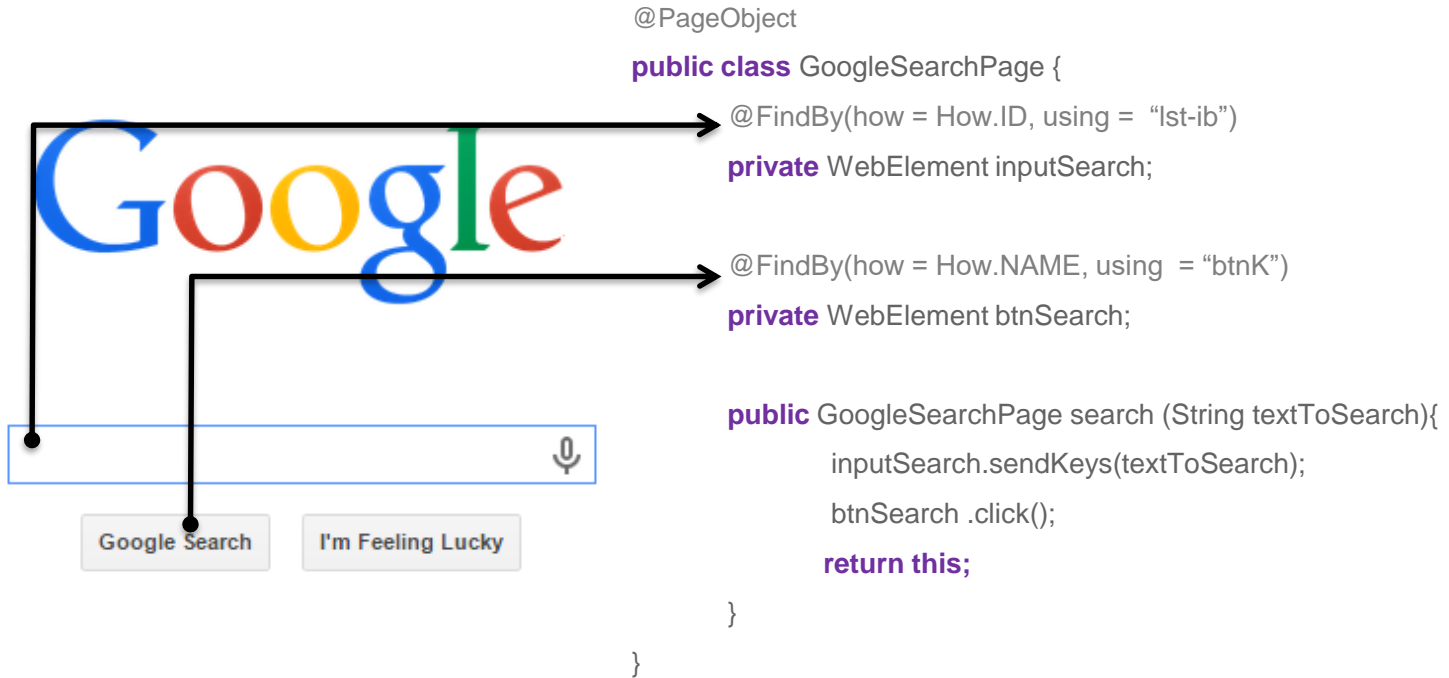
Page Objects

- Within your web app's UI there are areas that your tests interact with.
- A Page Object simply models these as objects within the test code.
- This reduces the amount of duplicated code and means that if the UI changes, the fix need only be applied in one place.

Page Objects



Page Objects



How to find Elements?



```
<div id="gs_lc0" style="position: relative;">
  <input class="gsfi" id="lst-ib" maxlength="2048" name=
    "q" autocomplete="off" title="Search" type="text" value
    aria-label="Search" aria-haspopup="false" role=
    "combobox" aria-autocomplete="both" dir="ltr"
    spellcheck="false" style="border: none; padding: 0px;
    margin: 0px; height: auto; width: 100%; position:
    absolute; z-index: 6; left: 0px; outline: none;
    background: url(data:image/gif;base64,R01GODlhAQABAID/
    AMDAwAAAACH5BAEAAAAALAAAAAIAAAEAAICRAEAOw%3D%3D)
    transparent;">
  <div class="gsfi" id="gs_sc0" style="color:
    transparent; padding: 0px; position: absolute; z-index:
    2; white-space: pre; visibility: hidden; background:
    transparent;">
  </div>
</div>
```

div div #sbtc div #sfddiv div #sb ifc0 #qs lc0 input#lst-ib.gsfi

@FindBy parameters:

how: ID

using: "lst-ib"

Exercise 4

- Following the best practices, now we have to change the FE test project to use PageObjects instead of making calls to the webdriver directly inside the test methods.
- Complete the scenario adding a step to test the logout link.

Exercise 4 - Tips

- You should change the existent scenario and create a PageObject to represent for instance, the index page.
- In your page object you will have methods to represent the possible actions in the page.



Leonardo Araujo Santos
Felipe Carvalho

29/05/2017