Behavioural Driven Development - BDD

Softworx Overview

- Canadian Distributor for Continuous Testing

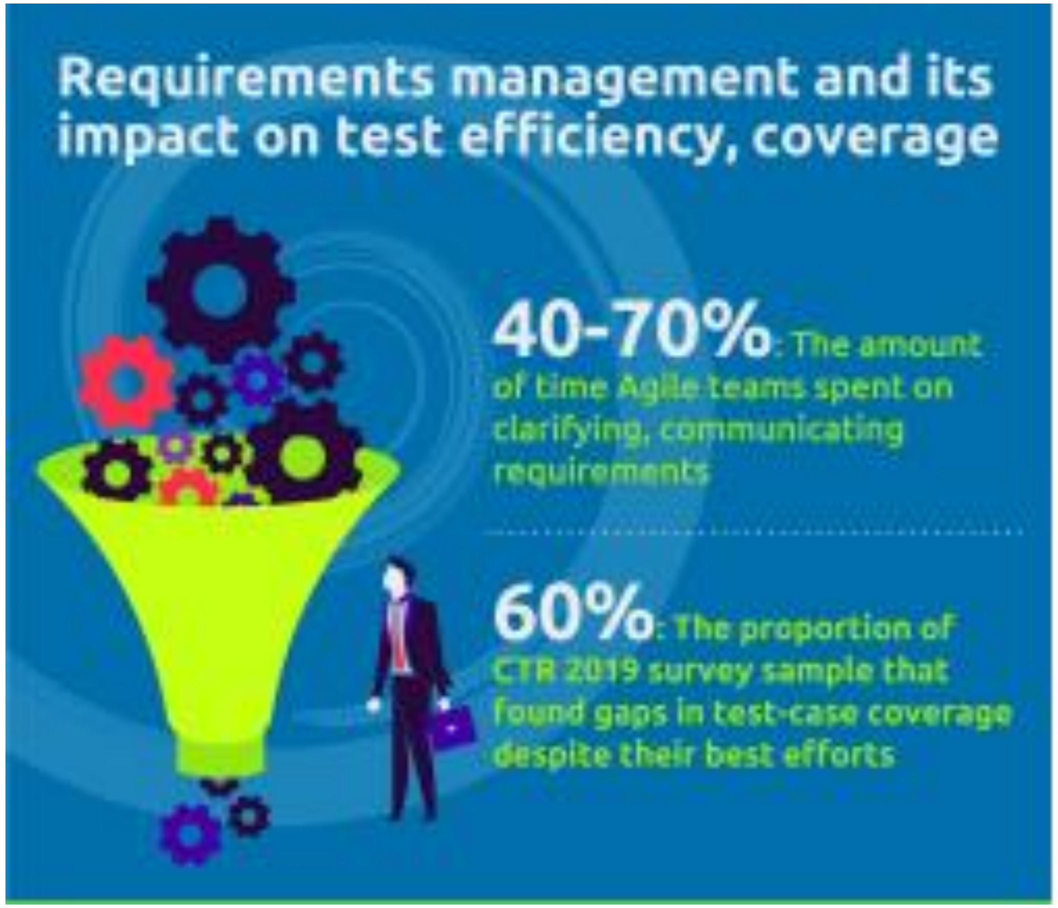
CONFORMIQ

Optimized Test Case & Test Script Generation

PARASOFT®
We make software work.™

Automated Code Analysis & Regression Testing
SOAtest; JTest; C++Test; dotTEST; DTP; Virtualize

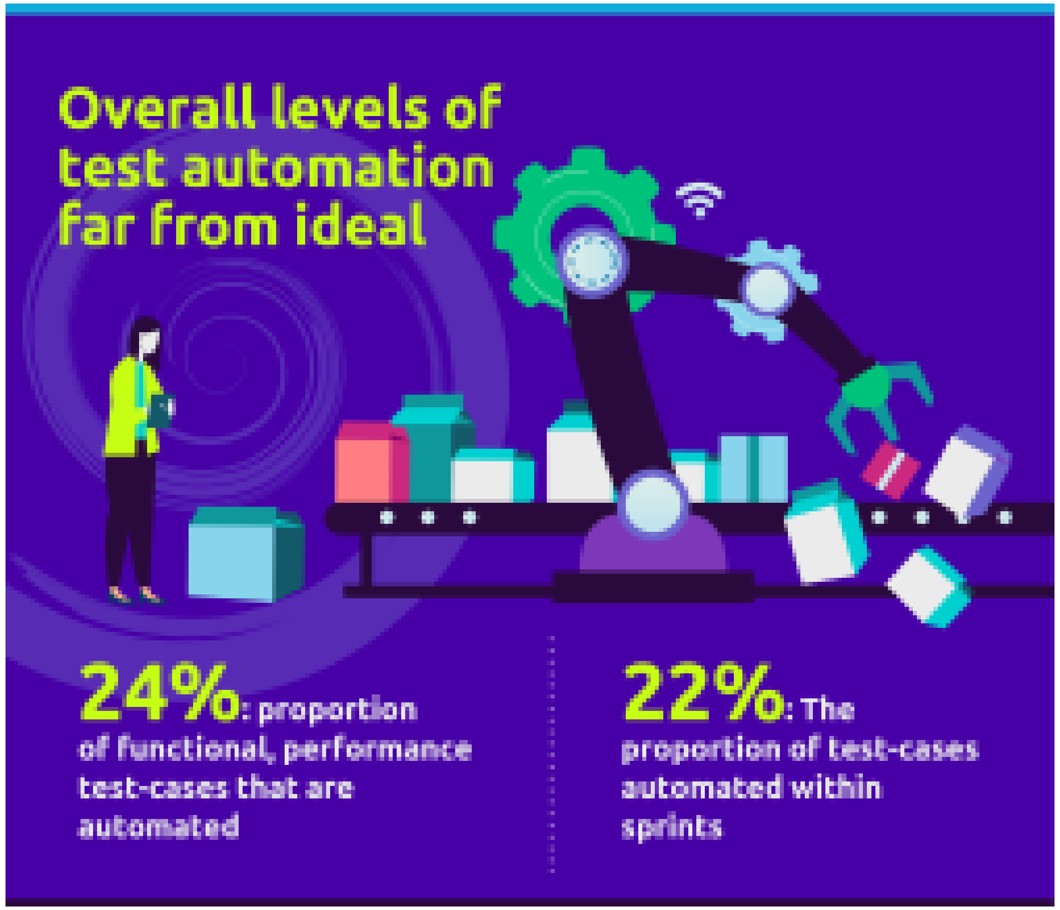
Requirements to Automation – What is the reality!



Requirements management and its impact on test efficiency, coverage

40-70%: The amount of time Agile teams spent on clarifying, communicating requirements

60%: The proportion of CTR 2019 survey sample that found gaps in test-case coverage despite their best efforts



Overall levels of test automation far from ideal

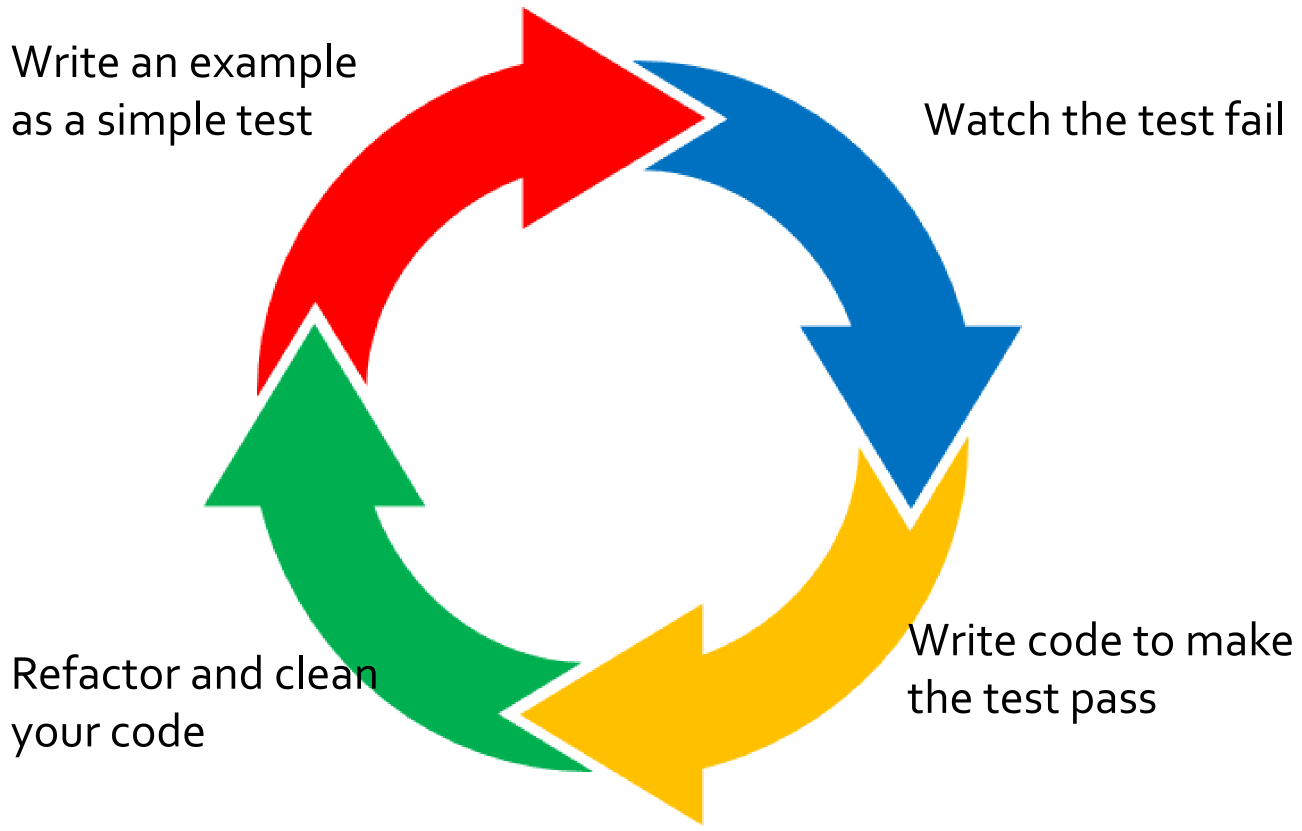
24%: proportion of functional, performance test-cases that are automated

22%: The proportion of test-cases automated within sprints

Requirements to Automation execution – So why the problem?

- Communication – different voices –> different interpretations
- Doing requirements / development / testing in silos
- We always do it this way -> We have too much invested in the old way
- We aren't waterfall and we aren't agile – we'll change when we get to agile
- We have too many other technologies and applications and environments –> How will this work with my existing stack
- This is going to be too hard to learn and implement
- We have to still do manual testing

Let’s solve it with Test Driven Development (TDD)


Write an example as a simple test
Watch the test fail
Write code to make the test pass
Refactor and clean your code

Focuses on driving the development of individual functions in the code

TDD works satisfactorily, as long as the business owner is familiar with the unit test framework being used and their technical skills are strong enough, which is not always the case.

Nice Try but let's move to BDD – what's that?

- Behavioral Driven Development (**BDD**) is a software development approach that has evolved from TDD (Test Driven Development).
- It differs by being written in a shared language, which improves communication between tech and non-tech teams and stakeholders.
- **"BDD practitioners explore, discover, define, then drive out the desired behaviour of software using conversations, concrete examples, and automated tests."**
- **We collaborate, we record that collaboration in some form of specification, and then we automate that specification to drive out the implementation.**
  - BDD is a 3-level approach.
  - The first is to get developers, testers and people from the business to talk to each other. That is the beginning of BDD. Anyone who thinks "we're doing BDD because we use 'Given', 'When', 'Then'" often aren't. 'Given', 'When', 'Then' has nothing to do with BDD. BDD stands for Behaviour-Driven Development and the real intent is to try and work out what your customer or business wants from the software before you start working on it. The first way of doing this is to actually **collaborate** with those people.
  - The second thing, once we've got that collaboration, is to somehow **record**that in a way that is meaningful to anybody reviewing it, who might come around and look at it later, who might want to comment on it. Typically, that gets done using a ubiquitous language. People often use the 'Given', 'When', 'Then' words but that's not necessary. The idea is that we've collaborated and this is shared understanding. This has made sure there is a collaborative goal that we're trying to achieve and once we've got good at collaborating, it's worth trying to capture that so that not everybody needs to be in the room at the same time, so that that shared learning can be propagated.
  - Finally, if it's appropriate to our teams and to our projects, we **automate** our tests to drive the behaviour.
- **There is no mention of Cucumber anywhere. Cucumber is not part of BDD. Cucumber is something that has been created to help people automate in a specific way.**

Similar process to TDD but at the level of features .....

Features are driven by user needs .....

User needs are typically described in an natural language

OK so why BDD over TDD?

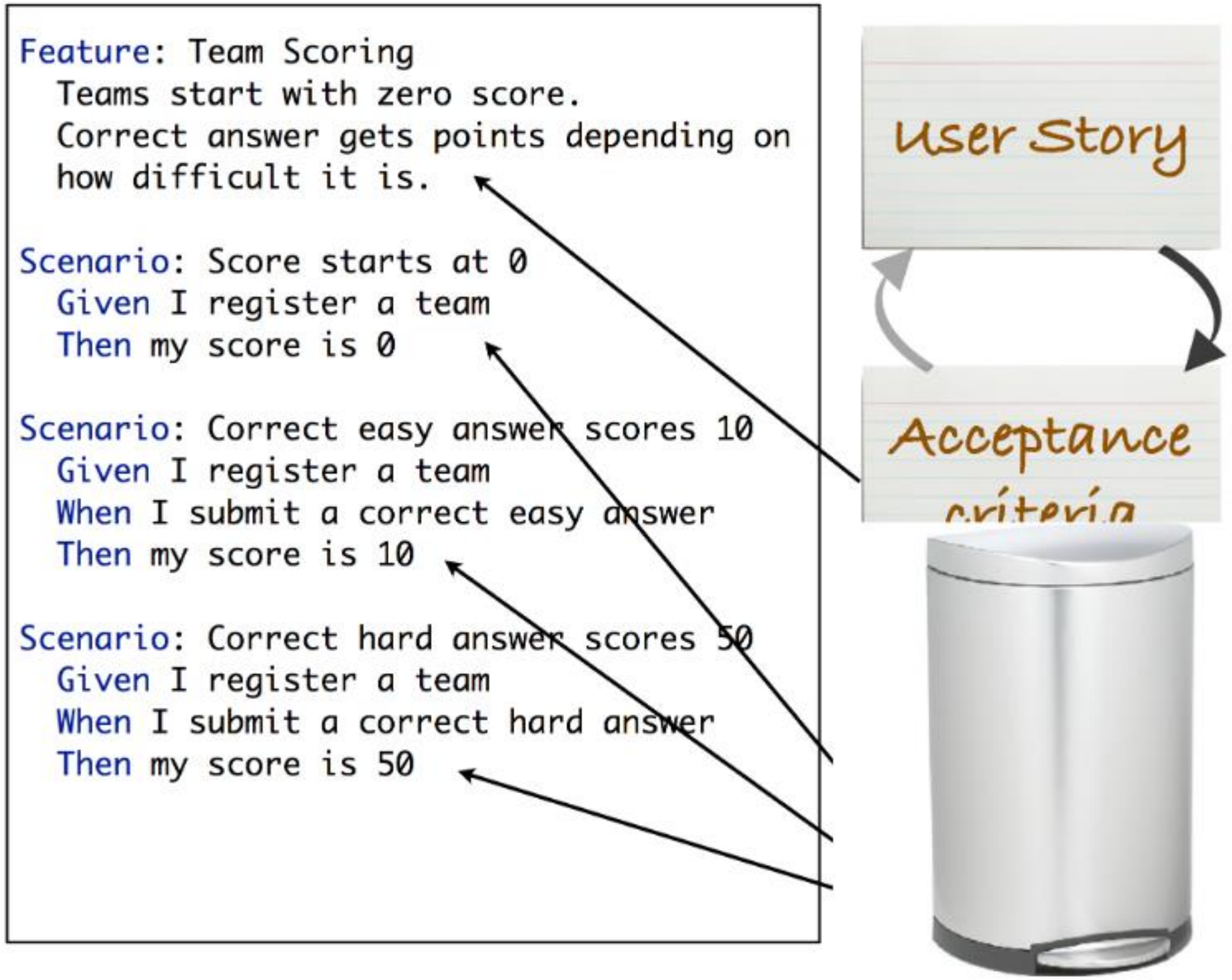
- BDD focuses on projecting a clear understanding of the software's behavior through discussion with the stakeholders.
- It extends the features of test-driven development in a natural language procedure that helps the non-programmers contribute to the betterment of the software product. That is the reason why this practice gives clarity on why the code has to be written, rather than the technical details of the code.
- It also minimizes the radius between technical aspects of the code and the domain-specific language that a business stakeholder uses in his day-to-day life.
- It helps in iterating each element of code that provides some aspect of the behavior, which in collaboration with the other modules of the software, provides the entire application's behavior.
- Thus it functions very similarly to any automation test-based code, but helps make it easier and give a very high-level brief about the technical functions of the software without diving in too deep.
- So which comes first, the User Stories or the Requirements? It all depends on the project. User stories can be used to generate requirements. They can even be used in a Requirements Traceability Matrix (RTM) as a part of the documentation.
- If the client would prefer working with the business analysts, the developers can identify the requirements and, also working with the client, can begin to create user stories. The advantage here is that the client begins to see working software almost immediately. As a result, the client can see what they are asking for and what the developers are planning on delivering. And this will generate ideas, guided by their experience, of other activities they want the software to support.

OK, so BDD is natural language ….tell me more

- The Gherkin language is one of the important aspects of the Behavioral Driven Development strategy. It is a business readable domain-specific language that lets the members of the team understand the behavior of the software without understanding the details of how the backend code has been implemented.
- Gherkin serves two purposes – **documentation** and **automated tests**. Gherkin's grammar is defined by Treetop grammar,
- There are a few conventions while using the Gherkin language:
  - 1. A single Gherkin source file contains a description of a single feature and
  - 2. The source files have a .feature extension. Like Python and YAML.
- Gherkin is a line-based oriented language that uses indentation to define the language structure. Line endings terminate statements (e.g. steps). Either spaces or tabs may be used for indentation (but spaces are more portable). Most lines start with a keyword.

BDD Examples

- **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’.



```
Feature: Team Scoring
  Teams start with zero score.
  Correct answer gets points depending on
  how difficult it is.

Scenario: Score starts at 0
  Given I register a team
  Then my score is 0

Scenario: Correct easy answer scores 10
  Given I register a team
  When I submit a correct easy answer
  Then my score is 10

Scenario: Correct hard answer scores 50
  Given I register a team
  When I submit a correct hard answer
  Then my score is 50
```

Feature files are just plain text files. They have a semi-structure, there is a syntax, and the **keywords are in blue.** The intent is that anyone from your domain should be able to read your feature files and understand exactly what the intent is of the system.

Here is a feature file.

It has a **name at the top saying what the feature is**, it's got some text which tells you what the behaviour or the acceptance criteria is and below it has a number of scenarios that show how the system behaves given certain situations.

The important thing here is that the examples that you might come up with when you're collaborating get recorded as the scenarios.

The **acceptance criteria, which are the rules, the way the system should be behaving are captured in the text** at the top and the important thing here is that user stories, which a lot of Agile teams get very hung up on, are just rubbish at the end of the implementation and should be bent.

BDD Example

Title: To maintain 20 iPhone 7 models in my inventory

- I am a Cell Phone Store Owner
- In order to address fresh orders
- I need to maintain 20 iPhone 7 models in my inventory

Scenario 1:

- Given that a customer has purchased an iPhone 7
- Then my Inventory balance should go down to 19
- When I transfer one model from the Warehouse to Inventory
- Then the inventory balance should go up by 1, to 20

Scenario 2:

- Given that a customer has purchased an iPhone 7
- Then my Inventory balance should go down to 19
- And the customer has decided to return the model
- Then the inventory balance should go back to 20

Cool….so now I just execute this natural language?……Not quite

- You need a way to execute the natural language – > an execution tool

Hiptest	TestLeft	Cucumber	EasyB	JDavE	Concordion
Jbehave	BeanSpec	SpecFlow			

[illegible]

Cucumber Example

Feature: Is it Friday yet?

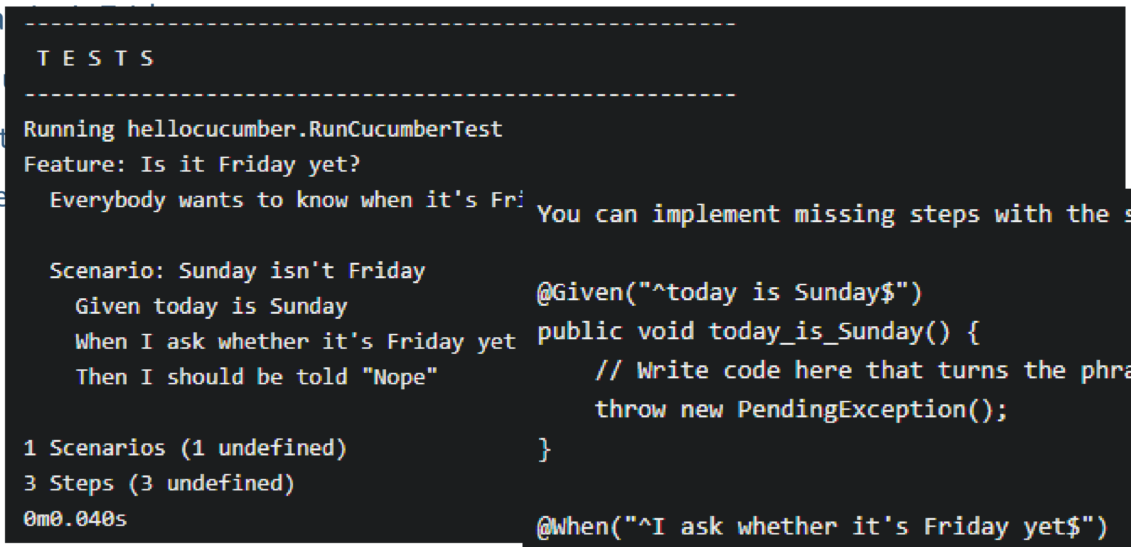
Everybody wants to know when it's
Friday

Scenario: Sunda

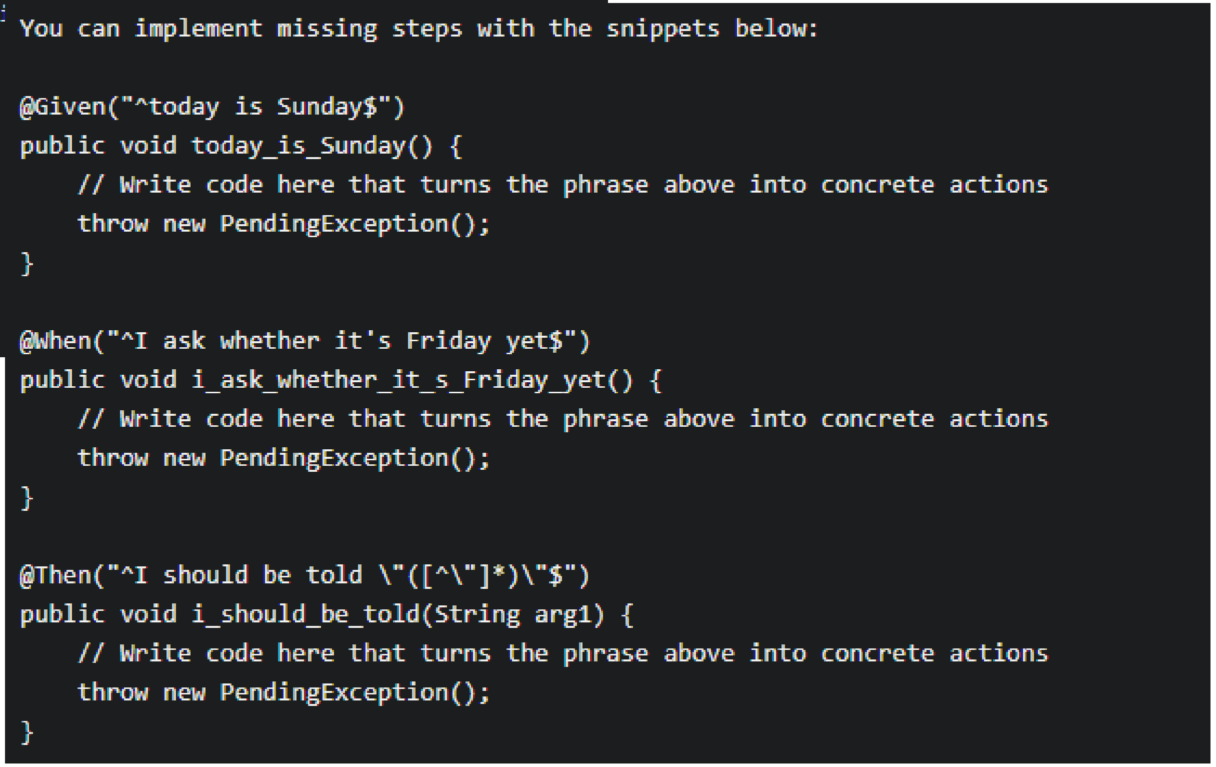
Given today is S

When I ask whet

Then I should be



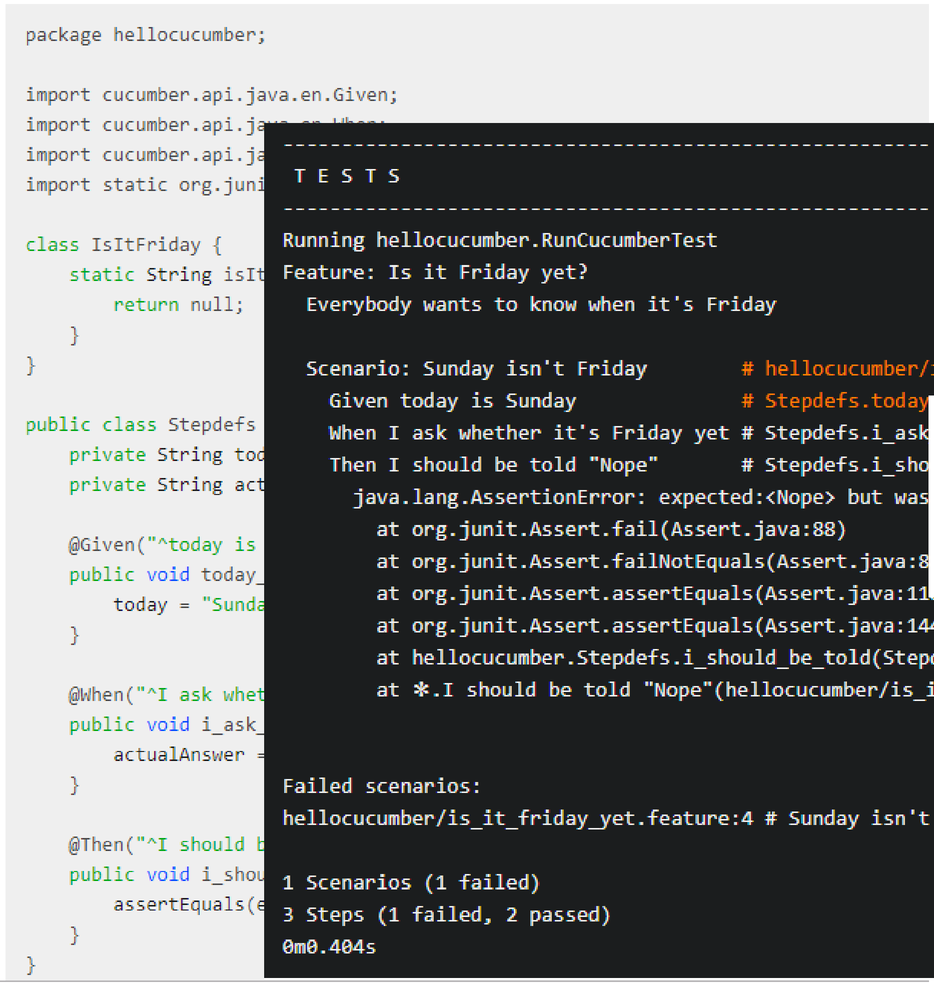
```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running hellocucumber.RunCucumberTest
Feature: Is it Friday yet?
  Everybody wants to know when it's Fri

  Scenario: Sunday isn't Friday
    Given today is Sunday
    When I ask whether it's Friday yet
    Then I should be told "Nope"

1 Scenarios (1 undefined)
3 Steps (3 undefined)
0m0.040s
```



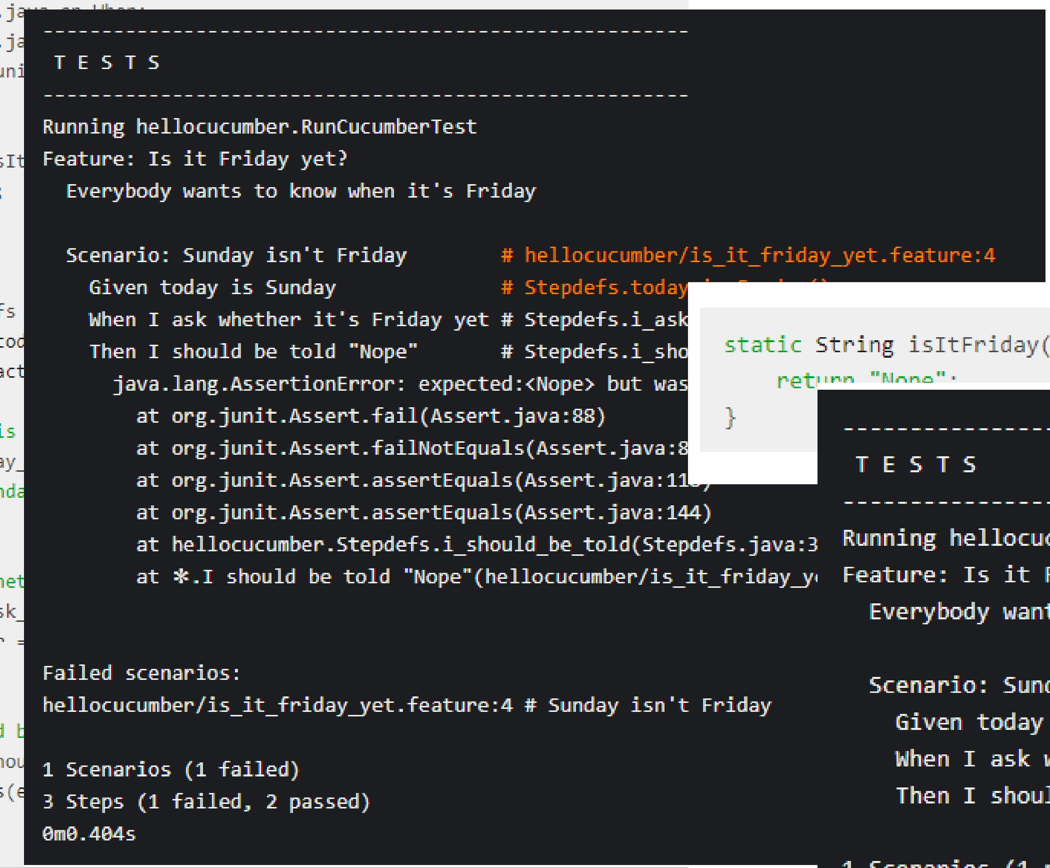
```
You can implement missing steps with the snippets below:

@Given("^today is Sunday$")
public void today_is_Sunday() {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@When("^I ask whether it's Friday yet$")
public void i_ask_whether_it_s_Friday_yet() {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Then("^I should be told \"([^\"]*)\"$")
public void i_should_be_told(String arg1) {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}
```

Cucumber Example - continued

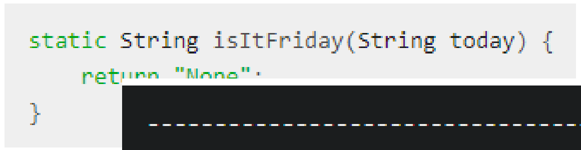
Change your step definition code to this:



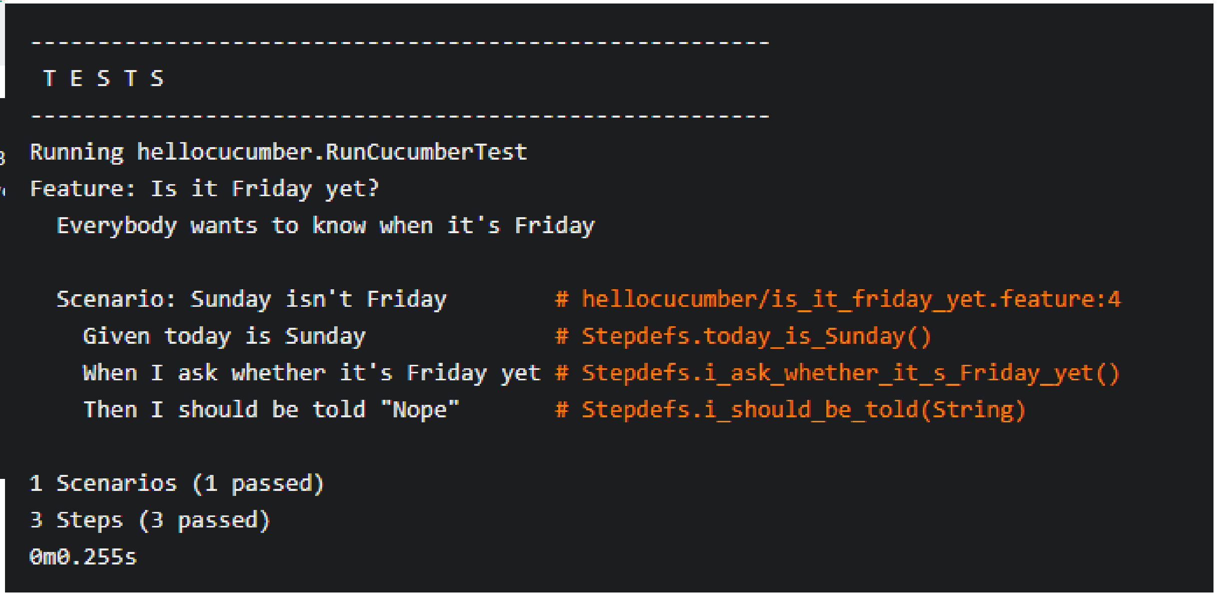
```
package hellocucumber;

import cucumber.api.java.en.Given;
import cucumber.api.ja
import cucumber.api.ja
import static org.juni

class IsItFriday {
    static String isIt
        return null;
    }
}

public class Stepdefs
    private String tod
    private String act

    @Given("^today is
    public void today_
        today = "Sunda
    }

    @When("^I ask whet
    public void i_ask_
        actualAnswer =
    }

    @Then("^I should b
    public void i_shou
        assertEquals(e
    }
}
```



```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running hellocucumber.RunCucumberTest
Feature: Is it Friday yet?
  Everybody wants to know when it's Friday

  Scenario: Sunday isn't Friday        # hellocucumber/is_it_friday_yet.feature:4
    Given today is Sunday              # Stepdefs.today
    When I ask whether it's Friday yet # Stepdefs.i_ask
    Then I should be told "Nope"       # Stepdefs.i_sho
      java.lang.AssertionError: expected:<Nope> but was
        at org.junit.Assert.fail(Assert.java:88)
        at org.junit.Assert.failNotEquals(Assert.java:8
        at org.junit.Assert.assertEquals(Assert.java:11
        at org.junit.Assert.assertEquals(Assert.java:144)
        at hellocucumber.Stepdefs.i_should_be_told(Stepdefs.java:3
        at ✱.I should be told "Nope"(hellocucumber/is_it_friday_y


Failed scenarios:
hellocucumber/is_it_friday_yet.feature:4 # Sunday isn't Friday

1 Scenarios (1 failed)
3 Steps (1 failed, 2 passed)
0m0.404s
```



```
static String isItFriday(String today) {
    return "Nope";
}
```



```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running hellocucumber.RunCucumberTest
Feature: Is it Friday yet?
  Everybody wants to know when it's Friday

  Scenario: Sunday isn't Friday        # hellocucumber/is_it_friday_yet.feature:4
    Given today is Sunday              # Stepdefs.today_is_Sunday()
    When I ask whether it's Friday yet # Stepdefs.i_ask_whether_it_s_Friday_yet()
    Then I should be told "Nope"       # Stepdefs.i_should_be_told(String)

1 Scenarios (1 passed)
3 Steps (3 passed)
0m0.255s
```

Cucumber Example - continued

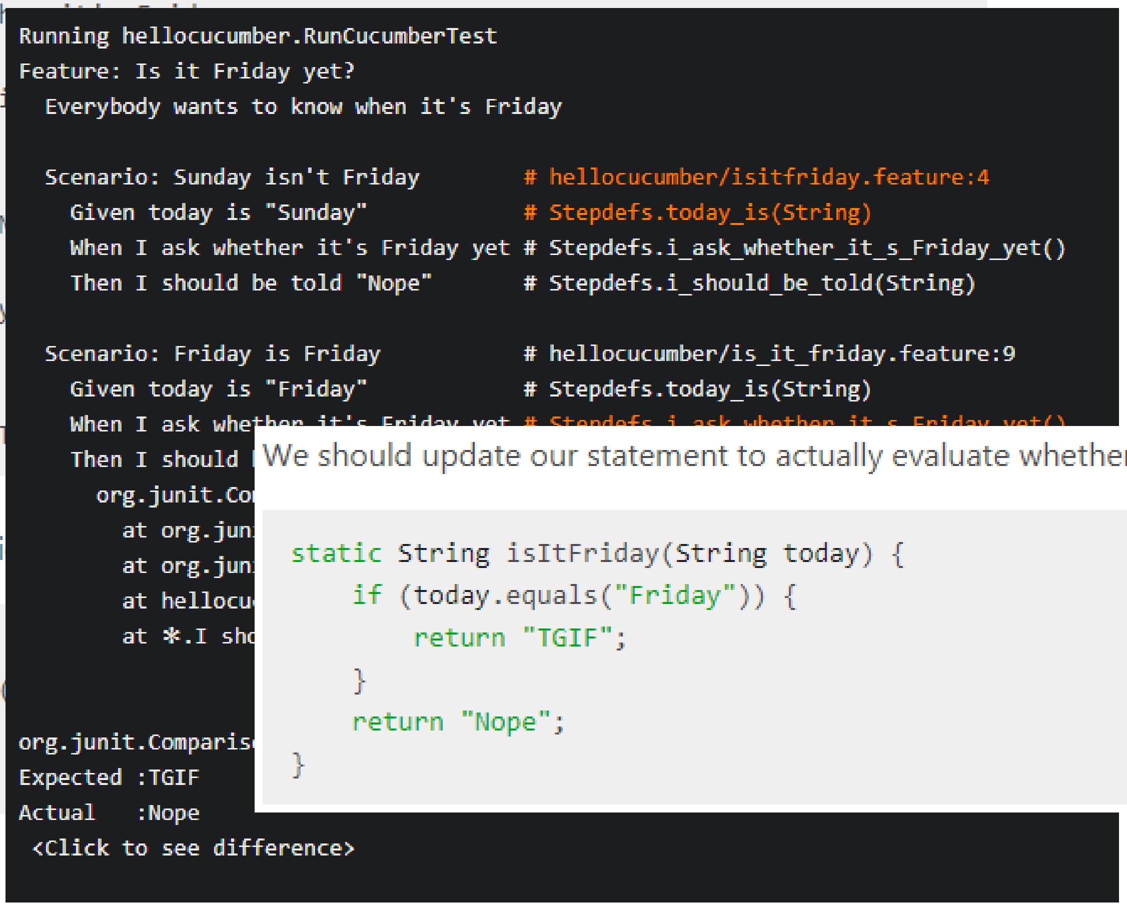
The next thing to test for would be that we also get the correct result when it *is* Friday.

Update the `is-it-friday-yet.feature` file:

```
Feature: Is it Friday yet?
  Everybody wants to know wh

  Scenario: Sunday isn't Fri
    Given today is Sunday
    When I ask whether it's
    Then I should be told "N

  Scenario: Friday is Friday
    Given today is Friday
    When I ask whether it's
    Then I should be told "T
```

We'll need to add a step definiti

```
@Given("^today is Friday$")
public void today_is_Friday(
    this.today = "Friday";
}
```



```
Running hellocucumber.RunCucumberTest
Feature: Is it Friday yet?
  Everybody wants to know when it's Friday

  Scenario: Sunday isn't Friday        # hellocucumber/isitfriday.feature:4
    Given today is "Sunday"            # Stepdefs.today_is(String)
    When I ask whether it's Friday yet # Stepdefs.i_ask_whether_it_s_Friday_yet()
    Then I should be told "Nope"       # Stepdefs.i_should_be_told(String)

  Scenario: Friday is Friday           # hellocucumber/is_it_friday.feature:9
    Given today is "Friday"            # Stepdefs.today_is(String)
    When I ask whether it's Friday yet # Stepdefs.i_ask_whether_it_s_Friday_yet()
    Then I should
      org.junit.Co
        at org.jun
        at org.jun
        at hellocu
        at ✱.I sho

org.junit.Comparis
Expected :TGIF
Actual   :Nope
 <Click to see difference>
```

We should update our statement to actually evaluate whether or not `today` is equal to `"Friday"`.

```
static String isItFriday(String today) {
    if (today.equals("Friday")) {
        return "TGIF";
    }
    return "Nope";
}
```

ETC in an iterative mode

Benefits / disadvantages of BDD

- **Benefits to using BDD**
  - Since communication is essential between the parties - user/client and the development team, absence of any one of them, can cause the process to have ambiguities and lack of answers to the questions/doubts raised by either side
  - You are no longer defining ‘test’, but are defining ‘behavior’.
  - Better communication between developers, testers and product owners.
  - Because BDD is explained using simple language, the learning curve will be much shorter.
  - Being non-technical in nature, it can reach a wider audience.
  - The behavioral approach defines acceptance criteria prior to development.
- **Disadvantages of BDD**
  - the need to dedicate a team of developers to work with the client. The short response time required for the process means high levels of availability
  - BDD is incompatible with the waterfall approach.
  - If the requirements are not properly specified, BDD may not be effective.
  - Testers using BDD need to have sufficient technical skills.
  - Requires injecting snippets of code-----and continued maintenance

Demo

Model Based tool using Gherkin to auto generate/optimize test cases and test scripts using Java and then run them in Cucumber