Software Engineering and Architecture
Lecture 5: Agile Testing (1)

Olivier Liechti
olivier.liechti@heig-vd.ch

MSE | MASTER OF SCIENCE IN ENGINEERING

Today.

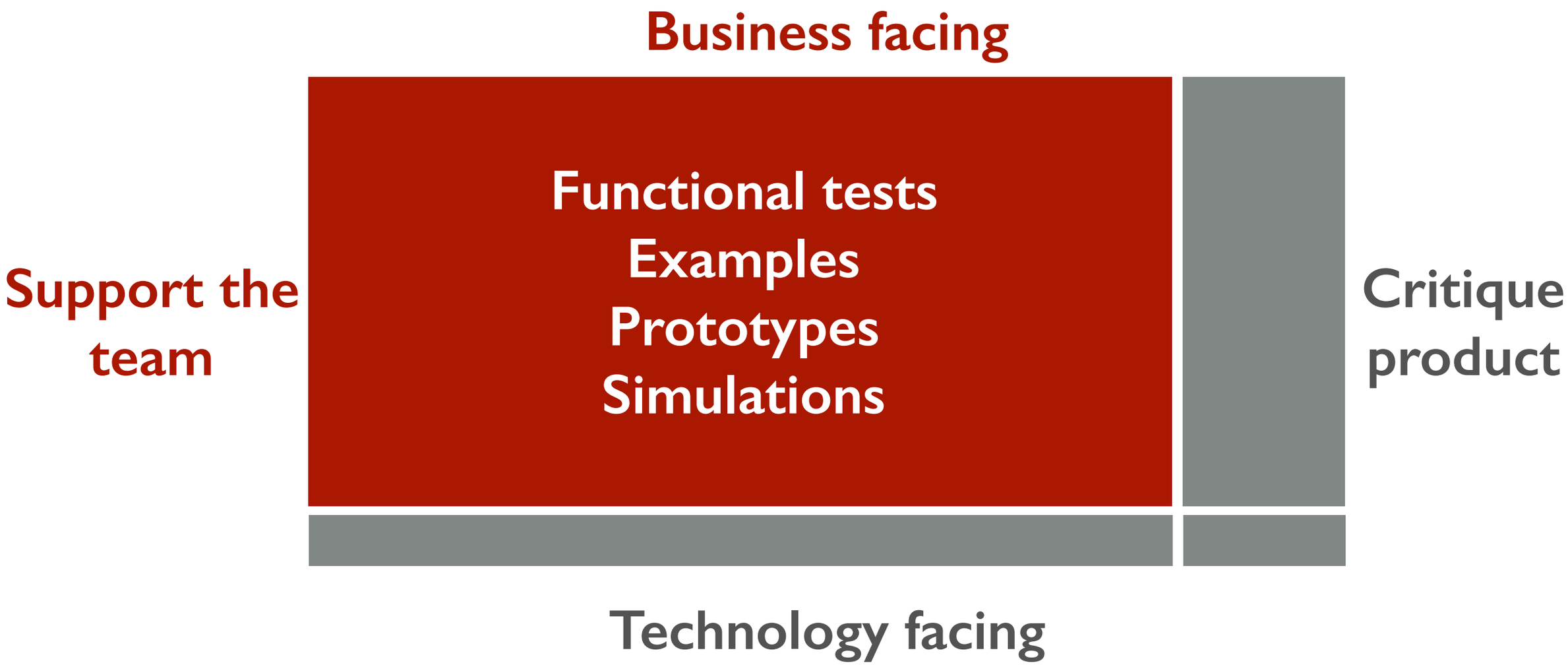
Agile testing - BDD
15h - 15h30

OO Reengineering Patterns (3 x 10’)
15h30 - 16h00

Open Affect, API specification and validation
16h00 - 17h20

Week	Theory	OO Reengineering	Practice
#1	Agile, Scrum		Intro to Docker
#2	Software evolution	Introduction	Specify and implement a micro-service
#3	Continuous delivery (1)	*Setting Directions*	Intro to Jenkins & Travis
#4	Continuous delivery (2)	*First Contact*	Our first build pipeline
#5	Agile testing (1)	*Initial Understanding*	Intro to Cucumber
#6	**Agile testing (2)**	***Detailed Model Capture***	**Add tests to pipeline**
#7	Agile metrics	*Tests: Your Life Insurance!*	Add Sonar to pipeline
#8	Continuous improvement	*Migration Strategies I*	Add non-functional tests
#9	Wrap-up	*Migration Strategies II*	

Agile testing quadrants: Q2



Functional tests

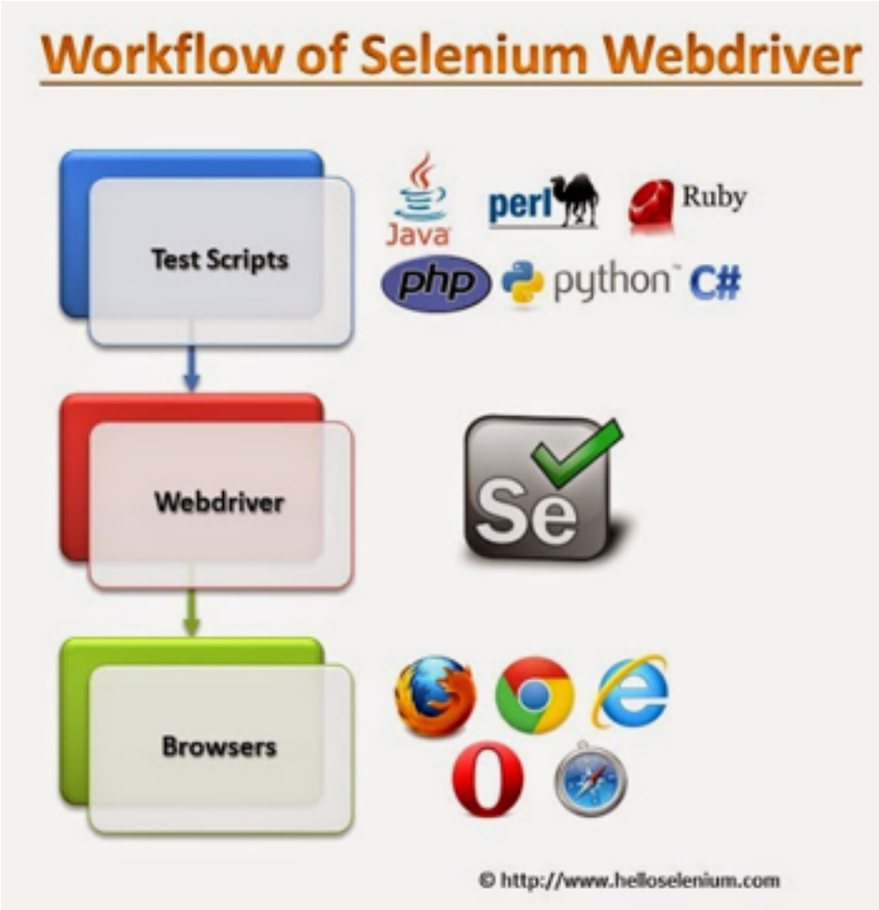
- With **functional tests**, we want to validate that the system does what it it supposed to do **from the users point of view**.
- Very often, this means **defining usage scenarios** (**test cases**). We describe the steps to be followed by users and the expected results.
- When we evaluate a software release, we can **check** whether the defined test cases can be executed with success.

Manual functional tests

- In many organizations, test cases are documented in **test management software**. They are executed by **human operators**.
- This is a **repetitive process** with little added value.
- This is a **slow process**.
- It creates **overhead** and often gives a **false sense of confidence**.
- If you release every 3 months, it “might” be possible to do manual test campaigns. If you release on a weekly basis, it is just not possible.

Automated functional tests

- There are now **tools** that can be used to **simulate human users**.
- With these tools, you write scripts. When the scripts are executed, they **control a web browser** and check that the content of the pages is.
- **It is not a free lunch**. Writing these scripts takes time. Maintaining these scripts (when the UI changes) takes a lot of time.
- Integration tests are slower than unit tests. Automated functional tests are **a lot slower** than integration tests.
- For this reason, they are not executed as often (at a later stage in the continuous delivery pipeline).



Behaviour Driven Development (BDD)

- With Unit Tests, developers have a way to **specify and check** the behaviour of a tiny piece of code.
- The same principle can be applied with higher-level, **business oriented tests**. This is the idea of “behaviour driven development” or BDD.
- BDD is a method that **facilitates the collaboration** between business analysis, developers and testers. It gives them a **common vocabulary**.

*"My response is **behaviour-driven development** (BDD). It has evolved out of established agile practices and is designed to make them more accessible and effective for teams new to agile software delivery.*

*Over time, BDD has grown to **encompass the wider picture of agile analysis and automated acceptance testing**."*

Dan North, 2006

BDD: Naming & Vocabulary Matters

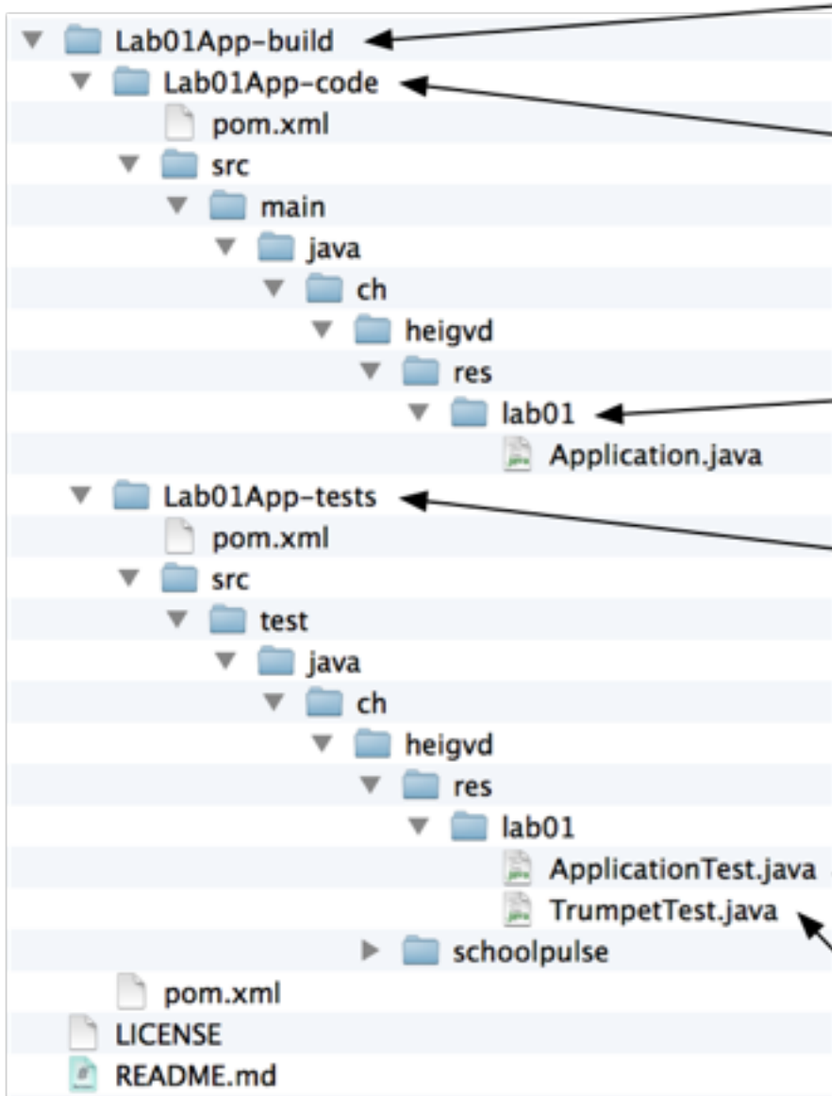
- “Test method names should be sentences”.
- Compare the two representations of the same “specification”. It suggests that tools can support communication by emphasizing a common language for the domain.

see http://agiledox.sourceforge.net/

```
public class FooTest extends TestCase {
    public void testIsASingleton() {}
    public void testAReallyLongNameIsAGoodThing() {}
}
```

```
Foo
- is a singleton
- a really long name is a good thing
```

Example: the “Instruments” lab



This is a "parent" project, which contains the two others. It is what we use to build and test our application. You can open this project in Netbeans.

This is a project that you can open in Netbeans. It is the project that contains the sources for your application.

We give you a partial implementation. At the beginning, there is only one source file, with one bug. ***You will add code here.***

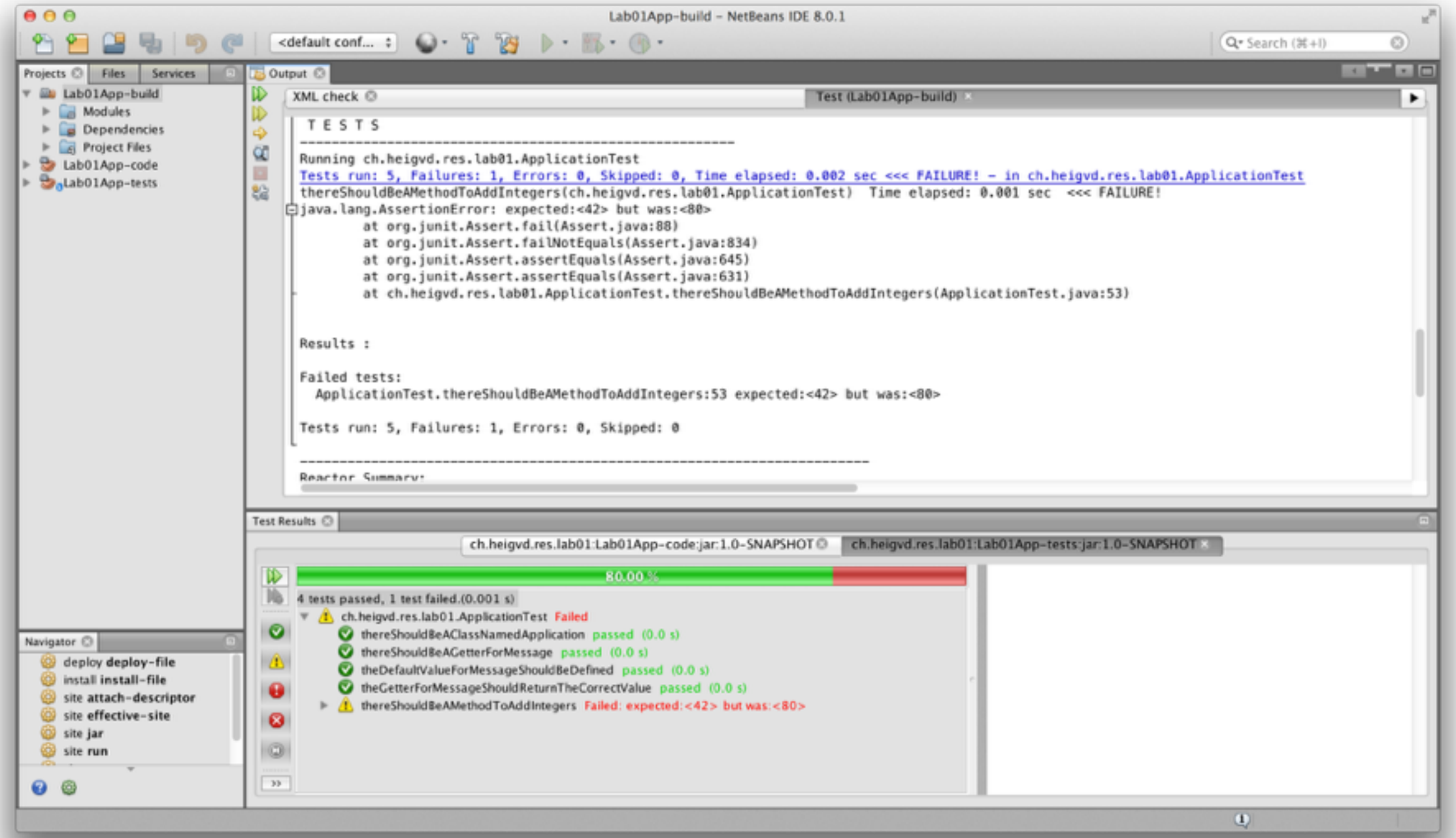
This is also a project that you can open in Netbeans. It contains the unit tests that specify what your application should do. The tests are used to validate your code.

This class contains unit tests to specify and validate the behavior of Application.java. At the beginning, one unit test fails (because we have one bug in Application.java).

This class contains unit tests for code that you have to write. The tests are commented so that you can compile the project after the clone.

```java
public class ApplicationTest {

  @Test
  public void thereShouldBeAClassNamedApplication() {
    Application application = new Application();
    assertNotNull(application);
  }

  @Test
  public void thereShouldBeAGetterForMessage() {
    Application application = new Application();
    String message = application.getMessage();
    assertNotNull(message);
  }

  @Test
  public void theGetterForMessageShouldReturnTheCorrectValue() {
    String testValue = "does it work?";
    Application application = new Application(testValue);
    String message = application.getMessage();
    assertEquals(testValue, message);
  }

  @Test
  public void theDefaultValueForMessageShouldBeDefined() {
    Application application = new Application();
    String message = application.getMessage();
    assertEquals("HEIG-VD rocks!", message);
  }

  @Test
  public void thereShouldBeAMethodToAddIntegers() {
    Application application = new Application();
    int sum = application.add(40, 2);
    assertEquals(42, sum);
  }

}
```

```java
@Test
public void thereShouldBeAnIInstrumentInterfaceAndATrumpetClas
  IInstrument trumpet = new Trumpet();
  assertNotNull(trumpet);
}

@Test
public void itShouldBePossibleToPlayAnInstrument() {
  IInstrument trumpet = new Trumpet();
  String sound = trumpet.play();
  assertNotNull(sound);
}

@Test
public void aTrumpetShouldMakePouet() {
  IInstrument trumpet = new Trumpet();
  String sound = trumpet.play();
  Assert.assertEquals("pouet", sound);
}

@Test
public void aTrumpetShouldBeLouderThanAFlute() {
  IInstrument trumpet = new Trumpet();
  IInstrument flute = new Flute();
  int trumpetVolume = trumpet.getSoundVolume();
  int fluteVolume = flute.getSoundVolume();
  Assert.assertTrue(trumpetVolume > fluteVolume);
}
@Test
public void aTrumpetShouldBeGolden() {
  IInstrument trumpet = new Trumpet();
  String color = trumpet.getColor();
  Assert.assertEquals("golden", color);
}
```


Lab01App-build - NetBeans IDE 8.0.1
Projects
Files
Services
Lab01App-build
Modules
Dependencies
Project Files
Lab01App-code
Lab01App-tests
Output
XML check
Test (Lab01App-build)
T E S T S
Running ch.heigvd.res.lab01.ApplicationTest
Tests run: 5, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.002 sec <<< FAILURE! - in ch.heigvd.res.lab01.ApplicationTest
thereShouldBeAMethodToAddIntegers(ch.heigvd.res.lab01.ApplicationTest) Time elapsed: 0.001 sec <<< FAILURE!
java.lang.AssertionError: expected:<42> but was:<80>
at org.junit.Assert.fail(Assert.java:88)
at org.junit.Assert.failNotEquals(Assert.java:834)
at org.junit.Assert.assertEquals(Assert.java:645)
at org.junit.Assert.assertEquals(Assert.java:631)
at ch.heigvd.res.lab01.ApplicationTest.thereShouldBeAMethodToAddIntegers(ApplicationTest.java:53)
Results :
Failed tests:
ApplicationTest.thereShouldBeAMethodToAddIntegers:53 expected:<42> but was:<80>
Tests run: 5, Failures: 1, Errors: 0, Skipped: 0
Reactor Summary:
Test Results
ch.heigvd.res.lab01:Lab01App-code:jar:1.0-SNAPSHOT
ch.heigvd.res.lab01:Lab01App-tests:jar:1.0-SNAPSHOT
80.00 %
4 tests passed, 1 test failed.(0.001 s)
ch.heigvd.res.lab01.ApplicationTest Failed
thereShouldBeAClassNamedApplication passed (0.0 s)
thereShouldBeAGetterForMessage passed (0.0 s)
theDefaultValueForMessageShouldBeDefined passed (0.0 s)
theGetterForMessageShouldReturnTheCorrectValue passed (0.0 s)
thereShouldBeAMethodToAddIntegers Failed: expected:<42> but was:<80>
Navigator
deploy deploy-file
install install-file
site attach-descriptor
site effective-site
site jar
site run

BDD: “Ubiquitous Language” for Analysis

- BDD proposes a template to describe the intended behaviour of a system. The template is used to specify the acceptance criteria for a given user story.

```
Given some initial context (the givens),
When an event occurs,
then ensure some outcomes.
```

```
USER STORY
As a customer,
I want to withdraw cash from
an ATM,
so that I don’t have to wait
in line at the bank.
```

```
ACCEPTANCE CRITERIA
Given the account is in credit
And the card is valid
And the dispenser contains cash
When the customer requests cash
Then ensure the account is debited
And ensure cash is dispensed
And ensure the card is returned
```

BDD: Executable Specifications

- **“Acceptance criteria should be executable”**
- We need tools that allow:
  - **analysts** to write the acceptance criteria in plain english, following the previous template;
  - **developers** to write test fixtures that act as intermediary between the specification and the system to test;
  - the **continuous delivery pipeline** to execute the specifications automatically, to integrate the test results in the “live” specification, to notify the team about the results.

Process : When will be done?

```
Scenario:  trader is not alerted below threshold

Given a stock of symbol STK1 and a threshold of 10.0
When the stock is traded at 5.0
Then the alert status should be OFF
```

Executable Specifications

Acceptance criteria for stories are defined as scenarios.

Process : linking the specs with the system

Executable Specifications

```
Scenario:  trader is not alerted below threshold

Given a stock of symbol STK1 and a threshold of 10.0
When the stock is traded at 5.0
Then the alert status should be OFF
```

Test Fixtures

```
public class TraderSteps { // look, Ma, I'm a POJO!!

    private Stock stock;

    @Given("a stock of symbol $symbol and a threshold
of $threshold")
    public void aStock(String symbol, double threshold)
{
        stock = new Stock(symbol, threshold);
    }

    @When("the stock is traded at $price")
    public void theStockIsTradedAt(double price) {
        stock.tradeAt(price);
    }

    @Then("the alert status should be $status")
    public void theAlertStatusShouldBe(String status) {
        ensureThat(stock.getStatus().name(),
equalTo(status));
    }

}
```

System Under Test (SUT)

Process : let's see if we are done...

```
Scenario:  trader is not alerted below threshold

Given a stock of symbol STK1 and a threshold of 10.0
When the stock is traded at 5.0
Then the alert status should be OFF
```

Executable Specifications

The test results are displayed directly in the "living" specs (other reports and notifications are also useful!)

Process : yeah!!!!!!

```
Scenario:  trader is not alerted below threshold

Given a stock of symbol STK1 and a threshold of 10.0
When the stock is traded at 5.0
Then the alert status should be OFF
```

Executable Specificati

The test results are displayed directly in the "living" specs (other reports and notifications are also useful!)

Process : noooooooooo....

```
Scenario:  trader is not alerted below threshold

Given a stock of symbol STK1 and a threshold of 10.0
When the stock is traded at 5.0
Then the alert status should be OFF
```

Executable Specifications

The test results are displayed directly in the "living" specs (other reports and notifications are also useful!)

I can’t wait to get started... what should I do?

Tools

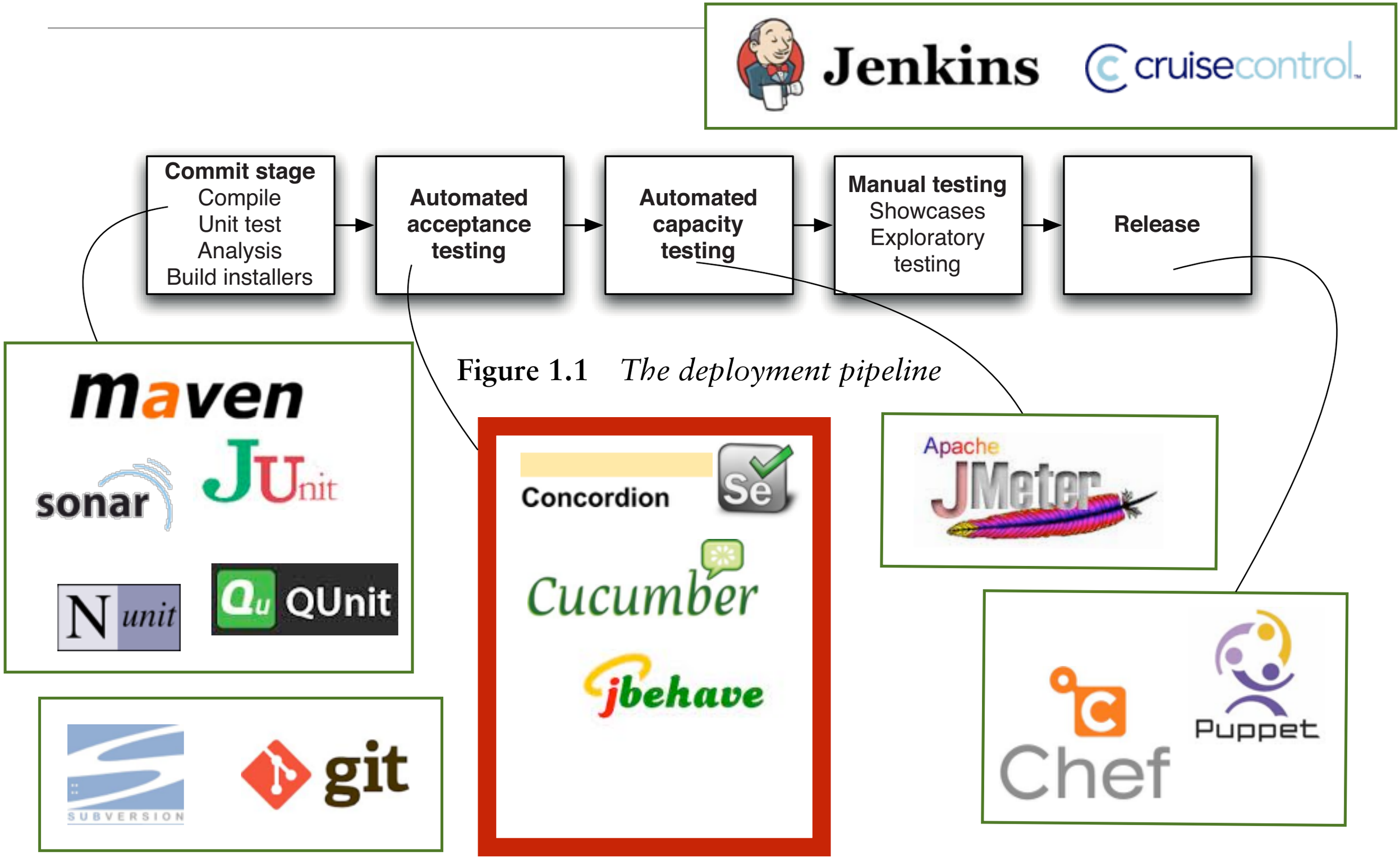


Figure 1.1 *The deployment pipeline*

Getting started with Cucumber JVM


Docs
Blog
Events
Videos
Training
Cucumber Pro
Support
cucumber
Simple, human collaboration
BDD Kickstart - Boston, US - August 2017
An open-source tool for executable specifications
A vibrant community
An ingenious company

Dependency

If you are going to use the lambda expressions API to write the Step Definitions, you need:

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-java8</artifactId>
    <version>1.2.5</version>
    <scope>test</scope>
</dependency>
```

Otherwise, to write them using annotated methods, you need:

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>1.2.5</version>
    <scope>test</scope>
</dependency>
```

While it's not required, we strongly recommend you include one of the Dependency Injection modules as well. This allows you to share state between Step Definitions without resorting to static variables (a common source of flickering scenarios).

PicoContainer

Dependency

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-picocontainer</artifactId>
    <version>1.2.5</version>
    <scope>test</scope>
</dependency>
```

Step dependencies

The picocontainer will create singleton instances of any Step class dependencies which are constructor parameters and inject them into the Step class instances when constructing them.

Step scope and lifecycle

All step classes and their dependencies will be recreated fresh for each scenario, even if the scenario in question does not use any steps from that particular class.

If any step classes or dependencies use expensive resources (such as database connections), you should create them lazily on-demand, rather than eagerly, to improve performance.

Step classes or their dependencies which own resources which need cleanup should implement org.picocontainer.Disposable as described at http://picocontainer.com/lifecycle.html . These callbacks will run after any cucumber.api.java.After callbacks.

```
Feature: Creation of fruits

  Background:
    Given there is a Fruits server

  Scenario: create a fruit
    Given I have a fruit payload
    When I POST it to the /fruits endpoint
    Then I receive a 201 status code
```

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.avalia.fruits.api.spec.SpecificationTest
Feature: Creation of fruits

  Background:                          # creation.feature:3
    Given there is a Fruits server

  Scenario: create a fruit                    # creation.feature:6
    Given I have a fruit payload
    When I POST it to the /fruits endpoint
    Then I receive a 201 status code

1 Scenarios (1 undefined)
4 Steps (4 undefined)
0m0.000s
```

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

@Given("^there is a Fruits server$")
public void there_is_a_Fruits_server() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

...
```

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.avalia.fruits.api.spec.SpecificationTest
Feature: Creation of fruits

  Background:                        # creation.feature:3
    Given there is a Fruits server # CreationSteps.there_is_a_Fruits_server()
      cucumber.api.PendingException: TODO: implement me
      at
io.avalia.fruits.api.spec.steps.CreationSteps.there_is_a_Fruits_server(CreationSteps.java:16)
      at *.Given there is a Fruits server(creation.feature:4)


  Scenario: create a fruit                 # creation.feature:6
    Given I have a fruit payload           # CreationSteps.i_have_a_fruit_payload()
    When I POST it to the /fruits endpoint # CreationSteps.i_POST_it_to_the_fruits_endpoint()
    Then I receive a 201 status code       # CreationSteps.i_receive_a_status_code(int)

1 Scenarios (1 pending)
4 Steps (3 skipped, 1 pending)
0m0.101s
```

```java
public class CreationSteps {

    private Environment environment;
    private DefaultApi api;
    private ApiResponse lastApiResponse;
    private ApiException lastApiException;
    private boolean lastApiCallThrewException;
    private int lastStatusCode;
    Fruit fruit;

    public CreationSteps(Environment environment) {
        this.environment = environment;
        this.api = environment.getApi();
    }

    @Given("^there is a Fruits server$")
    public void there_is_a_Fruits_server() throws Throwable {
        assertNotNull(api);
    }

    @Given("^I have a fruit payload$")
    public void i_have_a_fruit_payload() throws Throwable {
        fruit = new io.avalia.fruits.api.dto.Fruit();
    }

    @When("^I POST it to the /fruits endpoint$")
    public void i_POST_it_to_the_fruits_endpoint() throws Throwable {
        try {
            lastApiResponse = api.createFruitWithHttpInfo(fruit);
            lastApiCallThrewException = false;
            lastApiException = null;
            lastStatusCode = lastApiResponse.getStatusCode();
        } catch (ApiException e) {
            lastApiCallThrewException = true;
            lastApiResponse = null;
            lastApiException = e;
            lastStatusCode = lastApiException.getCode();
        }

    }

    @Then("^I receive a (\\d+) status code$")
    public void i_receive_a_status_code(int arg1) throws Throwable {
        assertEquals(201, lastStatusCode);
    }

}
```

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.avalia.fruits.api.spec.SpecificationTest
Feature: Creation of fruits

  Background:                       # creation.feature:3
    Given there is a Fruits server # CreationSteps.there_is_a_Fruits_server()

  Scenario: create a fruit                 # creation.feature:6
    Given I have a fruit payload           # CreationSteps.i_have_a_fruit_payload()
    When I POST it to the /fruits endpoint # CreationSteps.i_POST_it_to_the_fruits_endpoint()
    Then I receive a 201 status code       # CreationSteps.i_receive_a_status_code(int)

1 Scenarios (1 passed)
4 Steps (4 passed)
0m0.496s

Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.824 sec
```

Specify - Code - Validate

with

Swagger,
Spring Boot &
Cucumber JVM

Examples in these repos:

Bare minimum “Fruits” API

https://github.com/AvaliaSystems/TrainingREST/tree/swagger-springboot-cucumber-codegen

Simple “Open Affect” API

https://github.com/openaffect/openaffect-server

Multiple micro-services platforms

https://github.com/PestaKit

https://github.com/LozziKit

AvaliaSystems / TrainingREST

Unwatch 1 | Star 0 | Fork 2

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Insights | Settings

Training material

Edit

Add topics

8 commits | 5 branches | 0 releases | 1 contributor | MIT

Your recently pushed branches:

swagger-springboot-cucumber-codegen (3 minutes ago) — Compare & pull request

Branch: swagger-spring... | New pull request | Create new file | Upload files | Find file | Clone or download

This branch is 7 commits ahead, 1 commit behind master. — Pull request | Compare

wasadigi Add instructions — Latest commit bf874b3 4 minutes ago

swagger	Add instructions	4 minutes ago
.gitignore	Add intellij config to .gitignore	10 months ago
LICENSE	Initial commit	10 months ago
README.md	Add instructions	4 minutes ago

README.md

TrainingREST

Build and run the Fruit microservice

You can use maven to build and run the REST API implementation from the command line. After invoking the following maven goal, the Spring Boot server will be up and running, listening for connections on port 8080.

```
cd swagger/spring-server/
mvn spring-boot:run
```

You can then access:

- the API documentation, generated from annotations in the code
- the API endpoint, accepting GET and POST requests

Combined top-down & bottom-up development process

This is done only once

1. ***Specify*** *API v0.1 in api-spec.yml*
2. ***Generate*** *Spring Boot skeleton from Swagger editor*
3. ***Modify*** *pom.xml to enable API updates (the hard part)*

This is done many times

1. ***Update*** *API to v0.x in api-spec.yml*
2. ***Build*** *with maven (mvn clean install)*
3. *The swagger codegen maven plugin* ***re-generates*** *the interfaces (endpoint signatures, DTOs)*
4. ***Implement*** *the new behaviour*
5. ***Annotations*** *on the controllers are used to generate the live html documentation (springfox)*

Combined top-down & bottom-up development process

Apply the same process to

1) the specification (Cucumber scenarios)
2) the implementation (Spring Boot service)

1. ***Specify*** *API v0.1 in api-spec.yml*
2. ***Generate*** *Spring Boot skeleton from Swagger editor*
3. ***Modify*** *pom.xml to enable API updates (the hard part)*

1. ***Update*** *API to v0.x in api-spec.yml*
2. ***Build*** *with maven (mvn clean install)*
3. *The swagger codegen maven plugin* ***re-generates*** *the interfaces (endpoint signatures, DTOs)*
4. ***Implement*** *the new behaviour*
5. ***Annotations*** *on the controllers are used to generate the live html documentation (springfox)*

AvaliaSystems / TrainingREST

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Insights

Branch: swagger-spring... | TrainingREST / swagger / | Cre

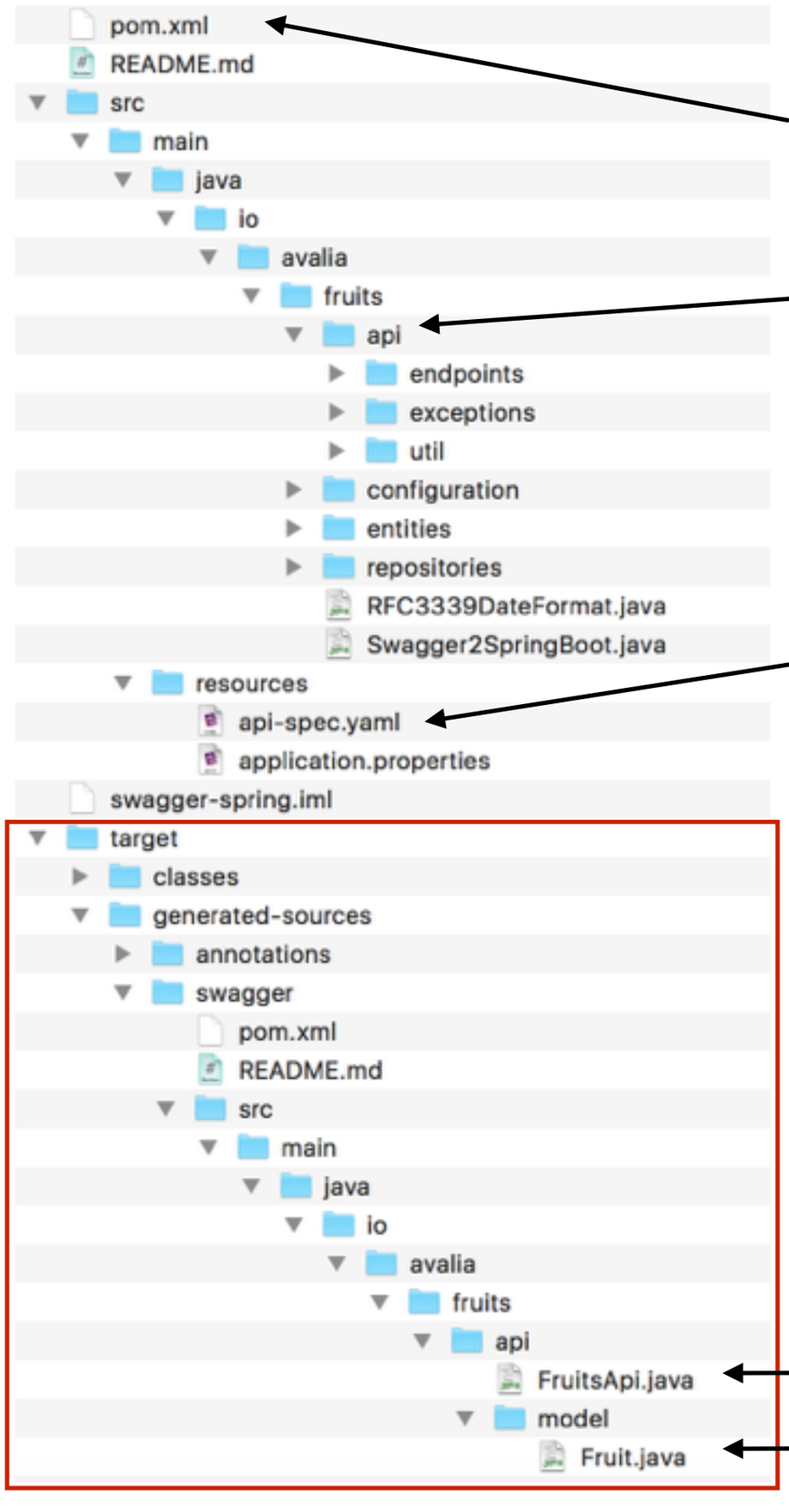
This branch is 6 commits ahead of master.

wasadigi Add swagger codegen in server, refactor packages, add top project

..	
examples	Add Postman collection to test Fruits API
fruits-specs	Add swagger codegen in server, refactor packages, add top project
full-project/.idea	Add swagger codegen in server, refactor packages, add top project
spring-server	Add swagger codegen in server, refactor packages, add top project

The implementation (Spring Boot)

The executable spec (Cucumber)



pom.xml contains the magic that puts all things together.

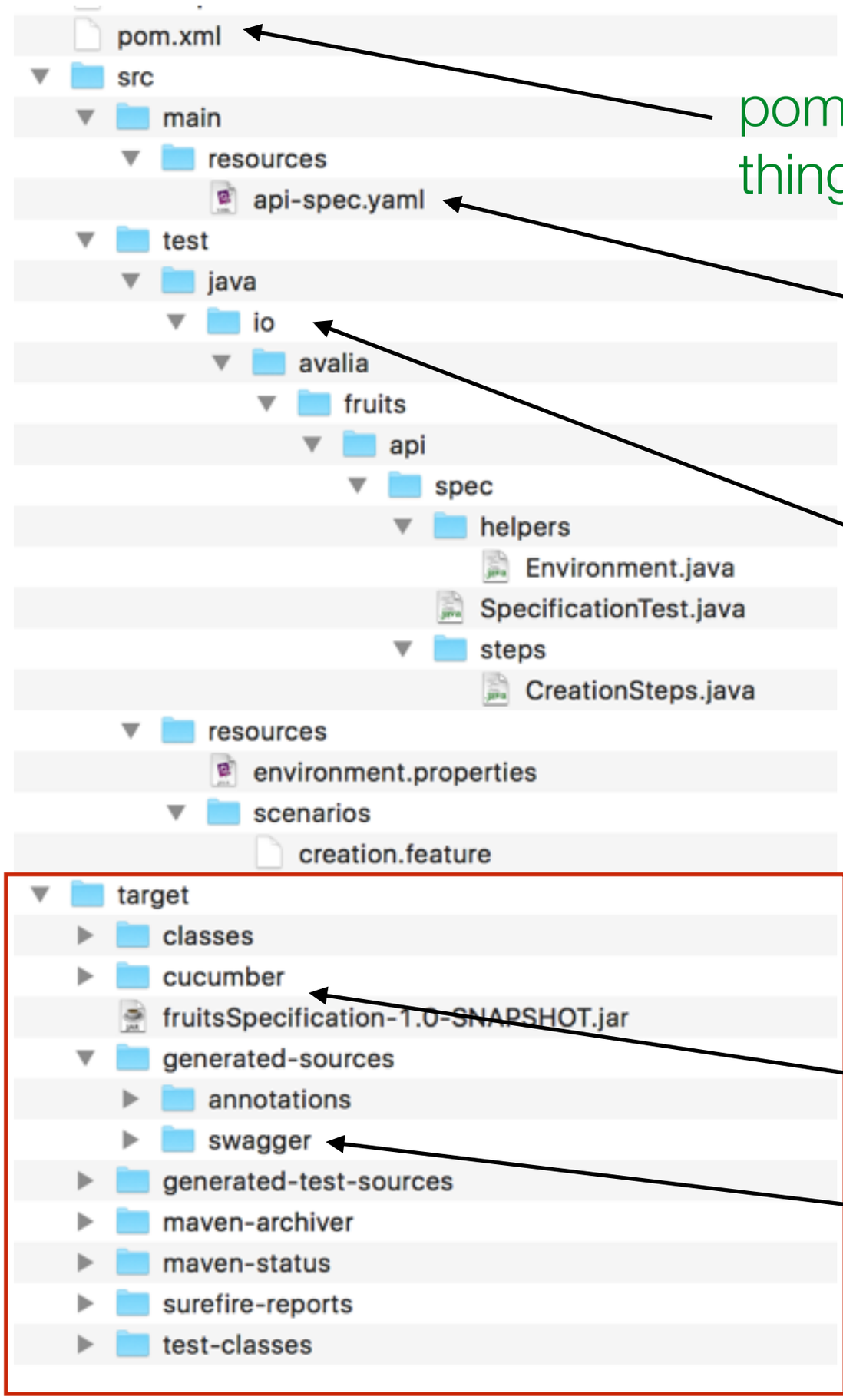
Our implementation: it is not modified automatically when we do a build. If we change the interfaces (endpoints / DTOs), we must update this code.

api-spec.yaml is our API spec. If we change endpoints or DTOs, we update this file.

This is generated each time when we do mvn clean install

Our REST endpoint, as defined with Swagger

Our DTO, as defined with Swagger



pom.xml contains the magic that puts all things together.

api-spec.yaml is our API spec. If we change endpoints or DTOs, we update this file.

The fixtures: the Java code that is executed when BDD scenarios are replayed by Cucumber

This is generated each time when we do mvn clean test

This contains the test results

This contains **client stubs**, that we can use in our fixtures to invoke the API