Lab 5 - Acceptance Testing with Cucumber

SENG301 Software Engineering II

Neville Churcher, Morgan English, Fabian Gilson, Karl Moore, Jack Patterson, Saskia van der Peet

23rd March 2023

Lab objectives

This lab aims to help students with their first use of acceptance test automation tools by using the *Cucumber* framework. Students will learn:

- why automated acceptance tests are important;
- how to set up the a `gradle` project with *Cucumber*;
- how to convert stories' acceptance criteria to code that will contribute to the success of a delivery.

1 Context

1.1 Acceptance testing

In Agile Software Development, acceptance criteria play a key role for the success of a product delivery. User stories are meant to be short and reflecting a *"promise for a conversation"*. Practical details are expressed in acceptance criteria representing the expected behaviour of a story under certain circumstances (e.g., normal or exceptional cases) or give more details about the exact attributes of concepts (i.e. domain entities) used in a story. This set of criteria is used by both the product owner and the developers as a sort of *"contract"* to agree on what is expected from a story to be successfully implemented (i.e. pass).

These tests are different from unit tests, which are aimed at developers and help them ensure the code works as expected or even drive the code design (e.g., Test-Driven Development). In some ways, *"unit tests ensure you build the thing right, whereas acceptance tests ensure you build the right thing"*.

1.2 Acceptance Test-Driven development (ATDD)

Together with *Story Test-Driven Development* and *Specification by example*, *Acceptance Test-Driven development* (ATDD) was coined by Kent Beck in 2003, one of Agile's father[1]. Despite Beck dismissing the idea later on, ATDD is a powerful technique used within the *Behaviour-Driven Development* (BDD[2]) philosophy, an agile software development process that recommends conversation and collaboration across disciplines, that is, among the developers, product owner and other stakeholders. ATDD is sometimes substituted with *Story Test-Driven Development* since acceptance criteria attached to stories are translated into a semi-formal language that enables automation testing from within the code itself. An example of such language is *Gherkin* using the general template "`Given, When, Then`" to describe user acceptance scenarios.

[1]See `https://www.agilealliance.org/glossary/atdd/`
[2]See `https://cucumber.io/docs/bdd/`

1.3 Acceptance testing and *Cucumber*

Cucumber[3] is a test automation tool that implements the BDD philosophy. Put in simple terms, *Cucumber* reads the acceptance tests written in the *Gherkin* language, a human-readable template-based language, and runs the acceptance tests. *Cucumber* is not synonymous of ATDD or BDD, but one of the available frameworks to automate your acceptance tests. Other ATDD tools exists, such as FitNesse[4] or to some degree, Selenium[5] and Cypress[6].

2 Domain, stories, and acceptance criteria

This Lab will build on the example case used in *Lab 4 - Domain modelling and Java Persistence API*. Please refer to that handout if you need to. We reproduce its domain model in Figure 1.

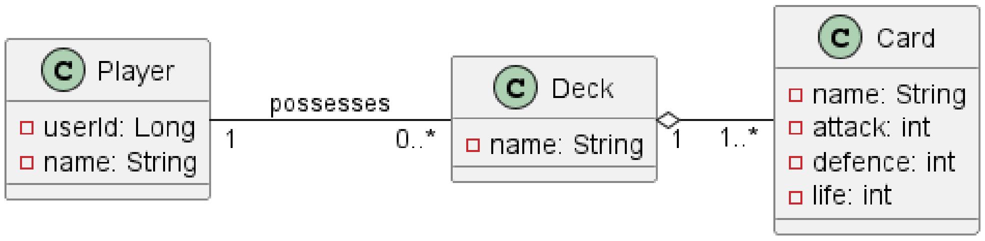

Figure 1: Domain model of a card battler App

2.1 Personae

To keep it simple, we modelled only one Persona.

Name Alex

Age 17

Description Alex is a young person with particular interest in card games, including Yu-Gi-Oh. They like to play with real cards (they own a lot, including rare and expensive ones), but over the latest years, they also started to play online. They are interested in developing an open-source version of Yu-Gi-Oh that is not tied to a specific platform or game console, to make the game more inclusive and attractive.

2.2 User stories

- U1 As *Alex*, I want to create a card so that I can use it in a deck.
  - AC.1 A card has a unique non-empty name, a strictly positive attack, a strictly positive defence, and positive life stats.
  - AC.2 A card name cannot contain non-alphanumeric or numeric-only values.
  - AC.3 A card cannot have a negative values for the attack, the defence, or life stats.
- U2 As *Alex* I want to create a deck so that I can keep track of my cards and use them to battle.
  - AC.1 A deck has a unique, non-empty alphanumeric name.
  - AC.2 A deck cannot have a numeric-only name.
  - AC.3 A deck must contain at least one card.

[3] See `https://cucumber.io/docs/cucumber/api/`
[4] See `http://www.fitnesse.org/`
[5] See `https://selenium.dev/`
[6] https://www.cypress.io/

3 Set up *Cucumber* with *Gradle*

Next to this handout, you will find a `.zip` archive with an updated version of the expected result from lab 3. In a nutshell, the project is structured as follows:

- **`main`** contains the sources of your project
  - **`java`** java source files including the `model` and `accessor` packages
  - **`resources`** application configuration files (i.e. `hibernate.cfg.xml` with *sqlite*)
- **`test`** contains all test sources
  - **`java`** java test sources
  - **`resources`** test configuration files (i.e. `cucumber.properties` and `hibernate.cfg.xml` with *h2*)
- **`build.gradle`** the dependencies configuration file

Compared to lab 3, two different `hibernate.cfg.xml` files are supplied to the application that you can uses interchangeably. One uses a *sqlite* database (under `main` package), the other uses an *h2* database (under `test`).

3.1 Add *Cucumber* dependencies

Take a look at the `build.gradle` file we have set up for you. It is a complete version from lab 4 (with *Hibernate*, *h2* and *sqlite* dependencies filled in). You will need to add *Cucumber* dependencies (line 31-32) and related task (line 40-62), as shown in Listing 1 where we reproduce the expected full `build.gradle` configuration file.

```
1  plugins {
2      // Apply the application plugin to add support for building a CLI application in Java.
3      id 'application'
4  }
5
6  repositories {
7      // Use Maven Central for resolving dependencies.
8      mavenCentral()
9  }
10
11 dependencies {
12     // Use JUnit Jupiter for testing.
13     testImplementation 'org.junit.jupiter:junit-jupiter:5.9.2'
14
15     // This dependency is used by the application.
16     implementation 'com.google.guava:guava:30.1.1-jre'
17
18     // use hibernate to persist (java) domain entities for us, aka JPA implementation
19     implementation 'org.hibernate:hibernate-core:6.1.7.Final'
20
21     // Sqlite as persistent DB
22     implementation 'org.xerial:sqlite-jdbc:3.40.1.0'
23     implementation 'org.hibernate.orm:hibernate-community-dialects:6.1.7.Final'
24
25     // use a in-memory database to store entities (can be substituted with any database)
26     implementation 'com.h2database:h2:2.1.214'
27
28     //Cucumber dependencies
29     testImplementation 'io.cucumber:cucumber-java:7.11.1'
30
31 }
32
33 application {
34     // Define the main class for the application.
35     mainClass = 'uc.seng301.cardbattler.lab5.App'
36 }
37
38 configurations {
39     cucumberRuntime {
40         extendsFrom testImplementation
41     }
42 }
43 
44 sourceSets {
45     test {
46         resources.srcDirs = ['src/test/resources']
47         java.srcDirs = ['src/test/java', 'src/main/java']
48         runtimeClasspath = project.sourceSets.main.compileClasspath +
49                 project.sourceSets.test.compileClasspath +
50                 fileTree("src/test/resources/test") +
51                 project.sourceSets.test.output + project.sourceSets.main.output
52     }
53 }
54 
55 task cucumber() {
56     dependsOn assemble, testClasses
57     doLast {
58         javaexec {
59             main = "io.cucumber.core.cli.Main"
60             classpath = configurations.cucumberRuntime  + sourceSets.test.output
61             args = ['--plugin', 'pretty', '--glue', 'uc.seng301.cardbattler.lab5.cucumber', 'src/test/resources/uc/seng301/
                 cardbattler/lab5/']
62         }
63     }
64 }
65 
66 tasks.named('test') {
67     // Use JUnit Platform for unit tests.
68     useJUnitPlatform()
69     // force the test task to run Cucumber too
70     finalizedBy cucumber
71 }
```

Listing 1: Additional configuration settings in `build.gradle` file.

line 29 the cucumber dependency

lines 38-42 this `configurations` section declares the `cucumberRuntime` as a testing configuration.

line 44-53 this enables to use a separate `hibernate.cfg.xml` file either to run or test the application. Compare both configuration files, you will notice that one uses *sqlite*, the other `h2`. It is good practice to use a non-persisting database for tests.

lines 55-64 specify a new task named `cucumber` that can be invoked by typing `$ ./gradlew cucumber` and will run the *Cucumber* tests for you. Because that task is a java class, it uses the `javaexec` engine (i.e. `java` command) with the parameters defined in line 63. These parameters are specifying:

- a *"pretty"* output format to display the results of the *Cucumber* tests;
- the `gradle.cucumber` package where the acceptance tests will be placed;
- the `src/test/resources` folder where the `cucumber.properties` are.

lines 66-71 we extend the `test` task to always run the `cucumber` task too.

3.2 Further configurations for *Cucumber*

You can take a look into the file `app/src/test/resources/cucumber.properties`. It contains a unique line to ask *Cucumber* to not print additional messages regarding the online publishing of test results. Additional properties can be set in that file, if need be (e.g., defining object factories for custom type mapping if you want to make *Cucumber* aware of more advanced types as test case parameters).

3.3 Run *Cucumber*

To make sure you have set up your `build.gradle` file correctly, run the `cucumber` task. To this end, execute the task by running the command `$ ./gradlew cucumber`. In *IntelliJ*, you may need to ask to refresh the configuration for *IntelliJ* to pick up the configuration changes. Click on the small arrow icon visible inside the `build.gradle` file (close to the *elephant* icon somewhere at the top right of the opened file editor), or via the same button in the `gradle` tool window in *IntelliJ* (accessible via the *"gradle"* button on the extreme right, close to the top). You should get a similar output as Listing 2.

```
> Task :app:cucumber
Mar 13, 2023 10:33:54 AM io.cucumber.core.runtime.FeaturePathFeatureSupplier get
WARNING: No features found at file:/.../uc-seng301-cardbattler-lab5/app/src/test/resources/uc/seng301/cardbattler/lab5/

0 Scenarios
0 Steps
0m0.114s

Deprecated Gradle features were used in this build, making it incompatible with Gradle 8.0.
You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own
     scripts or plugins.
See https://docs.gradle.org/7.4/userguide/command_line_interface.html#sec:command_line_warnings

BUILD SUCCESSFUL in 8s
9 actionable tasks: 8 executed, 1 up-to-date
```

Listing 2: First execution of `cucumber` task with no scenario yet.

Important notes:

- If your build fails with a message like *"It is currently in use by another Gradle instance"*, you may need to run `$ find ~/.gradle -type f -name "*.lock" -delete` since the lab machines are currently facing some disk access issues. After deleting these `lock` files, you should be able to rerun the task successfully.
- if you receive an error like "`Error: Could not find or load main class org.gradle.wrapper.GradleWrapperMain Caused by: java.lang.ClassNotFoundException: org.gradle.wrapper.GradleWrapperMain`", this means you miss `gradle`'s `jar` library. You can simply run `$ gradle wrapper` (not `./gradlew`) to fetch and setup the missing library (if you have `gradle` installed, e.g., on a lab machine).
- if you receive an error like "`bash: ./gradlew: Permission denied`", you can also run `$ gradle wrapper` (to fetch again an executable version) or `$ chmod u+x gradlew` (to make the `gradlew` script executable).

4 Writing acceptance tests with *Cucumber*

You will now learn how to set up a first acceptance test scenario.

4.1 Set up your first `.feature` file

As we defined in Listing 1, *Cucumber* will look under the `cucumber` package (under `test`) to find *Cucumber* features and tests.

1. You need to create folders matching your package names i.e. `uc/seng301/cardbattler/lab5` under `app/src/test/resources/`.
2. Inside that `lab5` folder, create a `cucumber` folder.
3. Create a file named `u1-card.feature` (as a new generic file), **filling the extension by yourself**.

Make sure you follow the naming convention for `feature` file names[7].

4.2 Defining the first scenario

In the `feature` file you just created, you can add the following test description, reproduced in Listing 3.

```
Feature: U1 As Alex, I want to create a card so that I can use it in a deck.
  Scenario: AC1 A card has a unique non-empty name, a strictly positive attack, a strictly positive defence, and positive
       life stats.
```

[7]This is the same convention you will use in SENG302.

```
3     Given There is no card with name "Minotaur"
4     When I create a card named "Minotaur" with attack: 60, defence: 50, life: 110
5     Then The card is created with the correct name, attack, defence, and life
```

Listing 3: First acceptance test scenario (U1-AC1).

A `feature` file can contain many scenarios and each scenario is composed of a description and some *"steps"*.

line 1 A feature file starts with the name of the feature. **As a convention in SENG301/302**, we use the story number (i.e. U1) and a shortened version of the story description.

line 2 Each acceptance criteria (AC) has its corresponding `Scenario`. Similarly to `Features` referring to the user story number, **we prefix the scenario description with the id of the AC** followed by a dash.

line 3 The `Given` clause denotes **preconditions** to your acceptance test.

line 4 The `When` clause expresses what is executed, the actual action that you test in this scenario.

line 5 The `Then` clause shows what are the **postconditions** (or effect) of the execution of the action above.

4.3 Run the (non implemented yet) scenario

After saving your file, you can run the `$ ./gradlew cucumber` task and should be facing a similar output as presented in Listing 4.

```
$ ./gradlew cucumber
> Task :app:cucumber FAILED

Scenario: AC.1 A card has a unique non-empty name, a strictly positive attack, a strictly positive defence, and positive life stats. # src/test/resources/uc/seng301/cardbattler/lab5/cucumber/u1-card.feature:2
Mar 13, 2023 10:40:54 AM org.hibernate.engine.jdbc.dialect.internal.DialectFactoryImpl logSelectedDialect
INFO: HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
  Given There is no card with name "Minotaur"
  When I create a card named "Minotaur" with attack: 60, defence: 50, life: 110
  Then The card is created with the correct name, attack, defence, and life

Undefined scenarios:
file:///.../uc-seng301-cardbattler-lab5/app/src/test/resources/uc/seng301/cardbattler/lab5/cucumber/u1-card.feature:2 # AC.1 A card has a unique non-empty name, a strictly positive attack, a strictly positive defence, and positive life stats.

1 Scenarios (1 undefined)
3 Steps (2 skipped, 1 undefined)
0m1.461s

You can implement missing steps with the snippets below:

@Given("There is no card with name {string}")
public void there_is_no_card_with_name(String string) {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}

@When("I create a card named {string} with attack: {int}, defence: {int}, life: {int}")
public void i_create_a_card_named_with_attack_defence_life(String string, Integer int1, Integer int2, Integer int3) {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}

@Then("The card is created with the correct name, attack, defence, and life")
public void the_card_is_created_with_the_correct_name_attack_defence_and_life() {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}

FAILURE: Build failed with an exception.

[ ... trace truncated ... ]
```

Listing 4: Execution trace of `cucumber` task with unimplemented features.

As you can see, *Cucumber* warns you that it found a Scenario (line 11 and following), but that it is undefined. It also prints skeleton code (lines 21-37) that you will copy-paste into a Java class to implement (i.e. simulate) this acceptance criteria.

For every step present in your *Cucumber* test (i.e. any of `@Given`, `@When` or `@Then`), a method is created with its textual description inside the annotation. You can also see that elements defined between quotes are mapped to methods parameters (e.g., line 19 where *"Minotaur"* has been transformed to a `String` parameter). Similar behaviour can be seen for integer values on line 27.

4.4 Creating the `Feature` class

To prepare the java class that will receive the skeleton code, you need to:

- Create a folders to match packages from your `main` which should give you `app/src/test/java/uc/seng301/cardbattler/lab5/cucumber`.
- Create a folder named `cucumber` under that new `lab5` folder.
- Create a Java class named `CreateNewCardFeature.java`

You can now copy-paste the skeleton code from the `cucumber` task (lines 21-37 in Listing 4) into the Java class you created in previous step and rerun the `cucumber` task. You will see that now, *Cucumber* tells you that the scenario is *"Pending"* and not *"Undefined"* with a similar (truncated) trace as Listing 5.

```
$ ./gradlew cucumber

[ ... trace truncated ... ]

Pending scenarios:
file:///.../uc-seng301-cardbattler-lab5/app/src/test/resources/uc/seng301/cardbattler/lab5/cucumber/u1-card.feature:2 # AC.1
    A card has a unique non-empty name, a strictly positive attack, a strictly positive defence, and positive life stats.

1 Scenarios (1 pending)
3 Steps (2 skipped, 1 pending)
0m0.219s

[ ... trace truncated ... ]
```

Listing 5: Execution trace of `cucumber` task with unimplemented features.

4.5 Implement the acceptance test for AC1

You can now implement this acceptance test in a similar fashion as Listing 6. Imports have been left out, but are composed of `io.cucumber.java` subpackages or classes, `org.hibernate` (see lab 4), `org.junit.jupiter.api.Assertions`, your `model` classes and the new `Accessor` classes (from the code base you received).

```java
public class CreateNewCardFeature {
    private CardAccessor cardAccessor;
    private Card card;

    private String cardName;
    private Integer cardAttack;
    private Integer cardDefence;
    private Integer cardLife;
    private Exception expectedException;

    @Before
    public void setup() {
        java.util.logging.Logger.getLogger("org.hibernate").setLevel(Level.SEVERE);
14          Configuration configuration = new Configuration();
15          configuration.configure();
16          SessionFactory sessionFactory = configuration.buildSessionFactory();
17          cardAccessor = new CardAccessor(sessionFactory);
18      }
19
20      @Given("There is no card with name {string}")
21      public void there_is_no_card_with_name(String cardName) {
22          Assertions.assertNull(cardAccessor.getCardByName(cardName));
23      }
24
25      @When("I create a card named {string} with attack: {int}, defence: {int}, life: {int}")
26      public void i_create_a_card_named_with_attack_defence_life(String cardName, Integer cardAttack, Integer cardDefence,
27              Integer cardLife) {
28          this.cardName = cardName;
29          this.cardAttack = cardAttack;
30          this.cardDefence = cardDefence;
31          this.cardLife = cardLife;
32          Assertions.assertNotNull(cardName);
33          Assertions.assertTrue(cardAttack > 0);
34          Assertions.assertTrue(cardDefence > 0);
35          Assertions.assertTrue(cardLife >= 0);
36      }
37
38      @Then("The card is created with the correct name, attack, defence, and life")
39      public void the_card_is_created_with_the_correct_name_attack_defence_and_life() {
40          card = cardAccessor.createCard(cardName, cardAttack, cardDefence, cardLife);
41          Assertions.assertNotNull(card);
42          Assertions.assertEquals(cardName, card.getName());
43          Assertions.assertEquals(cardAttack, card.getAttack());
44          Assertions.assertEquals(cardDefence, card.getDefence());
45          Assertions.assertEquals(cardLife, card.getLife());
46      }
47  }
```

Listing 6: Code snippet of the implementation of the acceptance test for AC1 in *Cucumber*.

lines 11-18 you can declare some code that will be executed before running the `feature` class (i.e. the scenario). These are called hooks[8] and have a similar semantics to the ones from *JUnit*. In this part, we need to instantiate the *Hibernate* features. It's important to note that the `@Before` annotation is from the `io.cucumber` package.

line 22 *Cucumber* reuses the assertion mechanisms from *JUnit*, i.e. from `org.junit.jupiter.api`.

lines 21,26 Note that, for readability reasons, we have renamed the default names for parameters to meaningful names and **we expect you to keep this readability habit**.

5 Implementing the acceptance test for AC2

Cucumber allows for parameterization of acceptance tests, where we can provide a table `Examples` in our feature file.

In the `u1-card.feature` file created above add the following test description, reproduced in Listing 7.

```
  Scenario Outline: AC2 - A card name cannot contain non-alphanumeric or numeric-only values.
    Given There is no card with name <name>
    When I create an invalid card named <name> with attack: 60, defence: 50, life: 110
    Then An exception is thrown
    Examples:
      | name      |
      | "673975"  |
      | "$*&ynsl" |
```

Listing 7: Second acceptance test scenario (U1-AC2) with (partial) examples.

[8]See https://cucumber.io/docs/cucumber/api/#hooks

line 1 To use parameterized testing we create a `Scenario Outline`.

lines 2-3 we reference variables in the `Example` table using the column name surrounded by carets. In this case `<name>`.

line 5-8 Here we use the `Examples` keyword to create the parameterized data denoted by `<name>`. In this scenario we only have one column, however you can create as many as you want as long as each column maps to a referenced variable in the scenario[9].

Now add the new step definitions to your `CreateNewCardFeature` file, reproduced in Listing 7.

```java
@When("I create an invalid card named {string} with attack: {int}, defence: {int}, life: {int}")
public void i_create_an_invalid_card_named_with_attack_defence_life(String cardName, Integer cardAttack,
        Integer cardDefence,
        Integer cardLife) {
    expectedException = Assertions.assertThrows(IllegalArgumentException.class,
            () -> cardAccessor.createCard(cardName, cardAttack, cardDefence, cardLife));
}

@Then("An exception is thrown")
public void an_exception_is_thrown() {
    Assertions.assertNotNull(expectedException);
}
```

Listing 8: Code snippet of the implementation of the acceptance test for AC2 in *Cucumber*.

5.1 Run the full scenario

By running the above scenario with `$ gradlew cucumber`, you should have a similar trace as Listing 9

```
1  [ ... trace with Hibernate logs removed truncated ... ]
2  Scenario: AC.1 A card has a unique non-empty name, a strictly positive attack, a strictly positive defence, and positive life
         stats. # src/test/resources/uc/seng301/cardbattler/lab5/cucumber/u1-card.feature:2
3  Given There is no card with name "Minotaur"                                                  # uc.seng301...
4    When I create a card named "Minotaur" with attack: 60, defence: 50, life: 110              # uc.seng301...
5    Then The card is created with the correct name, attack, defence, and life                  # uc.seng301...
6
7  Scenario Outline: AC2 - A card name cannot contain non-alphanumeric or numeric-only values. # src/test/resources/uc/seng301/
        cardbattler/lab5/cucumber/u1-card.feature:13
8  Given There is no card with name "673975"                                                    # uc.seng301...
9    When I create an invalid card named "673975" with attack: 60, defence: 50, life: 110       # uc.seng301...
10   Then An exception is thrown                                                                # uc.seng301...
11
12 Scenario Outline: AC2 - A card name cannot contain non-alphanumeric or numeric-only values. # src/test/resources/uc/seng301/
        cardbattler/lab5/cucumber/u1-card.feature:13
13 Given There is no card with name "673975"                                                    # uc.seng301...
14   When I create an invalid card named "673975" with attack: 60, defence: 50, life: 110       # uc.seng301...
15   Then An exception is thrown                                                                # uc.seng301...
16
17 3 Scenarios (3 passed)
18 9 Steps (9 passed)
19 0m2.066s
20
21 [ ... trace truncated ... ]
```

Listing 9: Output trace from *Cucumber* with passing scenarios.

line 8, 13 We can see here that AC2 is ran twice, once for each entry in the `Examples` table.

[9] See `https://cucumber.io/docs/gherkin/reference/#examples` for more details.

6 It's your turn now

6.1 Implement U1 AC3

You are required to write the acceptance test scenario for U1 AC3, this should use only the existing steps in the `CreateNewCardFeature` file. You should use `Scenario Outline` as done above to test the following value combinations for attack, defense, and life:

attack	defense	life
100	0	100
-100	10	100
100	10	-100

6.2 Implement U2

Now you are required to implement U2 including:

- Adding validation code to `DeckAccessor.createDeck()`.
- Creating the `u2-deck.feature` file and Scenarios (or Scenario Outlines where appropriate) for ACs 1-3.
- Creating the required step definitions in a `CreateNewDeckFeature` file.

Here are the ACs again:

U2 As *Alex* I want to create a deck so that I can keep track of my cards and use them to battle.

AC.1 A deck has a unique, non-empty alphanumeric name.

AC.2 A deck cannot have a numeric-only name.

AC.3 A deck must contain at least one card.

Follow the same step-by-step approach as described from Section 4.3. Because we want you to work in a TDD-way (write tests first), you will need to refactor the existing code in `DeckAccessor.createDeck()` to apply the appropriate validation. We also introduce Gherkin's `And` to combine multiple steps.