VU Qualitätssicherung in der Softwareentwicklung
LV-Nr. 716.066, SS 2021

Practical Task 1

ao. Univ.-Prof. Dr. Bernhard Aichernig
Alexander Grossmann, David Wildauer
qs@ist.tugraz.at

Graz, 23.03.2021

Submission and Technical Details

Due to: Monday 20.04.2021 23:59
Place of submission: TeachCenter Course

For Practical Task 1, you should submit a file containing all of your tests to the TeachCenter in the *Exercises* section. Only one file, **MessageBoardTests.java**, should be submitted, as tests will be performed on the unchanged framework with your submission of *MessageBoardTests.java* instead of the default one.

You are free to change the framework if you want, but note that tests are executed on the original version of the framework.

In case of abnormal group participation, i. e. some students had to do all the work themselves, they should send an email to the study assistants and we will decide whether students can get bonus points, or if other students should have their points deducted.

Tools

The only thing you need to do the task is the Java Development Kit, Version 8 or later. We will use `openjdk-8-jdk-headless` on Ubuntu to test your submission. The project uses **Gradle** 6.2.2 as a build automation system. Here is a list of useful commands. On Windows, you need to use `gradlew.bat` instead of `./gradlew`.

- `./gradlew clean` – removes the *buildDir* folder, thus cleaning everything
- `./gradlew build` – builds the project
- `./gradlew test` – execute the tests
- `./gradlew jacocoTestReport` – check test coverage, results in *build/reports/jacoco*
- `./gradlew pitest` – start mutation testing, results in *build/reports/pitest*

Gradle is configured using *build.gradle* in the project root folder. To speed up mutation testing, you can change the number of threads for pitest according to your systems capabilities.

Using Gradle, the project adds the following dependencies and plugins:

- JUnit 4.13
- JaCoCo Java Code Coverage Library
- Pitest 1.5.0

The first practical task consists of two parts, code coverage, and mutation coverage.

1 Code Coverage

The Goal of code coverage is to write unit tests that will achieve 100 % instruction and branch coverage. Instruction coverage ensures that each instruction of the code is executed at least once, whereas branch coverage aims to ensure that each one of the possible branches from each decision point are reached at least once and thereby ensuring that all reachable code is executed.

All unit tests have to be **meaningful**, i. e. they should check the expected result. (`Assert.isTrue(true)` would not be a meaningful check.)

To achieve 100 % code and branch coverage, it could be useful to derive helper classes. One example, `TestClient`, is given in *MessageBoardTests.java*. Note that all helper classes used for testing should be in that file, those classes shall be package-private.

For this part, **JaCoCo** is used to generate the desired results. Before generating reports, you need to run the tests. Therefore, you always need to execute the following commands to get a fresh result.

1. `./gradlew test`
2. `./gradlew jacocoTestReport`

Then you can find a HTML report in *./build/reports/jacoco/test/html/index.html*.

2 Mutation Coverage

To further evaluate the quality of your unit tests we use mutation testing. Mutation testing is a technique that evaluates the quality of tests by changing the original program in small ways and checking if your tests can detect the change.

You do not need to write separate unit tests for mutation coverage, the same tests are used for both parts. Therefore, the same rules apply to this part, too (meaningful checks, inheritance).

The tool used in this part is **Pitest**. To generate a report, you need to run `./gradlew pitest`. The report found in *build/reports/pitest/index.html* could be useful to find details why you were not able to kill some mutants. Watch out for equal mutants.

Pitest allows selecting a set of mutators. We chose the set called *STRONGER* in *build.gradle*. Unfortunately, you need to dig into the source code of pitest to find out which sets are available and which mutators they include.

3 Grading

The following requirements need to be met to achieve the total of 15 points:

- 100 % instruction and branch coverage,
- 85 % mutants killed,
- all tests need to include meaningful asserts and
- all tests need to succeed.

4 System Description

An overview of the application domain is described here. A more detailed description of the classes and methods can be found as documentation comments in the source code files. The corresponding implementation can be assumed to be correct. The system to be tested is a simple message board, which is implemented as a (simulated) actor system. The project is divided into two parts: the part implementing the actor system and the part implementing the message board functionality. This division between the actor system and message board can be seen in the package structure of the project.

Actor System The simulated actor system offers functionalities inspired by the actor model [1]. Basically, such systems consist of parallel acting actors, which communicate with each other by exchanging messages.
In the implementation under test, actors are represented by objects of the class `SimulatedActor`. The most important method of actors is `receive`, which contains the logic for the behavior of actors. The actors are managed by an instance of the class `SimulatedActorSystem`, which simulates the passing of time. For this, the method `SimulatedActor.tick()` is called to signal actors the passing of a time unit. Time has two different effects in our actor system. On the one hand, a message needs a certain number of time units after sending to reach an actor. This is simulated by the `CommunicationChannel` class. Secondly, actors need time to process messages. This time is defined by the method `Message.getDuration()`. `Message` is an interface that must be implemented by all messages.

Message Board The message board offers clients the ability to post short messages, "like" and "dislike" messages from others, and receive messages from a specific author. Clients can also report other clients for violations, which can lead to the reported client being banned. The system is divided into four different actor classes. There is a `Dispatcher` actor, several `Worker` actors, a `MessageStore` actor which is responsible for persistence and several `WorkerHelper` actors. Communications between a client and the system take place according to the following pattern:

- Client sends `InitCommunication` message to the Dispatcher
- Dispatcher chooses one worker and forwards the message to it
- Worker acknowledges the start of the communication
- Client sends messages to the Worker, which can be of type `Publish` to post something, `RetrieveMessages` to retrieve messages, `SearchMessages` to search for messages, `Like/Dislike` to like or dislike a message or `Report` to report another client

[1] http://en.wikipedia.org/wiki/Actor_model

- Client sends `FinishCommunication` to the worker
- Worker acknowledges this message and the communication is terminated.

Depending on the messages sent by the client, the workers perform various checks and communicate with the `IMessageStore` actor via a `IWorkerHelper` auxiliary actor. In the framework you will find a rudimentary test, which contains a communication setup and shutdown between a client and the system. However, the code coverage achieved by this test does not add to the points achieved.