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

Practical Task 2

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

20.04.2021, Graz

Submission

Due to: Tuesday, 18.05.2021 23:59
Place of submission: TeachCenter Course

For Practical Task 2, you should submit your files containing all of your tests to the TeachCenter in the *Exercises* section. Make sure that your files compile with the unmodified framework.

Please hand in the following files:

- `SimpleFunctionsTest.scala`
- `MessageBoardProperties.scala`
- `MessageBoardSpecification.scala`
- `MutantDescription.md`
- `MutantSequences.txt`

[0] In case of abnormal group participation, ie. some students had to do all the work themselves, they should send an email to 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 9 or later. We will use `openjdk-11-jdk-headless` on Ubuntu to test your submission.

The project uses **Gradle** 6.2.2 as 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 all local tests
- `./gradlew testSimpleFunctions` – execute all tests for *SimpleFunctions*
- `./gradlew testMBProperties` – execute proberty-based tests for the *MessageBoard*
- `./gradlew testMBModel` – execute model-based tests for the *MessageBoard*
- `./gradlew testMBModelMutants` – execute model-based tests on remote system with mutants
- `./gradlew pitest` – total mutation testing, results in *build/reports/pitest*
- `./gradlew pitestSimpleFunctions` – mutation testing for *SimpleFunctions*, results in *build/reports/pitestSimpleFunctions*
- `./gradlew pitestMBProperties` – mutation testing for *MessageBoard* using property-based tests, results in *build/reports/pitestMBProperties*
- `./gradlew pitestMBModel` – mutation testing for *MessageBoard* using model-based tests, results in *build/reports/pitestMBModel*

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 system's capabilities.

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

- Scala 2.13.1
- ScalaTest 3.1.1
- ScalaCheck 1.14.3
- JUnit 4.13
- Pitest 1.5.1

1 Introduction to Property-based Testing

The goal of this introductory task is to become familiar with the property-based testing of Scala/Java programs with ScalaCheck. The following functions are already implemented and you have to create properties which ensure their correct behavior. You can find them and their documentation in the *SimpleFunctions.scala* file:

- `insertionSort(xs: List[Int]): List[Int]`
  Sorts the list of integers ascending.
- `max(xs: List[Int]): Int`
  Returns the largest element. Must be called with non-empty list as input.
- `minIndex(xs: List[Int]): Int`
  Returns the index of the smallest element. Must be called with non-empty list as input.
- `symmetricDifference(xs: List[Int], ys: List[Int]): List[Int]`
  Returns a set of elements that are not in both lists. The two input lists should be sets/distinct.
- `intersection(xs: List[Int], ys: List[Int]): List[Int]`
  Returns a set of elements that are in both lists. The two input lists should be sets/distinct.
- `smallestMissingPositiveInteger(xs: List[Int]): Int`
  Returns the smallest integer > 0 that is not in the list.
  E.g. for input $[3, 4, -1, 1]$ it would return 2.

Please write your properties in the file *SimpleFunctionsTest.scala*. You can run your tests with `./gradlew testSimpleFunctions` and find your results in *build/reports/tests/testSimpleFunctions/index.html*. It should integrate nicely in your IDE, too.

To achieve all points, you should reason about each function as shown in lectures and in the demo, and write meaningful properties. Pay attention that one property cannot be inferred from another, as then if the more general property holds, so will more specific one, and the second one is nonsensical.

We provided example properties for `insertionSort`. There are also properties to show that it does not matter if you test Scala or Java. But please stick to the Scala version for your properties.

You can find good tutorials for ScalaCheck in their User Guide on GitHub.

Hint: To test your solution with your own mutants, you can just duplicate *SimpleFunctions.scala* to e.g. *SimpleFunctionsMutant1.scala* and change the appropriate import in your *SimpleFunctionsTest.scala*. You can also run pitest known from assignment 1, using `./gradlew pitestSimpleFunctions` to see your mutation score. A good mutation score for *SimpleFunctions.scala* would be $> 90\%$. Pay attention: The gradle pitest task shows an error message if you do not reach the configured coverage.

2 MessageBoard Testing

The goal of the second part is to test the following five commands of the `MessageBoard` using property-based and model-based testing.

2.1 Commands

- `Publish(String author, String message)`:
  This command should publish a message with the given author and message content. The System-Under-Test (SUT) should be successful if the system responds with `OperationAck` to `Publish`. If the system responds with `OperationFailed` or `UserBanned`, the SUT was not successful. Check in the post-condition whether the model is also successful or unsuccessful. If the SUT behaves differently than the model, the post condition should fail. If it behaves the same, the post-condition is not violated.
- `RetrieveMessages(String author)`:
  This command should retrieve all messages of the given author *author*. You can use the string representation in the implementation (`UserMessage.toString()`) and in the model (`ModelUserMessage.toString()`) to compare the individual messages.
- `SearchMessages(String searchText)`:
  This command should search and retrieve all messages containing the given *searchText* in the message itself or the author.
- `Like(String clientName, long messageId)`:
  This command should like the message with id *messageId* as user *clientName*. The SUT will fail if the messageId does not exist or if the system responds with `OperationFailed` or `UserBanned`. If the system replies with `OperationAck`, the SUT was successful.
- `Dislike(String clientName, long messageId)`:
  Similar to the command Like, this command should dislike the message with id *messageId* under the name *clientName*.
- `Report(String reporterName, String reportedName)`:
  This command should report the specified author *reportedName* by the user *reporterName*. If the system responds with `OperationFailed` or `UserBanned`, the SUT was not successful, but if it responds with `OperationAck`, the SUT was successful.

2.2 Requirements of Commands

The focus of the task is functional requirements. So you do not need to check details like the time a system needs to respond. Here is a list of all requirements to consider:

R1 A message may only be stored if its text contains less than or exactly `MAX_MESSAGE_LENGTH` (= 10) characters. This check is performed in the `Worker` class.

R2 A message may only be saved if no identical message has been saved yet. Two messages are identical if both author and text of both messages are the same.

R3 A message may only be liked/disliked if it exists.

R4 A message may only be liked/disliked by users who have not yet liked/disliked the corresponding message.

R5 It should be possible to retrieve a list of all existing messages of an author.

R6 It should be possible to search for messages containing a given text and get back list of those messages.

R7 A user may report another user only if he has not previously reported the user in question.

R8 If a user has been reported at least `USER_BLOCKED_AT_COUNT` (= 6) times, he/she cannot send any further *Publish*, *Like*, *Dislike* or *Report* messages.

R9 Successful requests should be confirmed by sending `OperationAck`. Requests are considered successful when a message has been saved, a Like or Dislike has been added to a message, or a report for an author has been added.

R10 Requests that are not successful should be confirmed by sending `OperationFailed` or `UserBanned`.

You can assume that the system will always respond. If you send messages of type `ClientMessage` to the system, messages of type `Reply` (or `FoundMessages`) are sent back after a finite number of time units.

The focus on functional requirements results in further simplification. There is no need to test how the system behaves when several requests are made simultaneously.

This results in the following proposed structure to test the SUT:

1. Initialize communication between the system and a test client (object of the class `TestClient`)
2. Communication to implement the required functionality of the commands
3. Evaluation of system responses
4. Termination of the communication

2.3 Property-based Testing

In this part, you should write ScalaCheck properties to ensure the requirements listed above. Please write them in the file *MessageBoardProperties.scala*. There you also find an example.

You can run your tests with `./gradlew testMBProperties` and find your results in *build/reports/tests/testMBProperties/index.html*. It should integrate nicely into your IDE, too.

Our suggestion is to use the example shown in the demo video as a skeleton which you more or less reuse in all examples. It is provided with the framework. You will need to use custom generators like `validMessageGen`. More information about them can be found in the ScalaCheck User Guide and on `https://www.scala-exercises.org/scalacheck/generators`.

Of course, you can run pitest here, too, by using `./gradlew pitestMBProperties`. A good mutation coverage for the MessageBoard would be $> 50\%$.

2.4 Model-based Testing

The MessageBoard is to be regarded simply as a database containing messages. Abstractly such a database can be modeled as a set of messages.

For model-based testing, we use the so-called "Stateful Testing" feature of ScalaCheck. More details on how to use it are given in the UserGuide of ScalaCheck. Instead of the term "model", ScalaCheck uses the term "state".

In this exercise, we use the model class `ModelMessageBoard`, which consists of a list of `ModelUserMessage`s, a list of `ModelReport`s, a boolean whether the last command was successful and a second boolean to declare if the user of the last command is banned.

To reduce the amount of research for you, we already provided stubs for all commands in the file *MessageBoardSpecification.scala*. For every command you can find a Generator (of type `Gen[XyzCommand]`) and a case class `XyzCommand extends Command`. In this case, class three methods are important for you:

- `run(sut: SUT): Result`:
  Here you execute the command on the System-Under-Test. You should return a result that can be checked in the post-condition. We already suggested result types, but you are allowed to change them.
- `nextState(state: State): State`:
  Here you execute the command on the model. Please pay attention: The model has to be immutable. You cannot modify the variables of the model directly; instead, you need to use `copy()` to return a modified instance.

- `postCondition(state: State, result: Try[Result]): Prop`:
  Here you need to check if the behaviour of the SUT and the model is the same. Please pay attention: The state passed as the argument is the state **before** the invocation of `nextState()`, so you need to use `nextState(state)` to get the actual model after processing the command.

As an example, we already implemented the ReportCommand. So you do not need to implement that command.

To restrict the number of generated command variation, we restricted the possible strings for *Author*, *Reporter* and *Message* (can be seen in `val genXyz`).

You can run your tests with `./gradlew testMBModel` and find your results in *build/reports/tests/testMBModel/index.html*.

Of course, you can run pitest here, too, by using `./gradlew pitestMBModel`. A good mutation coverage for the MessageBoard would be $> 50\%$.

2.5 Kill mutants – find errors

To test your tests, we will use mutants. You can try to kill some of those mutants by running `./gradlew testMBModelMutants`. The test will then connect to our server, upload your *MessageBoardSpecification.scala* file, and run your model-based tests.

Running the tests for one mutant takes at least 30 to 60 seconds. Please do not test all of your mutants at once many times a day, since this will lead to longer waiting times for all other participants or even an overload of the server. So do not abuse the system. **Otherwise, we need to change to only one test run per group and day.**

There are 5 implementations, maybe not all of them are mutants. Try to kill those mutants and figure out what is wrong. Please provide sequences to kill the mutants in the file *MutantSequences.txt*. Each line in that file starts with the number of the mutant followed by a colon. After that colon you should write the sequence you found (in `ARG_0` of the test output), e.g. `Publish(Bob, msg_w_9ch), Report(Laura, Bob), Like(Bob, msg_w_9ch, Alice)`, or `NO_MUTANT` if the implementation is correct.

Additionally, please describe in the Markdown file *MutantDescription.md* what is wrong with the implementations. To get more information about the things that happen, you can use `println()` etc. The output will be reported back to you.

If you would like to earn a bonus point, you can also try to figure out what's wrong with the two bonus implementations. Therefore, uncomment the methods in *MessageBoardMutantTest.java*. To succeed, you may need to modify the generators (`genAuthor` etc.).

3 Grading (20 Points)

- **9 Points:** Property-based testing
  - **3 Points:** SimpleFunctions
  - **6 Points:** MessageBoard
- **11 Points:** Model-based testing
  - **8 Points:** MessageBoard
  - **3 Points:** Describing errors in 5 implementations