Static and dynamic verification techniques – Exercises

Systems Engineering Laboratory 1

Note: Please treat these exercises also as professional work. For example, use meaningful commit messages.

Note: You should continue with your GitHub repository from the previous lab. If you did not participate in the previous lab let us know.

1 Static techniques

1.1 Code review

1. Clone your repository from the previous lab.
2. The readme of your repository contains the expected behavior of the system. However, changing the reference speed is not implemented yet. If the user changes the position of the joystick, the reference speed does not change (the function is written, but it is not called). Both members of the team should **create their own new branch** and **implement this feature** based on the readme. The periodic change of the reference speed can be implemented with for example timers, threads or with other constructs, it should be your decision.
3. After the implementation is complete, each member should commit their results (on their own branch), **open a pull request** (for their own branch) and request a review from the other team member.
4. Inspect the pull request of the other team member.
5. Go to the **Files changed** tab and **Start a review**.
6. Add comments to specific **source lines** identifying problems or suggesting changes.
7. After adding the comments you can submit your review with the button in the top right corner and select **Request changes**.
8. Your team member should make changes according to the review and add a new commit to the ongoing pull request.
9. If the commit solves the problem, then **Approve** the review.
10. After both reviews are complete, pick **one of two** implementations and **merge** that pull request into the master branch.

For more information see Reviewing changes in pull requests.

1.2 SonarQube

SonarQube is a quality management platform incorporating several functions. It runs different scanners performing code analysis, a database to store results and a web dashboard to view the results.

1.2.1 Getting SonarQube

1. Download the latest version of SonarQube from sonarqube.org/downloads.
2. Extract the downloaded archive file to some location.

1.2.2 Starting SonarQube

1. Go to the folder where SonarQube is extracted and go under `bin`.
2. Select the folder corresponding to your OS and inside that folder and run `./sonar.sh console` (Linux / Mac) or `StartSonar.bat` (Win).
   - On Linux you can use the `uname -a` command to find out whether you are running a 32-bit or 64-bit kernel.
   - On Linux pay attention that the actual folder (.) is not in the path, therefore the command has to be prefixed with `./` as above.
3. It takes time to start all three components of SonarQube Server (Compute engine, Search server, Web server), wait until you see all these three lines in the log:

   ```
   jvm 1    | 2016.09.21 17:38:03 INFO  app[o.s.p.m.Monitor] Process[es]
   is up   ...   jvm 1    | 2016.09.21 17:38:45 INFO  app[o.s.p.m.Monitor]
   Process[web] is up   ...   jvm 1    | 2016.09.21 17:38:59 INFO  app[o.s.p.m.Monitor]
   Process[ce] is up
   ```

4. Open http://localhost:9000/ in a browser to see the (currently empty) SonarQube dashboard.
5. You can log in with the default *admin / admin* credential to configure settings (but it we won't need it for the current exercises).

1.2.3 Running SonarQube

SonarQube can be easily executed from a Gradle build with the following steps.

1. Add the following line to the `.gradle/gradle.properties` file in the root of the project. (Create the folder and the file, if it does not exists. Note, that a `gradle` folder may already exist, but you need a folder named `.gradle`.)

```
systemProp.sonar.host.url=http://localhost:9000
```

2. Add SonarQube as a plugin to the `build.gradle` file in the root. NOTE: this needs to be at the very **top of the file**.

   ```
   plugins {          id "org.sonarqube" version "2.6.2" apply false   }
   ```

3. Modify the line `subprojects {` to be `subprojects { subproject ->` and add the following line below the other plug-in(s).

   ```
   apply plugin: 'org.sonarqube'
   ```

4. Run the analysis using the following command (from the root of the project):

   ```
   ./gradlew sonarqube
   ```

5. Open http://localhost:9000/ in a browser to go to the **dashboard**.
6. Click on of the projects. **Inspect** the bugs, vulnerabilities and code smells.
7. If there are any issues, select one of them that can be fixed quickly and **fix it**.
8. **Run** SonarQube again and inspect the results.

For more information see the SonarQube Gradle documentation.

SonarQube could be integrated in the continuous integration pipeline (e.g. triggered by each commit or called from Travis), but it would require some preparations, that could not fit in the schedule of the current exercise. An alternative solution for automated static analysis is Codacy, which can be configured for open source GitHub repositories with only a few clicks.

2 Unit testing

2.1 Implementing a new feature

Before performing unit testing one of the components in the system, a new feature must be implemented first. The sensor of the train `TrainsSensorImpl` shall have an alarm functionality. The alarm indicates that the difference between the reference speed and the speed limit is too large. Such large difference may occur for example when a wrong speed limit was given.

When the alarm is triggered in the sensor, it must set the user (`TrainUser` and `TrainUserImpl`) to alarm state. This requires two new methods for the interface and the class: `getAlarmState()` and `setAlarmState(boolean alarmState)`, and a private field inside the class. The alarm in the sensor analyzes the value given in its `overrideSpeedLimit()` method. There are two margins to implement, when the alarm has to be triggered.

1. *Absolute margin*: If the new speed limit is under 0, or over 500.
2. *Relative margin*: If the new speed limit is more than 50% slower than the actual reference speed (e.g., 150 to 50 is an alarming situation, because 50 is more than 50% less than 150).

2.2 Unit testing the new feature

1. Design and document at least four different test cases for the `TrainSensorImpl` class, in which the feature under test is the recently implemented alarm.
2. Implement at least four test cases you defined in the previous step into the `TrainSensorTest.java` file that is already created. The file contains only method stubs, thus delete them and use your own test methods with utilizing various features of JUnit. Use mockito to mock the two dependencies of the class: `TrainController` and `TrainUser` (i.e., do not use `TrainControllerImpl` and `TrainUserImpl`). Use verification (e.g., `verify()` and `when()`) in these mocks to ensure that the communication out of the unit under test is well-implemented.
   - The dependencies for JUnit and Mockito should only be added to the `build.gradle` file of the `train-sensor` project.
3. Execute the test cases with JUnit and check their outcome. If there is a failing test case, create a GitHub issue in your repository with the details, and resolve the issue by fixing your implementation. Also, document each step taken for fixing a bug found by a failing test case.

2.3 Measuring code coverage

Measure the code coverage of your tests using the JaCoCo plugin.

1. Add the line `apply plugin: 'jacoco'` to the `build.gradle` file in the root of the project.
2. Execute the `./gradlew build` task.
3. Execute the `./gradlew jacocoTestReport` task.
4. Observe the results under the `build/reports/jacoco/test` folder in each project containing your tests. (Do not confuse it with the folder `build/jacoco`.)
   - If you cannot find the folder check if you executed the `build` task before `jacocoTestReport`.
5. Check the coverage. If some important functionality was not tested, extend your tests and check coverage again.