Assignment 4 - Static Analysis

Version: April 2, 2021

Objectives

- Understand how a static analysis tool helps discover potential defects and other quality issues.
- Understand merging in git.

Setup

This is again an individual assignment. You are required to have a document in which you answer questions and include screenshots.
You will continue to work in your private repository.

Some Git things first (8 points)

Let's talk about Git a little. We deleted the "GivenBlackbox.java" from our Whitebox branch since we did not want our tests to fail, which was a drastic move. What do you think would happen if we (do NOT do this) merged Whitebox into master?
Answer: Your "GivenBlackbox.java" would also be deleted on master. Which is not really what we want, since we want to keep our test files.
So when you work with Git and you change or delete a file in a branch, that might overwrite things in a way which you do not want it to. It depends highly on: who their ancestor is, what changes have been made, which changes are newer, and whether those changes conflict. This will be something that you will run into. Every time you begin working, I advise you to pull the Dev branch and merge it into the branch you are working on. This will reduce the likelihood of complicated merges. And remember, you can always go back in time. So let's look at how you can get your file back (this works for other changes as well).
First switch to your Whitebox branch. Create a new branch off of Whitebox and call it StaticAnalysis. Switch to that branch -we do not want to change Whitebox since that is still being graded. So make sure the branch Whitebox does not change.
Run "git log" in your command line. This will show you a log of all previous commits on this branch and its ancestors. Look at the commit messages. Are your commit messages clear and specific? Can you actually understand what you did by reading them (rhetorical)?. If you cannot, then please change how you write your commit messages in the future. You will probably get a lot of log entries (if you committed often). So let's filter them by deleted files (other filters are also possible).

```
git log --diff-filter=D --summary
```

This should give you the information of any commit where a file was deleted (in our case that should be exactly one – more if you deleted other files – which you might have done to clean your repo).
Now we can checkout the file we want from a specific commit:

```
git checkout COMMITNR~1 src/test/java/GivenBlackBox.java
```

COMMITNR is the long commit number given in the log-entry. This represents where we deleted the file (the '~1' tag is a call to the commit **before** that deletion commit – since we want the file before it was deleted. If you want exactly that version then omit the ~1). This also works for directories if needed. It should look something like this:

```
git checkout b77d051a79624b3e14529c0d330f80b5e79a3d94~1 src/test/java/GivenBlackbox.java
```

If you struggle with this go back to the Git videos from the beginning of class. I talk about these things in there as well.
When you check your explorer/Finder/command line, in your assignment folder you should see the "GivenBlackbox.java" file again.
Now you need to add and commit this file so it is back in your current version of your StaticAnalysis branch.
We still do not want to run our Blackbox test when running *gradlebuild* since our build would fail. Gradle allows us to specify which test files to run. Since we want to update our recent changes in all our branches, we are going to get a run a few steps that are a little "complicated".
Pay attention to what exactly I want you to do and follow it exactly.
Switch to your **Blackbox** branch. Now add the following code to your build.gradle file:

```
test {
    exclude '**/GivenBlackbox.class'
}
```

Run *gradleclean* in your command line. Then, when you run *gradlebuild*, it should not run the BB test class, and your build should be successful (your Whitebox test might still fail here. That is ok).
Now, merge Blackbox into the StaticAnalysis branch.
Running *gradlebuild* on StaticAnalysis should now run all your tests successfully (your Whitebox tests should all pass now) and thus your build should pass. If it does not, you should make sure you correct your tests and code in your StaticAnalysis branch.

Cleaning up (5 points)

Your repository should already be clean at this point, but if it is not, you really need to clean it up now. I no longer want to see temporary files in your GitHub repo when you submit the assignment. This means that any unnecessary hidden files and build files should NOT be in your repo anymore. If you do not know what this means, please refer back to the Git videos and read about what should be in a gitignore file. Make sure you delete them and they are ignored through a good gitignore in your master, Blackbox and StatciAnalysis branch (DO NOT change Review and WhiteBox since we might still be grading these). If you have already done this, then there is nothing further left for you to do in this section; these points are a gift for being well-organized.

Setup for this Assignment (7)

Travis CI (5 points)

Our goal is that our build is always successful and Continuous Integration helps us with this. Let's set up Travis CI for your assignment which will use Gradle to build your assignment code.
Download the given Travis CI file (.travil.yml) and add it to your **master** branch. You will find the file on Canvas with this assignment. Add it to your root directory (same place where your build.gradle file is). Add and commit this file.
Now, we also want Travis CI to use a specific Gradle and Java version.
Gradle has something called Gradle wrapper which can be used to specify the Gradle and Java version. To use it, you need to run "gradle wrapper" in your command line (also check here: `https://docs.gradle.org/current/userguide/gradle_wrapper.html`). This will generate the needed files. Add all generated files except the .gradle/ folder to your repository (depending on your gitignore, it might ignore the .jar file. Make sure it is included though).
Now, merge these changes into all of your branches! This should only add the TravisCI changes (thus does not mess up grading for assignment3). Make sure it does and that you do not overwrite something. You can consider doing this as PullRequest on GitHub to have an easy overview of what will be changed.
Got to `https://travis-ci.com` and login with your GitHub username and password. You should be able to see your assignment project here. If not, check on the Travis CI help on what to do.
TravisCI should now build your project whenever you commit something to GitHub; it will tell you automatically if your build is successful or not. TravisCI will also run your Unit Tests if they are included in Gradle (as they are for your project). This way, you will see if your tests pass. Now you have the means to see if your builds are working upon each commit.
Answer in your document: Which branches fail on TravisCI? Explain why! Include a screenshot in your submission document. You can initiate a build on TravisCI directly in case it did not build that branch yet.
Now you know: how to recover a deleted file, how to make Gradle exclude a test file, and how to set up Continuous Integration.

Branch (2 points)

So looking at our branches we have master -> Blackbox -> Review, Blackbox -> Whitebox and now Whitebox -> StaticAnalysis. Review and Whitebox/StaticAnalysis are parallel workflows but descendents from BlackBox.
Commit frequently. Commit throughout working on different tasks whenever you think it makes sense. Work on your **StaticAnalysis** branch for the following tasks (none of the other branches should change for now).
On Canvas you find a file checkstyle.xml, add this file to your repository in a config/checkstyle directory. This file represents rules for Checkstyle which we will use next.

Task 1: Style checking using Checkstyle (13 points)

1. Open the build.gradle file from the project and uncomment the things related to Checkstyle (not Spotbugs yet).
2. Commit and push this change (It should also trigger a build on TravisCI).
3. You can use Eclipse or Gradle for working with Checkstyle. I am not giving instructions for Gradle, not Eclipse. If you want to use Eclipse, please research how it can be done.
4. Checkstyle is already included in the given Gradle file. You just need to uncomment that part. Checkstyle in Gradle uses a config file which is included in your repository in the */config* directory. Make sure you have this directory, and that it is version controlled.
   a) Depending on your Java and Gradle version, you might want to change the toolVersion from 8.8. This version is pretty old and basically leads to Gradle choosing the newest version, 8.41. I used the Gradle file as given with the JDK mentioned on Canvas and it all worked well.
   b) Look at the Gradle file, you should get a rough overview of what this Checkstyle code does
   c) Run *gradlebuild*
   d) Check out the Checkstyle report (reports folder)
5. Find the Checkstyle report after calling *gradlebuild* and take a capture of the report and include it in your document.
6. Answer in your document: How many violations did Checkstyle find in main? How many in test (easiest to see through Gradle reports)?
7. You will have a bunch of violations. Most of them will be about indentations and tabs. The Checkstyle style file I gave you assumes that there are no tabs and that the indentations are 4 spaces. You can change settings in your IDE to have only spaces instead of tabs. You should do so (Use Google if you do not know how). Then, go to each java file and let your IDE correct the indentations. This should greatly reduce the violations you get. How many violations do you have now in main, and how many in test?
8. Now, fix violations in the project (all the files in the main package) until you are below 25 violations in that package.
9. When you are done, take a screenshot of your report.
10. Take a screenshot of your command line window with *gradlebuild* and its output; please include the generated HTML file from main on the screenshot also (Remember, your asurite/name should still be somewhere on the screenshot).
11. Make sure that the code is still running correctly (as correctly as the first version).
12. Add your screenshots to your document.

Task 2: Static Analysis using Spotbugs (14 points)

Task 2.1: Gradle

- Add SpotBugs to the given Gradle file (just uncomment what is commented for it - there is something at the top and something at the bottom. You need both. Depending on your Gradle version, it might throw an issue. If needed, ask Google how to fix it.
- run *gradlebuild*
- Gradle should have created an HTML file for you with a couple of bugs (in reports folder).
- Take a screenshot of your report and command line window (and asurite) and put it in your document. We only care about the main.html.

Task 2.2: Fixing the bugs

1. Fix ALL errors reported by SpotBugs. Add the comment 'SER316 TASK 2 SPOTBUGS FIX' in all CAPS where you make code changes – or on the same line if you prefer. This is just so I can easily find your fixes.
    - Hint: Almost all of the bugs that are being reported by SpotBugs have a Stack Overflow post with the exact description shown in the bug explorer of SpotBugs. This will help you correct these errors.
    - Hint: If you have the HTML file, you can click on the bug and find out more information about it.
2. Take a screenshot of either your HTML file and command line, or of your XML and Eclipse environment showing the SpotBugs run. Add it to your document (As always, show your asurite somewhere).

Task 3: Comparing (3 points)

Now we have done the Static Analysis after our Whitebox Testing, but not after our Code Reviews (but in parallel). So let's first check how you did without Static Analysis, and if your Code Review already improved the code.
Do the following for your Review and Blackbox branch:
Switch to your Review and Blackbox branch and uncomment all the Checkstyle and Spotbugs things in the gradle file (commit and push this change).
What do you get now for these branches? Write in your document how many Checkstyle and Spotbugs violations/bugs you got and compare to the **initial** report you got from your StaticAnalysis branch (in main and test for Checkstyle and Spotbugs). How does this compare to the violations on the StaticAnalysis branch after correcting the violations and bugs? Make a nice table in your document showing all this data for your different branches (Blackbox, Review, StaticAnalysis -before all your fixes and after). You do not have to change the code in the old branches (eg. do not correct the violations/bugs).
Answer: Did it get better or worse? Can you explain why?

Task 4: Putting it together (5 points)

Now let's put things together. This might get a little tricky since you worked on (almost) all the files in the "parallel" branches, StaticAnalysis and Review. So you will probably get a good amount of merge conflicts.
Start off with creating a "Dev" branch which is based off of Blackbox (so we are pretending to have a rather old version in it).
Now merge all of your changes together, so that your newest and best code is in Dev at the end. It is your task to figure out the best way to make this merge happen.
Make sure that: all your tests cases/classes are included (including GivenBlackbox.java), your Gradle file excludes your GivenBlackbox.java test and that Checkstyle and Spotbugs are activated.
Run *gradlebuild* on this Dev branch and check the number of violations. In your document, record how many violations were reported by Checkstyle and Spotbugs.
Answer: Did it get better or worse, and can you explain why?
Reduce the Spotbugs violations to 0 and your Checkstyle violations in the main package to below 25 again (if they are over this threshold).
Run all your Unit Tests (excluding GivenBlackbox, which should still not run through Gradle) to make sure things are still working as they should. If any tests fail now, correct your code (either the method since it broke, or the test if the test was wrong).
Your build of Dev should be successful at the end. This should show on TravisCI -take a screenshot and add it to your document!
Do you think your code got better? In what order would you use these quality practices in the future?

About the merge

Maybe you got a little annoyed with the merge at the end and started wondering why I did not just have you work on one branch, or have them all descend from each other. It would have been easier since you would have avoided all the conflicts.
But I assume in your project you will have already learned that conflicts are hard to avoid, and that sometimes, work happens on the same file. Every one of you should be able to resolve these conflicts.
It also demonstrated the importance of fixing Checkstyle and Spotbugs violations on a large scale in your project. To avoid further conflicts, it's best to make these corrections quickly before anyone else starts working on it

Final Code review (10 points)

Do a final code review on your Dev branch and check that your code conforms to good coding standards, e.g. variable naming, indentation, and everything as private as possible. Make appropriate changes if needed and give a brief overview of what needed to change to improve the code in your document.
Of course your tests should still pass and the thresholds on Checkstyle and Spotbugs should not be passed.

Submission

On Canvas submit

1. GitHub link to your private repo (yes again)
2. Your PDF document including
    - All answers and screen shots mentioned above (sorry I am not listing them here again).