Northeastern University
College of Computer and Information Science

BlackSheep: Plagiarism Detector

Report

Team # 104

Prashant Havangi

Manpreet Kaur

Peng Tong

Athul Muralidharan

Problem Overview

Plagiarism is the practice of taking someone else's work or ideas and passing them off as one's own. The goal of this project was to design, implement, test, and evaluate an application that would help instructors detect plagiarism.

Plagiarism situations could be where two or more students submit similar solutions to an assignment, in which one version derives from another version. The derivations could be through one or more behavior-preserving transformations. Examples of such transformations are moving functions or methods to another location in the same file, renaming variables, classes, and methods, extracting sequences of statements into methods, etc.

The goal of this project to develop a plagiarism detector that could identify these situations and flag them.

The plagiarism detector must work on programs written in Python.

The main job in the project was to design an algorithm, written in Java, for detecting similarities between two programs written in Python. The detector must go beyond a “textual diff,” and consider more sophisticated transformations such as:

- the renaming of variables
- extracting code into functions
- moving code around

The detector must be able to handle multi-file projects, where files with similar content have different names.

The project was to be implemented in 3 phases:

- In Phase A, a set of use cases was to be developed that informally described the behavior of the system from a user’s perspective and a mock-up of the systems user-interface.
- In Phase B, a set of UML Diagrams was to be developed that reflected the high-level architecture of your system and a set of Java interfaces reflecting this design.
- In Phase C, the system was to be implemented in Java. This phase was broken down in three sprints.

Result

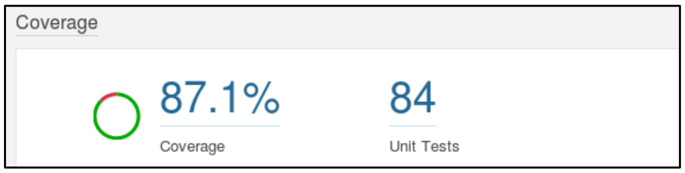
Completion claims

- We developed a web application to detect plagiarism in python files. The functionality expectations we have implemented includes: user login and register, employs three comparison strategy that delivers results against two projects (code may split among several files) and computes an overall score using a weighted polynomial function, performs the comparison against multiple submissions, provides usage statistics of the number of plagiarism detection cases run and system status, shows the result as both percentage and detailed code matches, UI is fully functional for both base and stretch expectation, keeps system logs activity, deploys system on AWS. The environment expectations we have implemented includes: uses smart commits in git, runs Jenkins and SonarQube on all pull-request (PR) submissions, GitHub informs the team of PRs via slack or email.

Quality claims

The code quality of our system is good. We have followed test-driven approach for the entire develop process. To achieve that, we wrote tests for each new class to ensure high test coverage. Jenkins was used to implement system integration while SonarQube was used to ensure the quality of the code. Besides, master has restricted access. Code merge was possible only via Pull Requests (PRs).

- **Test coverage**
  We used test-driven concept while developing the system, and SonarQube quality gate was applied to ensure that test coverage meet the expectation. Figure 1 shows test coverage of our system based on SonarQube:



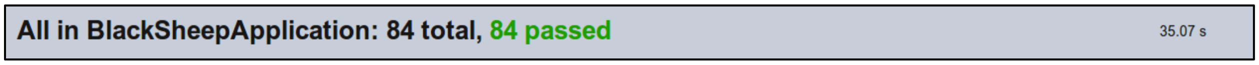
Coverage

87.1% Coverage

84 Unit Tests

Figure 1 Test coverage

- **Number of test cases**
  We have totally 84 tests for our system and all of them were passed, as is shown in Figure 2



All in BlackSheepApplication: 84 total, 84 passed 35.07 s

Figure 2 Number of test cases

- **Number of bugs:**

When our team wrote the Java code, we strictly follow the Java Code Conventions with detailed Javadoc included. Besides, our system has no bugs on code level according to the result on SonarQube (see Figure 3).

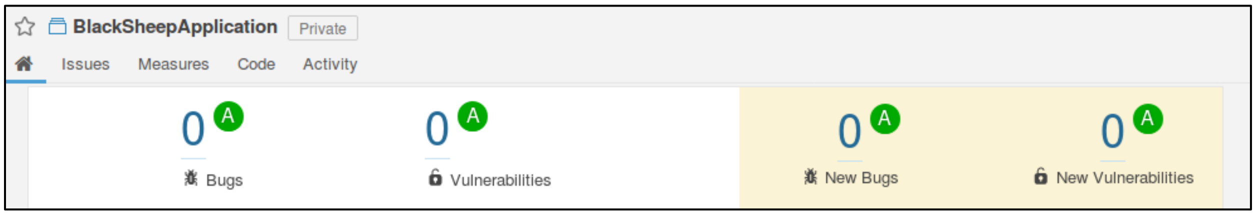


Figure 3 SonarQube result

- **Number of pull requests**
  To ensure the quality of code in master branch. All branches could only be merged to master through Pull Request and must pass the quality tests on SonarQube. For the whole development process, we have created totally 104 Pull Request (shown in Figure 4).

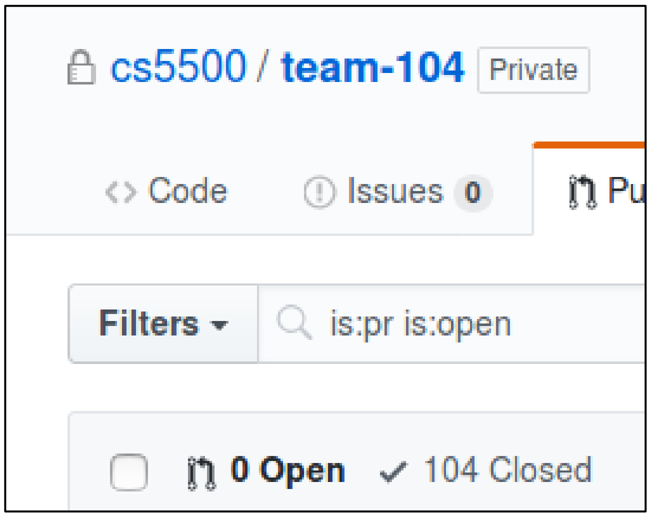


Figure 4 Number of pull requests

Development process

We first met after the team were formed and we had a quick introduction of each other. The main objective of the introduction session was to know the skill sets each one of us had. We shared our background, interests and experience. We discussed the project's objectives/goals and started to think about what role each will play in the team.

Phases A and B, required us to sit together and come up with an architecture for the application. We met the professor (who was our client) a couple of times to understand the client requirements.

We created the use cases and the UML designed based on our understanding of the client requirements.

For phase C, we decided to divide the team as follows:

- Dev Ops: 1 person. He handled the Jenkins and SonarQube installation and setup. He was our main point of contact for any problems related to these
- Backend: 2 people. They worked on java APIs to receive data from front-end and return results to UI, integrate all the algorithms and implement design patterns.
- Frontend: 1 person. He worked on front-end of the application

Efficient process management

We integrated slack with GitHub to receive notifications for all the commits and merges to the repository.

GitHub was integrated with Jenkins and SonarQube is for Continuous Integration and Quality control.

SonarQube minimum test coverage was set to 85%.

Code commit process

The master was set to protected. This prevented direct commits to it.

Hence, for every merge to the master, a pull request had to be created which in turn triggered the quality check in SonarQube.

A merge could not be completed without approval of at least one reviewer and all checks passed from SonarQube.

SDLC

We followed Agile methodology for the project.

Throughout the project, we had daily sync up meetings to ensure that we are on the right track and that no one is blocked.

During Phases A, B and Sprint 1, we had daily slack meetings for 15 mins where each team member addressed each of the following questions:

1. What was done the previous day
2. Is there any blocker

3. What is the plan for the next day

Sprint 2 onwards, we had daily stand up meetings in person but followed the same approach as slack up meetings.

We followed test-driven approach for the implementation.

Coding best practices followed

We used SonarLint to ensure that best practices are followed.

We strictly adhered to the best practices listed below:

1. Generating secure password hash
2. Use logging extensively
3. CamelCase Naming Conventions
4. Class Members should be private
5. Avoid Empty Catch Blocks
6. Use StringBuilder or StringBuffer instead of String Concatenation
7. Use Interface References to Collections
8. Return Empty Collections instead of Null
9. Avoid unnecessary Objects
10. Release database connections when querying is complete.
11. Use Finally block
12. Handle Exceptions

Some of the things that did not work at the beginning, but we did address these issues as we went ahead were:

- Lack of communication among the team (During Phase A).
- No clear roles/responsibility (Start of sprint 1).
- We worked alone, taking up tasks on JIRA during initial Phase of sprint 1, later we started to look after each other and aided each other, if required.
- We used to not discuss the problems each one is facing, either while completing the tasks or the timeline/schedule issues, hence we would create a hole when someone was busy. We became more open as the project went by.

Retrospective

Things the team liked:

- Developing a system with Agile process helped us to get prepare for real work.
- Usage of Jenkins and SonarQube let us understand the important of code quality and continuous integration.
- Applying concepts learned in class (like design pattern, test strategy) helped us be more familiar with the software development process.
- The process in development of the project, we liked the JIRA and it helped us keep track of the progress we are doing.
- Having the slack ups were useful and prevented us from meeting in person.
- Smart commits were quite helpful and efficient.

Things the team did not like:

- The pace of project was kind of fast, sometimes we did not have enough time to implement all the functionalities.
- The different TAs had very different judging and scoring criteria.
- Teammates had different classes and schedule, thus, there was limited time for us to work together.

Changes in the course required to support great experience:

- The description of assignments should be clearer so that we can better understand the expectation of the work.
- Should make a standard criterion for all TAs when they were grading the assignments and projects.