Plagiarism Detector

CS 5500: Foundations of Software Engineering - Team 3

George Dunnery

Yujia Liu

Jacob Piersall

Akashbir Singh

Northeastern University

Functionality & Design

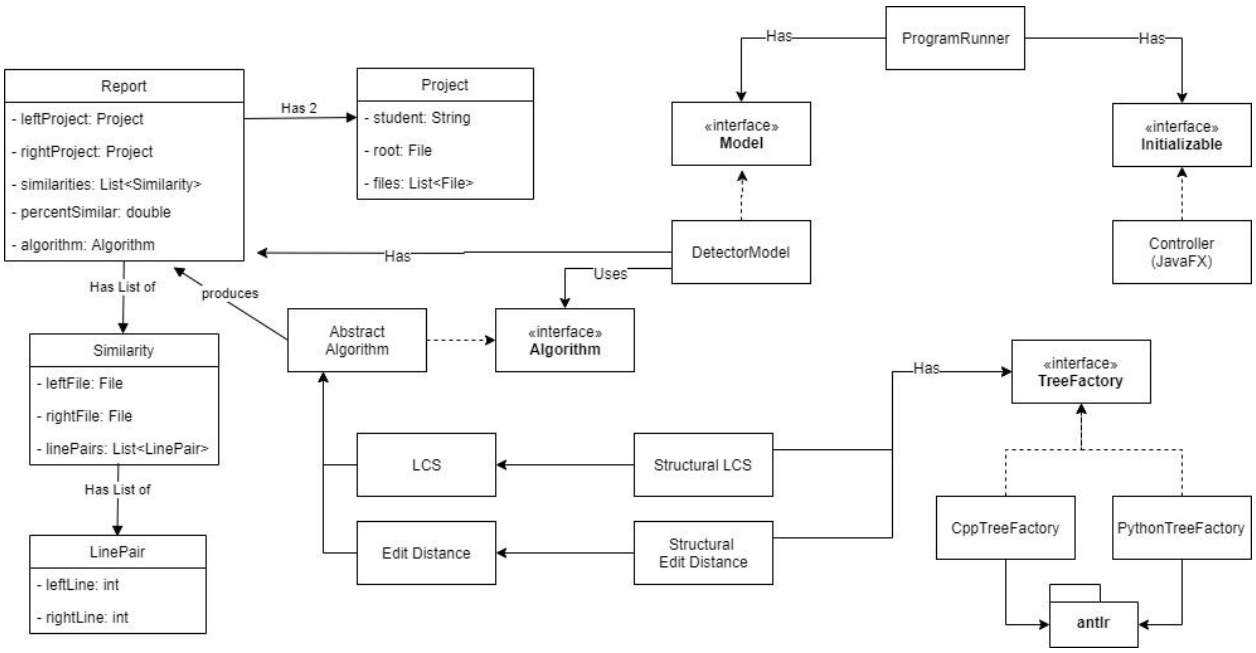
The Plagiarism Detector is an application intended to programmatically detect instances of possible plagiarism in code-based projects. The system is designed around the core concept of comparing two student projects in a side-by-side manner. Throughout this report and the implemented code, this comparison is reflected by a naming scheme which prefixes items like projects, files, and lines with the words “left” and “right” to help conceptualize the program’s logic. The application supports the selection of two projects consisting of an arbitrary number of files which can be selected at the directory level. The files on the left and the right can be viewed as plain text in the central white boxes prior running the detection algorithm. The user can then choose from four algorithms to use for plagiarism detection and press the detect button to run the selected algorithm. After a detection algorithm is finished, the similar lines of the currently selected files are automatically highlighted. The user can now select different pairs of files on the left and right to view the similarities between each pair of files. A status bar provides a visual representation of how similar the two projects are overall.

Algorithms

The Plagiarism Detector uses two dynamic programming algorithms and includes a variant of each that processes the string version of an abstract syntax tree for a total of four algorithms. The first is the Longest Common Subsequence (Cormen et al., 2009) algorithm which finds the longest sequence of matching characters between two strings, including non-contiguous subsequences. The second is the Edit Distance (Mishra, 2019) algorithm which calculates the minimum number of operations required to convert one string into the other. Both of these algorithms follow the same meta-procedure: two projects are put in, every file in the *left*

project is compared with every file in the *right* project, and every string in each *left* file is compared with every string in the *right* file. An intuitive result of this process is that the Plagiarism Detector excels at detecting relocated code at the expense of efficiency. Each of these algorithms was then extended as the "structural" version, which uses the string representation of an abstract syntax tree generated by the ANTLR package from each file. The structural versions are more performant because they compare file to file rather than the minutiae of the line to line comparisons. An intuitive result of these structural algorithms is that it can detect renamed variables with better accuracy than the line by line comparison because the node types match.

UML Diagram



Support For Multiple Programming Languages

The system was designed with multi-language support in mind from the beginning. This requirement is part of what inspired the use of the LCS and Edit Distance algorithms - since each algorithm orchestrates string comparisons, they both will technically work with any language. The structural versions of each algorithm are explicitly tied to the available languages in the ASTFactory classes. Currently, the system supports C++ and Python 3.x as a result of the incorporation of the ANTLR package for generating abstract syntax trees. Structural algorithm support for any other language can be added by including the corresponding grammar file (.g4), running ANTLR to generate the other files, and adding an ASTFactory class implementation for the new language. This design allows for simple and fast expansion to additional languages at any point in the future.

Architecture & Design Patterns

Several design patterns were used throughout the development of this application. The most visible design pattern, which is responsible for the overall architecture of the Plagiarism Detector, is the Model-View-Controller pattern. The model handles all the application data, the view displays information and offers an interface for the user to interact with, and the controller facilitates the exchange of information and commands between the model and the view. The core concept behind this design pattern is to reduce coupling by creating interchangeable components which can be swapped out for alternatives without compromising the functionality of the other components. The factory pattern was used to simplify the process of generating an abstract syntax tree (see interface *TreeFactory*). This pattern was particularly useful because it allowed the encapsulation of the variations between the tree generation process for C++ and Python 3

using ANTLR, and furthermore provides a perfect entry point for adding support for additional languages later. The strategy pattern was used to enforce uniformity in the input and output of the algorithm used for plagiarism detection (see interface *Algorithm*). This pattern was also very useful because it provides an easy way to change the algorithm used without affecting the surrounding code, and additionally provides an entry point for adding more algorithms in the future. The singleton pattern was also used for the algorithm classes to simplify the process of creating algorithm objects and to avoid confusion between multiple instances of the same algorithm. Lastly, composition was used liberally among the data structures that track the similarities between student projects. For example, a pair of similar lines are stored in a LinePair, which is stored in a Similarity as a List<LinePair> for two particular files, which is then stored in a Report as a List<Similarity> for the two projects. This pattern provided a neat and intuitive way to organize the data that was acquired during plagiarism detection and helped encapsulate the inner workings of how similarities were stored.

Evolution From Phase B to Phase C

The design of the system experienced many changes as it evolved from the initial plan in Phase B to its final implementation in Phase C. The strategy pattern using the Algorithm interface and the use of the MVC pattern for the overall architecture remained the same throughout the process, but many other ideas were modified or dropped entirely. The ASTFactory became the TreeFactory, but the pre-processing concept was eliminated due to concern over contaminating the acts of plagiarism with additional modifications. One major

change was the decision to use the ANTLR package instead of creating the AST's from scratch. Initially, the plan was to use a single SimilarityEvent object to keep track of instances of possible plagiarism. This evolved into the hierarchical set of data structures consisting of the Report, Similarity and LinePair that are present in the final implementation to prevent unnecessarily storing duplicate data (like file names) and provide an internal layer of encapsulation. The use of JavaFX was added during Phase C at the recommendation of the course professor. Overall, the majority of the method signatures that were originally planned were changed, broken down into simpler methods, or removed entirely as the process of system construction demanded alterations. Lastly, many packages were created to keep all of the project files organized which were not part of Phase B.

Strengths and Weaknesses of the Plagiarism Detector

The following is a discussion of the reliability of the Plagiarism Detector in the context of several common strategies for concealing plagiarism. The detection of code which has been reordered or moved to a different part of the file works exceptionally well when using LCS and Edit Distance as a result of the line by line comparison. Renamed variables can be detected by using the structural version of either algorithm, or by lowering the similarity threshold in either of the string comparison algorithms (although the latter performs which lower reliability). The creation of sub-functions, or coalescence into one larger function is detected moderately well, with the caveat that the new method signatures and return statements of creating sub-functions (or lack thereof when condensing functions) can make the copied code appear less plagiarised than it actually is. One area that was not addressed at the time of the final implementation was

the conversion of for loops to while loops and vice versa. Similar content within the loops will be detected, but the loop statement itself will remain concealed if this conversion is performed by a student. Lastly, an interesting “bug” that was encountered occurs during string comparison when the longer file is in the outer loop of the line to line comparison. The symptom of the bug is that there appears to be an extra similarity between the same two files when the longer file is selected on the left as compared to when the longer file is selected on the right (the two projects should each contain a pair of copied files to replicate this scenario).

Reflections & Lessons Learned

The software engineering process was a learning experience for all members of the team. Following the agile process proved to be more difficult than anticipated. The team met in person at least once per week and held scrum meetings, but progress was slow and the team suffered from a lack of structure while working independently. Pair programming was used frequently during the meetings and was found to increase both the quality and quantity of the code pushed to production because the partners would catch each other’s mistakes and were readily able to help fill in the gaps when one person was unsure of how to proceed. The testing process was difficult due to the changes that were made to method signatures and class hierarchies; old tests were sometimes unavoidably invalidated due to a necessary change. Some test signatures were created before the actual test code was written as a placeholder for some aspect of the system that was pending testing or as yet available for testing. To ensure that the tests were eventually written, they simply contained a fail() command with a message that the test still needed to be written. An implication of this practice was that continuous integration became unwieldy

because the team could not tell if any tests were broken by a new update. A better system of planning out tests would be used in the future. The decision to learn and incorporate new packages as critical components in the application despite the fact that no team member had any experience with them was a risky but worthwhile choice. The team was ultimately able to learn and apply the ANTLR and JavaFX packages to create a solid application, but the risk of losing significant amounts of time and energy on experimenting with these packages was recognized early on.

References

1. Tip, F. (2019). *CS 5500: Foundations of Software Engineering*. Boston, MA: Northeastern University
2. Pressman, R. S., & Maxim, B. R. (2015). *Software Engineering: A Practitioner's Approach* (8th ed.). New York, NY: McGraw-Hill Education.
3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). Cambridge, MA: The MIT Press.
4. Mishra, R. (2019). *Edit Distance | DP-5.* GeeksforGeeks. Retrieved from: https://www.geeksforgeeks.org/edit-distance-dp-5/