DasYarn Final Write-Up

Team Members : Matt Gurman, Amanda Dupell, Brian Ward

1. Installation instructions and userguide, documenting the functionality of your system (10 points)

Instructions to make an Installation:

After unzipping our project release, you will see that our project consists of two typescript projects.

1: **das-yarn** : our React front end.

2: das-yarn-express-ts : our express backend API server.

It is important to start the backend server first as it needs to be running on port 3000. To do this please navigate to the root directory of **das-yarn-express-ts** (make sure you have yarn installed) and enter the command **'yarn install'**. This should install all the dependencies necessary for the project. You may then enter the command '**yarn start'** this should fire up the server, again it is important that this server is running on port 3000. If the server is running correctly you should see:

> *'Express server has started on port 3000. Open http://localhost:3000/ to see results Listening...'*

You can now open a new terminal window and navigate to the das-yarn root directory. You should now follow the same steps as above:

1. 'yarn install'
2. 'yarn start'

Here you should see be prompted to ask if its okay to start the server on port 3001 as port 3000 is already taken. You can type 'Y' to give the okay and the front-end will build and should automatically open your browser to the project.

The home page will be the point of submission. please hover over any of the inputs that might not be clear for further explination.

Troubleshooting: if for some reason you cannot get the server to run on port 3000 but you can get it to run on another port you can edit the url for api endpoint in the front end. to do this navigate to '**das-yarn/src/services/SubmissionService.ts**' you can then edit the parameter '**url**' to match the port you are able to run the server on.

Instructions to make a Submission

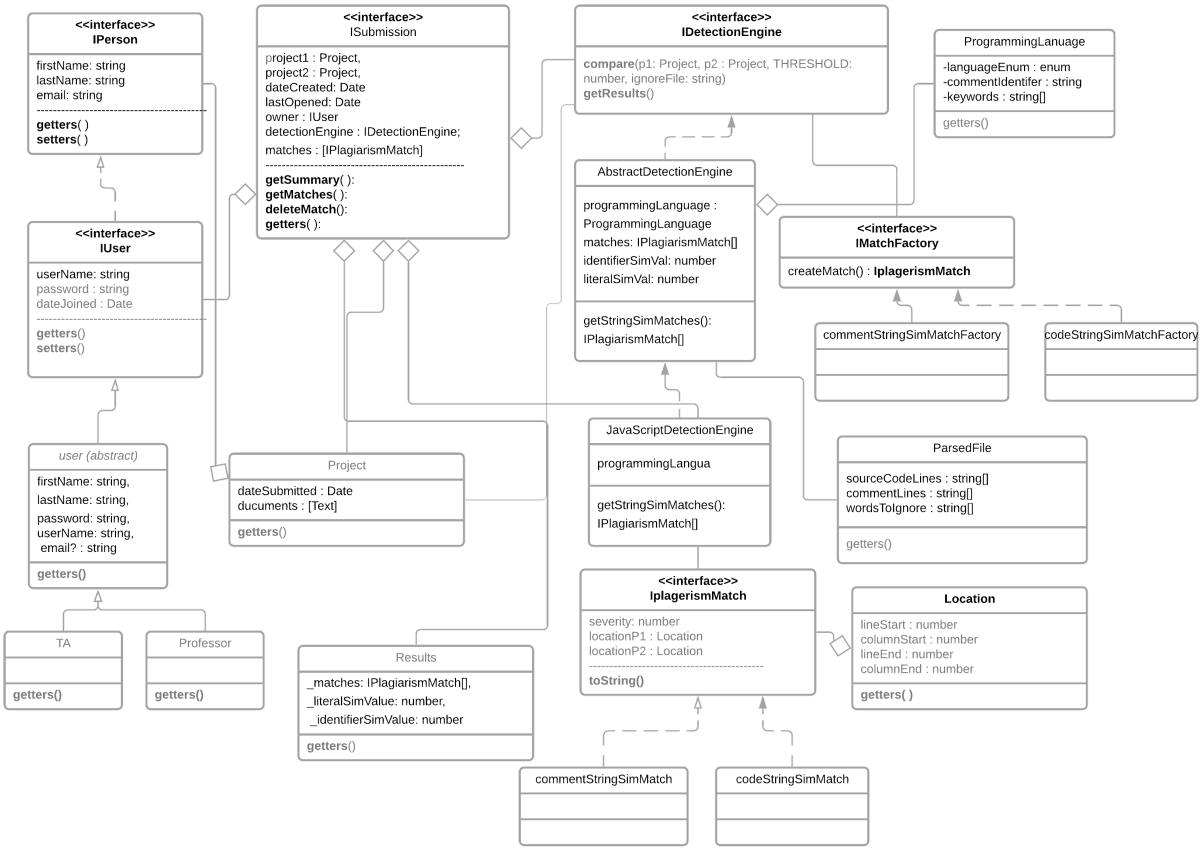
To upload a submission, first fill out the form by giving a submission name, the programming language that the submission will be in, a Sorensen-Dice Similarity threshold for what constitutes a plagiarism instance (see section 3 for a more detailed explanation for what this is), and a file upload that contains any code you want to be ignored in the analysis. Then, upload two projects in each file upload container. Once you have files uploaded, click run analysis. This will bring you to the results page, where you will see the results of the plagiarism analysis.

On the top of the results page you will see percent values for the identifier and literal similarity values (again, see section 3 for a more detailed description). To the right of that, you will see the total number of comment and code matches.

Under that, there are two windows, one fort each project. You can scroll through each and navigate between files within the projects. Code that is highlighted red represents code that was flagged for potential plagiarism.

At the bottom of the page is our Match view, where the user can toggle between matches of code that our plagiarism algorithm. For each match, the code from each project is displayed and the user can decide to discard it or keep it. Both of those buttons will toggle to the next match.

2. High level architecture diagrams and discussion of the overall infrastructure that you designed to implement the project (10 points)



Updated Project UML

The entry point to our design is a submission interface that holds the data relevant to a submission for plagiarism review. This data is gathered in the front end, packaged up and sent to the backend server: a submission name, two separate lists of files to be compared, the language that the programs are in, and an optional file that contains code to be ignored by our algorithm. Due to our decision not to use a database, we realized that creating an implementing submission class was unnecessary; however, we kept the interface in our model because implementing a backend is something we hope to do in the future, and having an interface for submissions allows us to incorporate new kinds of submissions if need be.

The submission interface relates to another interface called IDetectionEngine, which contains the core algorithmic logic to detect plagiarism. We felt it was a good design decision to separate the data for a submission and the logic used to

process that data and give results for our analysis. Again, using an interface for the detection engine allows us to implement detection engines for multiple programming languages, although with the time given we were only able to implement one for JavaScript.

For representing instances of plagiarism, we have a plagiarism match interface that represents different kinds of matches that we detect in our detection engine. This allowed for multiple kinds of matches. A match relates to a Location class that contains match data: which file and where inside of that file the match is. To create matches, we implemented a match factory interface that was also extensible to create different kinds of matches. The matches are the core of our plagiarism analysis.

Finally, we have a class hierarchy that represents Users of our system. Due to time constraints, we were unable to implement this in our frontend. At the top was a Person interface that could represent a User of our system or an author of a submission. Implementing the Person interface was the User interface, which was either a Professor or a TA. From there, we implemented the User interface with the Professor and TA classes. These would be useful if we extended our program to include login capabilities and persisting data between user sessions in a backend.

3. Describe the algorithm that you implemented for detecting plagiarism (10 points)

For our plagiarism detection engine we have three different metrics to compare projects with the extensibility to add more as the tool development moves forward.

1 & 2. Identifier Values Similarity, Literal Values Similarity

The first two values are created from parsing the files into AST's. We chose to use the AST's to collect all the Identifiers and Literals in each project and compare them. Although these values alone do not perfectly correlate with

plagiarism, we believe that they do provide insight into the overall similarity of the projects. Identifiers can represent a lot of information in a project such as methods used, functions called, etc. A similarity of literals provides even more insight on the similarity of the projects as literals are much more unique. To calculate these similarity values we decided to use the Jaccard Index: a token based similarity.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

Once each file has been parsed into its respective AST we can travel down the AST adding each Identifier to its project set. Once each set of identifiers has been created we can calculate the sim value using the formula above. The Jaccard-Index will give a value of 0.0- 1.0 which we converted to a percentage to display to the end user. The Literal Value is calculated in the same manner.

3. **Similarity Matches**

The first and most descriptive form of comparison is done by a line x line comparison via a token-based string comparison. The string comparison that we have implemented right now is the Sorensen-Dice index.

$$\text{similarity} = \frac{2 * |X \cap Y|}{|X| + |Y|}$$

Two strings are read into the comparator, they are then split on empty space, cleaned (i.e trimmed and translated to lowercase), and are then separated into their respective sets. The Formula above is then calculated on the two sets to give a 0.0 - 1.0 similarity value.

The Sorensen-Dice Similarity class implements an IStringSimilarity Interface which will allow for easy interchangeability of different similarity calculators. Other

similarity calculators we played with were Longest Common Substring, and Jaccard Index, but we ultimately found that the Sorensen-Dice Similarity calculator gave the most relevant value.

In future iterations it might be beneficial to use multiple string similarity calculators and average the values, or perhaps give the user control over which calculator(s) they might want to use. As previously mentioned, the string sim calculator will be used to compare each line of project A to each line of project B. There is however some pre-processing to fine-tune these results.

Pre-Processing

Influencing our pre-processing was a CodeMatch research paper that used this same method for pre-processing, which involved splitting any given file into an array of code and an array of comments, so these could be analyzed separately.

As a file is parsed we first determine whether or not a line is a line of code or a line of comment (including block comments). This way can only compare comment lines to comment lines and code lines to code lines. There are also lines which we will want to ignore, i.e lines that consist of only a closing bracket. To do this we are going to check that line contains an alpha character, if not : ignore it.

Removal of Keywords and the Given Ignore File.

We might also want to ignore language keywords, as they will certainly increase the similarity value. To do this we simply take in a file which holds the keywords for the specified language, and remove any of these words from the lines before the comparison is executed.

Using this same functionality we are able to offer a parameter for the user to upload a file of words or lines of code that will also be ignored from the matches. This is a particularly nice feature for teachers, as many assignments often come with starter code. In order to get the plagiarism tool to ignore the starter code all the user needs to do is upload a file containing that code.

Creating Matches

Now that our lines have been pre-processed, we can continue to compare each line of code and comments respectively. We chose to have our results class hold an array of IMatches, an interface that represents a certain type of match between the two projects. We also implemented an IMatchFactory so that we may easily extend the detectionEngine to create different types of matches. A

commentMatch or codeMatch are created if the two compared lines have a simValue greater than the user-given threshold. This gives the user the ability to adjust the sensitivity of the similarity matches to best fit their expectations.

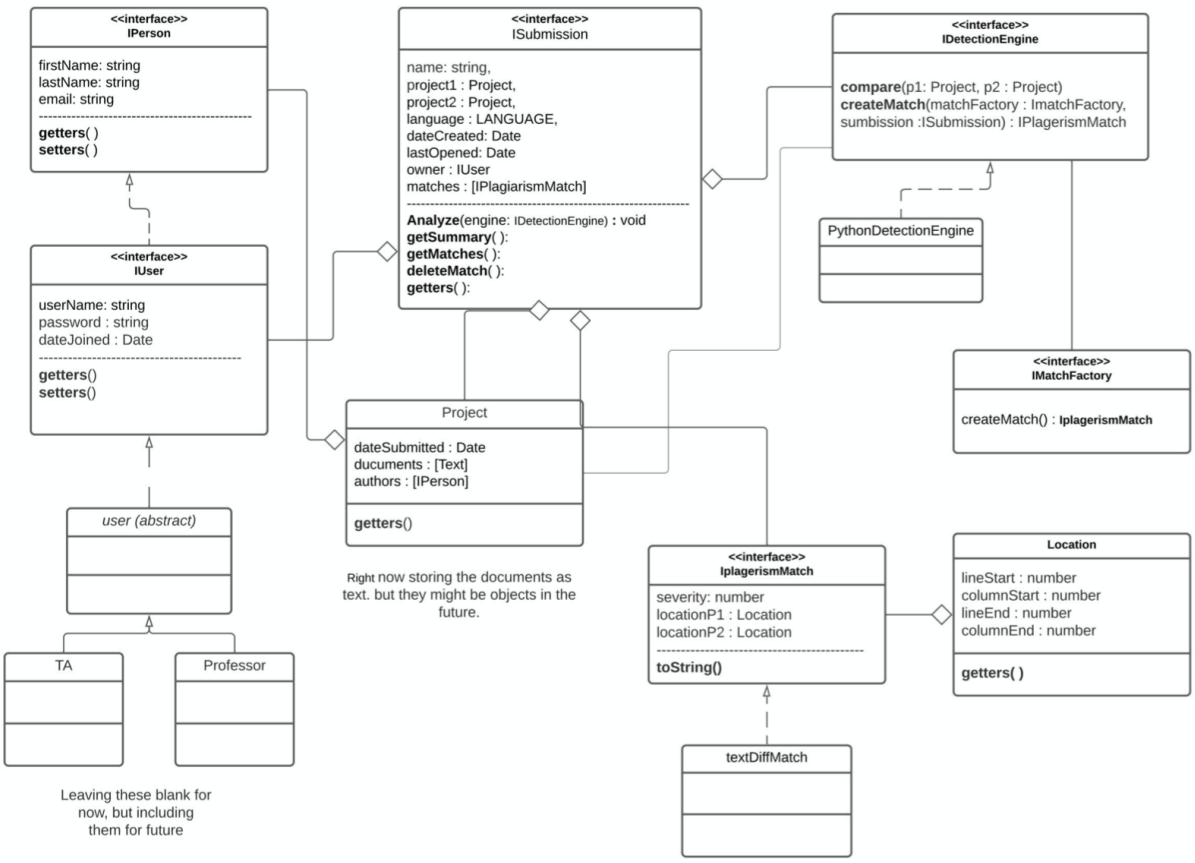
We also included the functionality to combine matches if there were successive matches in a row in both projects. This way the number of matches is decreased and matching snippets of code or comments will all be included as its own match.

Removal of Matches

Once these matches are passed to the front-end the the user is given more control in the ability to keep or discard any of these matches. keeping a match will of course keep the match and discarding the match will be accepting the similarity of the two lines and will allow the presented values to update automatically with that match removed. Our goal in this project was to capture as much information about the similarities of the project and simply display the information to the user to make the final decision on whether or not plagiarism might have taken place. That is why we focused on keeping as much control and customization in the hands of the user.

4. Discussion of how the the design and functionality of your system evolved since the phase B plan. Include a discussion of how you came to your final implementation of the plagiarism detection algorithm. (10 points)

OLD UML FROM PHASE B:



As far as our user interface goes, we made small changes to the original design to accommodate the changes to our implementation plan. We got rid of the persisting header and sign up and log in flow, as we did not have time to implement that piece of our backend service. This also meant that we did not have time to complete the submission history page for a user that was part of our original mock up. Instead of having a modal pop-up to display instructions on the proper way to submit, we chose to include that information as a part of the submission component that the user interacts with to submit a project.

The final results page has a number of changes from our original wireframe that stem from functionality changes. First, the user is presented with a broken down summary of their analysis. This summary includes a literal similarity value, an identifier similarity value, the total number of code matches and the total number of comment matches. This differs slightly from our original plan because when

first designing the visual interface for our program, we weren't fully sure of what our final match object was going to look like and what types of matching we would support. Below the broken down section is the program view for the files. This view is for the user to display matches in files that they submitted for analysis. The only difference in this section from our original design was the color of the highlight. In our plan, we had said that the color of the match would coordinate to the severity of that instance of plagiarism. However, after implementing the algorithm, we found that the severity of a match doesn't give much more meaning than by just highlighting. Therefore, we chose to highlight all matches with the same red indicator color. The last section of the results page is our match view which goes into a narrow view of a match to allow a user to keep or discard it from the final results. This section of the page did not change much from our original plan.

As far as implementation, we chose to make it as extensible as possible in order to use different similarity calculators.

5. Reflection on software engineering processes that your team applied in the project, including design patterns, development process, version control, code review, and testing (10 points)

To manage and track changes to our code, we used GitHub pull requests and Slack as our main form of communication. GitHub allowed us to easily branch off of master and create our own branches where we made our changes. Code review was done by teammates, where one or both of a member's teammates were required to approve code changes before merging them to master.

Additionally, we defined a style guide to govern coding practices, such as using camelCase for variable and function names, ProperCase for class names, and including line breaks after opening brackets and before closing brackets, etc.. These were mostly taken from Google's style guide for TypeScript.

We utilized all four software design principles discussed in class:

1. Abstraction:

   We abstracted common fields, methods and functionality within class hierarchies, as demonstrated with our detection engine — an interface defined the common fields and methods in the hierarchy, the abstract class defined shared functionality and the implementing class took care of class specific code/functionality

2. Encapsulation:

   Every class we wrote contained its own data and functionality and didn't rely on the functionality.

3. Modularity:

   The parts of our software program are modular: they can be used elsewhere in the program — or in other software programs — easily and without modification

4. Hierarchy:

   We have a few hierarchies in our program, and one we will discuss here is the IPlagiarismMatch. The key realization here is that we have a hierarchy of different kinds of matches, which allows us to substitute different kinds of matches into our code without problem. This is unlocked by the IPlagiarismMatch interface, of which all our implementing match classes implement.

A few design patterns that we implemented in our system:

1. Singleton

   We used a singleton pattern in a few different places:

   1. The instance of JavaScriptDetectionEngine in index.ts
   2. The submission service used in the frontend to make requests to the backend

2. Factory

We used the factory pattern to create matches, with an interface IMatchFactory defining the behavior for creating matches. In hindsight, we see now that an Abstract Factory would have been more applicable — a learning experience!