Brandon Pfoff, Cameron Graff, Clare Buntrock,
Sydney Lynch, Graham Kitchenka

Autograder 4.0: Blue Jay

COLORADOSCHOOLOFMINES
EARTH • ENERGY • ENVIRONMENT

Instructor-Facing Update

Table of Contents

Introduction and Requirements

Since 2014, professors at Colorado School of Mines have been saving time and energy by using Autograder, an efficient web interface that allows them to create and assign courses, assignments, and problems in their coding classes. Once these are made, students can log in and use C++ or Python to complete these problems and get feedback and grades almost instantly. The system is the result of multiple years of work and two other field sessions (Summer 2016 and 2017). It is used for homework and exams in multiple classes every semester and is a valuable tool teachers have come to rely on.

Before the summer of 2019, Autograder already worked. To make it work, however, instructors had to use some convoluted methods and many home-made Python scripts to get Autograder to perform as expected. Making new problems and grading were time consuming processes, and Autograder lacked the ability to test students on file I/O and prevent hardcoding. Additionally, the majority of the features relied on default selections and assumptions made by the primary programming language, Ruby, which made some administrative processes within the web application difficult and not user friendly for the instructors. Our client was Pr. Painter-Wakefield, a CSCI261 and 262 instructor that has used Autograder since its inception. Frustrated with the current system, Pr. Painter-Wakefield assembled our team to crack open Autograder and add six new features to make life easier for Mines instructors and make Autograder a good candidate to go open-source. Someday, Autograder could be used by instructors all over the world to help their students learn to code.

The tasks of updating Autograder were split between two teams: Autograder 1 was tasked with the student facing updates such as adding auto save features and improving the student side GUI. Autograder 2, our team, was tasked with the instructor facing updates to fix the problems listed above. We had to work closely with the other team to make Autograder the best it could be.

Functional Requirements:

1. A way to create and edit problems in the web application without having to upload extra files to the server; this feature needed to include support for header files and photos.
    a. Originally, test cases and starting code for a problem were stored in a local filesystem and pushed to the repository to make them available on the server. This made problems hard and time-consuming to create and impossible to share.
    b. The initial method of creating a problem involved doing some of the work on the web application and the rest of the work on a local machine with a file system. The end product was supposed to combine everything into one simple interface in the web application.
2. A way to import and export entire problems with just one button; this feature needed to include support for every type of supporting file including starter code, configuration, images, and more.
    a. This feature would allow instructors to share problems with each other, an important feature if Autograder is to become open source and used by different universities. Additionally, we were asked to make it possible for instructors to

filter problems by user, assignment, and term, and then export and import multiple problems at once.

3. A way to make some tests' input and expected output hidden from students but still contribute to the grade.
    a. When a student submits code to Autograder it is tested using many test cases chosen by the instructor. The results of these test cases are shown to the user in the testing bar - each test is represented as a red or green box on the bar depending on weather or not it passed. In previous versions of Autograder all tests were visible which means that if a student hovered over a test box the web app would show the test's expected input and output and the actual output. This created issues when students used it to hard code results.
    b. Instructors needed to be able to choose which tests are visible and which tests are hidden, if any. Hidden tests would be run and displayed like visible tests except when a user hovers over them they do not display expected input and output. Users should be able to see that hidden tests have been run and hidden tests should still contribute to the grade like a visible test would.
4. The ability for instructors to use text files in their problems for the students. This includes problems where the student can read from a file to answer a problem, or write to a file. It should be supported as both a whole program and as a standalone function.
    a. In its earlier iterations, Autograder could only support primitive types as inputs and outputs. File input and output also needs to be testable like every other Autograder problem, and the interface for making a new file I/O problem needs to be easy to understand and use.
5. A way to administer grading policies to how problems are graded, including late policies, extra credit, and different point values, as well as make different policies for different instructors and be able to select and change which policy applies to each assignment.
    a. In the past, instructors had to modify each student's score manually to incorporate grading policies, and every problem was only graded out of 10 points, regardless of whether it was required or extra credit. Additionally, the interface for making a new policy needed to be understandable and include every policy element a teacher could want, and there needed to be a way to apply different policies to different assignments.
6. A way to, with the minimum amount of buttons possible, import a CSV spreadsheet from Canvas that has an empty column for an assignment, find that column and populate it with all the students' grades for that assignment, and export the completed spreadsheet again (compatible with the grading policy feature).
    a. To get grades from the original Autograder, instructors used the default Rails export function, which generated a CSV file with every grade from every problem that every student had ever submitted. Instructors then had to use home-made Python scripts to clean this up and find the grades they wanted and then manually enter each into Canvas.
7. (Stretch) Port Autograder to the latest versions of Ruby, g++, and Rails.
    a. This would improve the support and security for Autograder.
8. (Stretch) Improve python support.

a. Students can do tests in Python, but the scope of what teachers can ask and assign is limited compared to Autograder's C++ support.

Non-Functional Requirements:

- Constant communication and merges with the other Autograder team
- Constant communication with our client and frequent merges onto the school server once new, functional features are added
- Making sure our team does not introduce security risks into Autograder or violate FERPA - the federal laws that protect students' privacy for grades and education records
- Maintain existing functionality of the application

System Architecture

For our project, we did not start building a system from scratch. When we began work it was on an already massive system with its own preexisting architecture. This architecture was too big to map without losing most of the important details and functionalities, but does center around some key, mappable, elements. The following are descriptions of models and database architecture, and after that is the architecture of every new feature our team implemented.

Models

Models are Ruby classes. Each model has tables in the database and its own helper functions, and each model has a corresponding controller and views that generate what the user sees in the web app. The models are like the puzzle pieces that make up the website. Autograder has the following models (Bold means new models we created and italicized means old models we modified):

- *Ability*
- Application_record
- *Assignment*
- Auto_save_state
- *Course*
- Current
- *Grade*
- **Policy**
- *Problem*
- Static
- *Submission*
- *User*

Database Architecture

Autograder is built on top of a large system of databases. Most models have a corresponding database, and some have extra cross reference tables for their complex relationships with each other. The following ERD shows all of the databases and how they relate to each other:

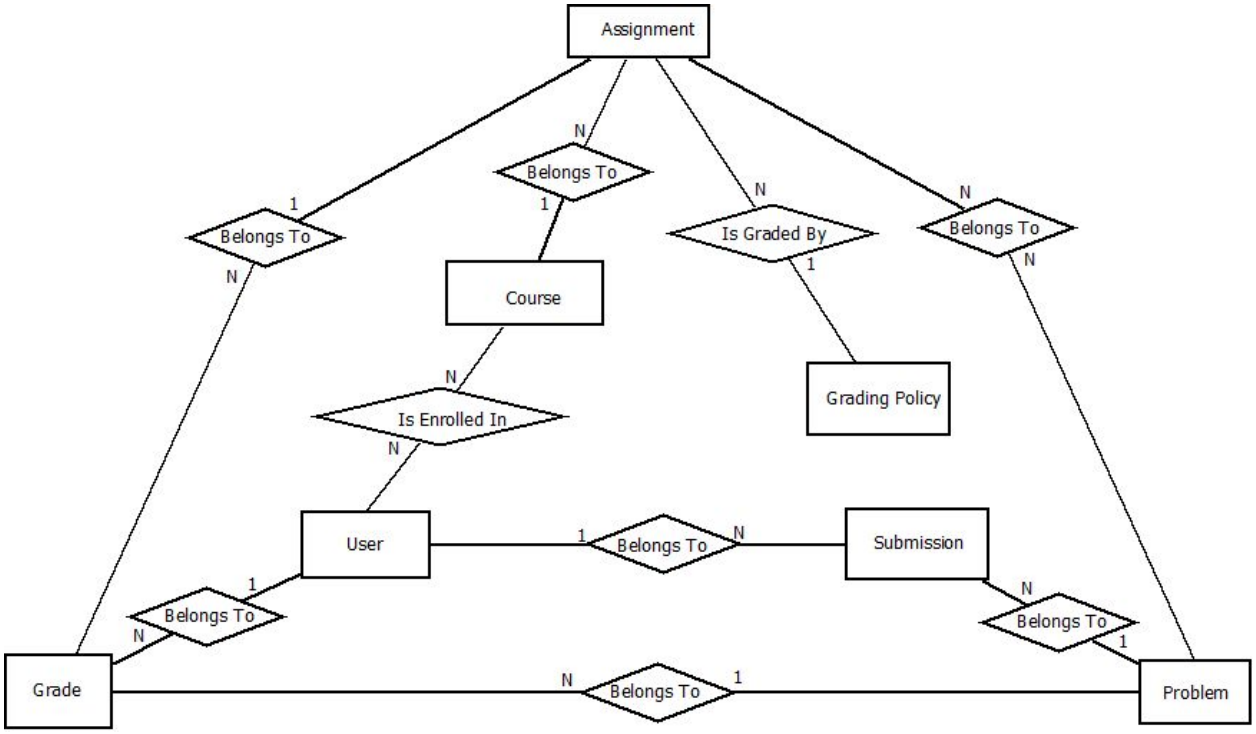

Figure 1: ERD of the Autograder Database system (simplified for clarity)

Problem Creator Architecture

Much of the work we did to add the feature where an instructor creates a new problem inside the web application was done in rails admin and the problem controller. Rails admin is an administrative interface with built in tools that can give the web app users an easy way to create, edit, import, export, and view new instances of each model. To add a problem editing interface we added new fields to the problem model and database with a migration, fields such as name, description, tests, and more. In rails admin we added these fields to the edit helper function so user input is automatically put into the appropriate fields. The ProblemGrader class then looks into these fields to display the problem, cutting out the need for additional files and finding file paths.

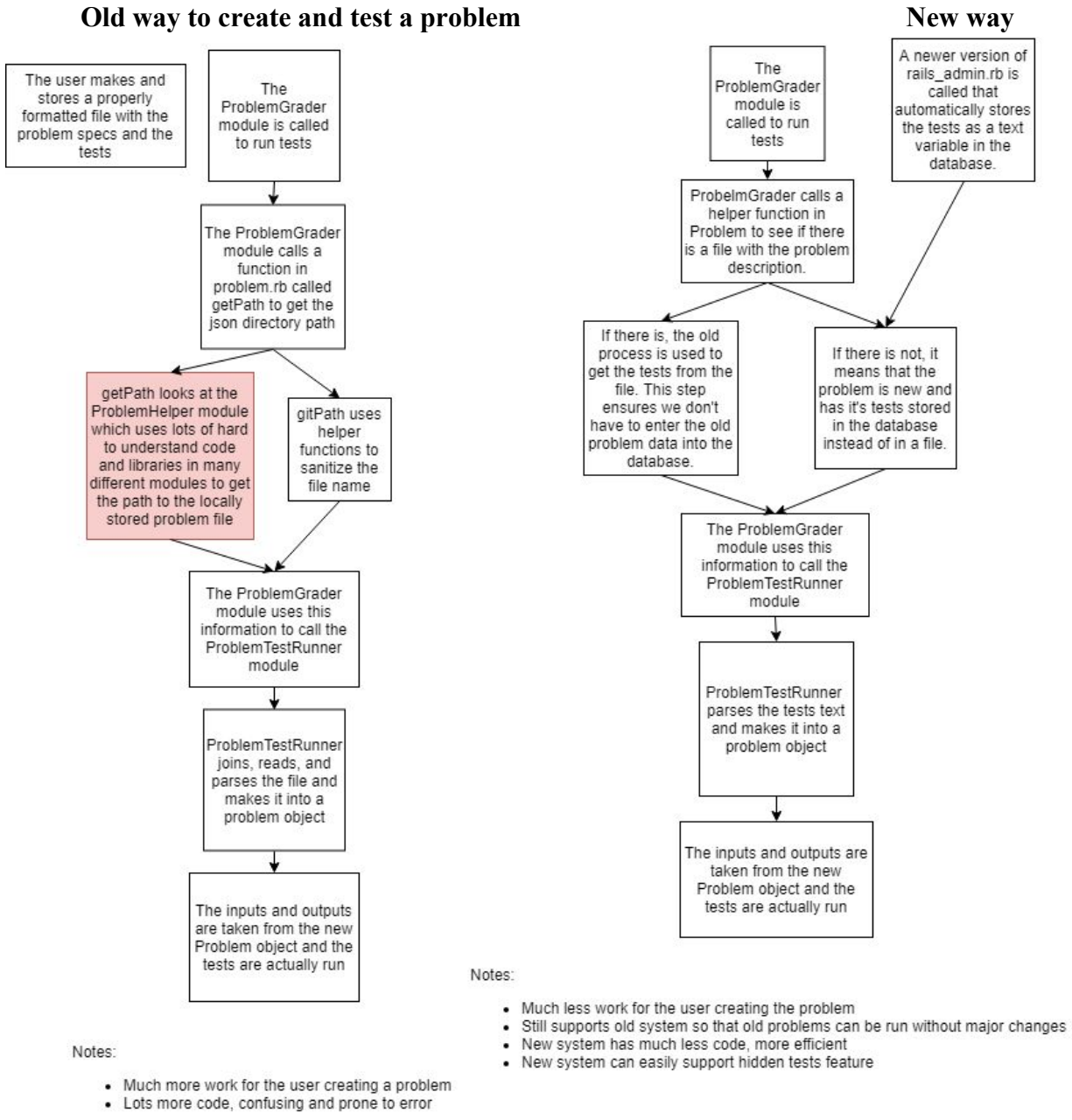

Figure 2: Problem Creator Flow Architecture

Problem Import/Export Architecture

Rails admin supports a default export action that allows a user to export specific field of a model as a CSV file. Our client wanted much simpler interface, that only used one button to download all relevant problem fields. We changed the default rails_admin library files to satisfy this requirement.

For importing problems, we were able to utilize another library - rails_admin_import. This library gave the user the ability to upload a CSV whose headers corresponded to model fields. The import function then parses the uploaded file and updates the appropriate model attributes based on the rows of the CSV. Again, we had to simplify the UI to meet our client's requirements.

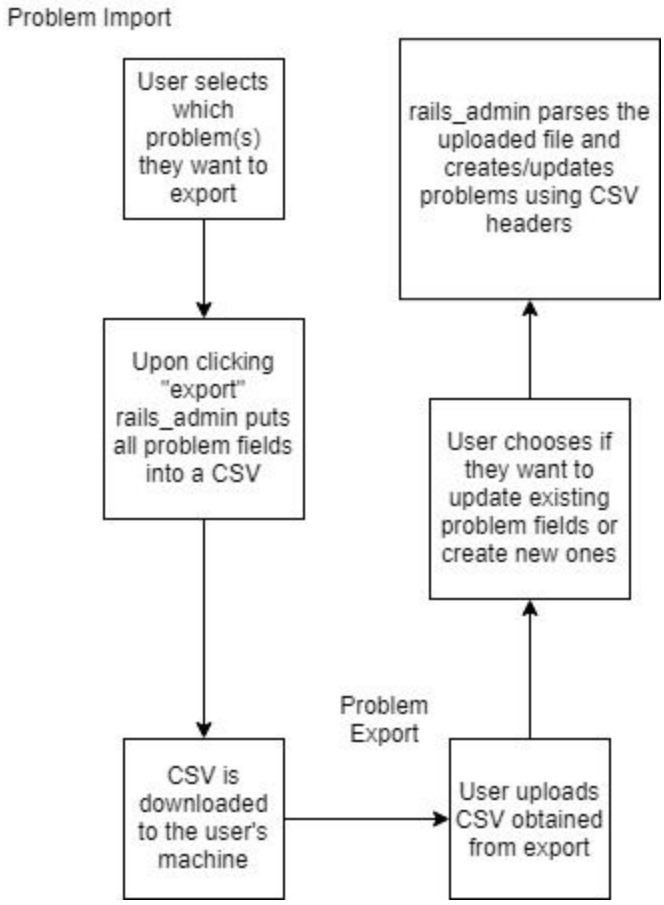

Figure 3: Problem Import/Export Flow Architecture

Hidden Test Architecture

To add the hidden tests feature we built off of the architecture we made for creating a new problem. Instead of distinguishing visible tests from hidden on one big list we added another field for the problem module. Now there is a field for visible tests and a field for hidden tests and instructors can fill each with as many tests as they want. The problem test runner looks in both

fields and counts the number of tests in each while parsing them into inputs and outputs and putting these into an array. The tests are run and displayed. Using the counts of each type, the regular tests are shown as normal. The hidden tests still take up space on the red/green testing bar, but when the student hovers over each hidden test all that is displayed is a message that the inputs and outputs are hidden. This feature prevents students from hard coding every test's output.

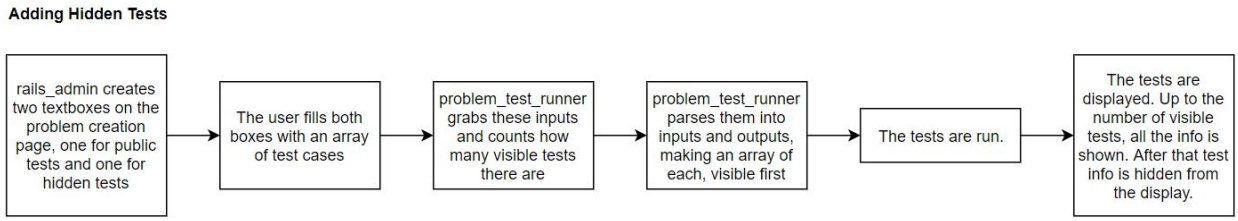

Figure 4: Hidden Tests Flow Architecture

File I/O Architecture

A problem test begins when the user hits the submit button. The practice problem is either a function or a program in its entirety (In C++ this means it is in main), so the test report goes down one of those two paths. From here there are two more paths: the program determines whether or not the problem has File I/O; If it doesn't, it runs the standard test protocol and creates a test report for the user. If the problem has File I/O then it has to figure out whether the File I/O involves reading from a file or writing to a file. Once that is determined, the program runs that specific test and generates the test report specific to that type of File I/O.

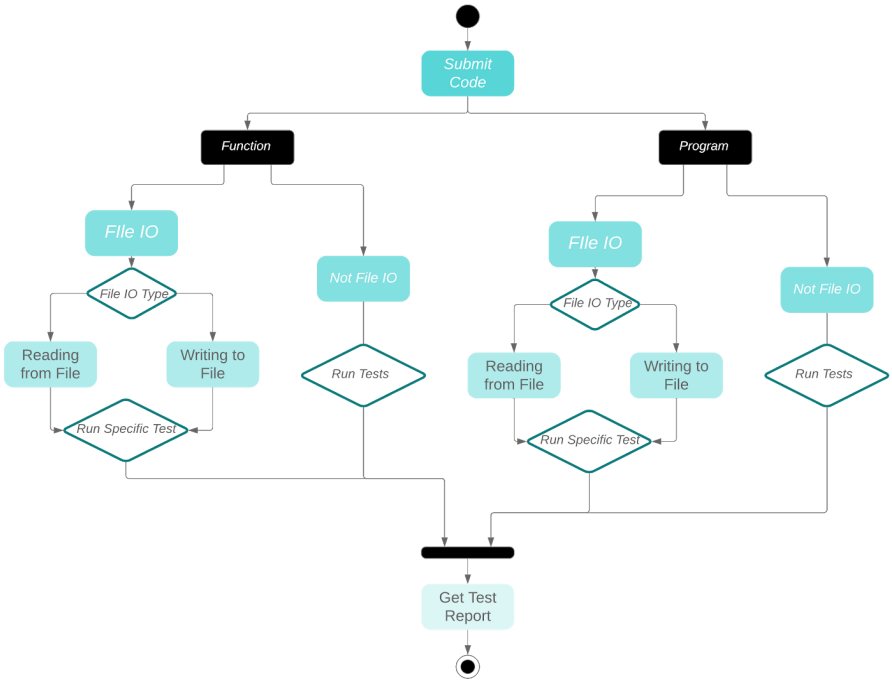

Figure 5: File I/O Flow Architecture

Grading Policy Architecture

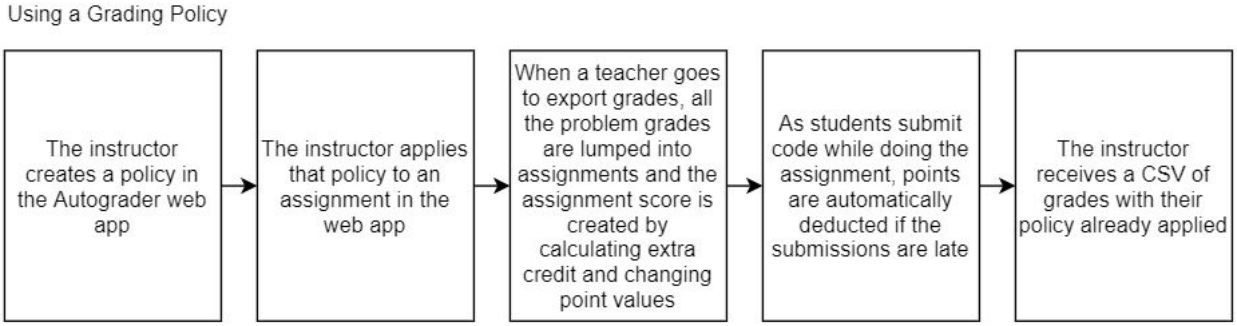

Figure 6: Grading Policy Flow Architecture

Before Autograder 4.0, Autograder graded every problem out of 10 points, regardless of extra credit, late submissions, or assignment weights. Additionally, autograder only graded problems, not whole assignments. Teacher had to manually recalculate each student's score, a time consuming process. To solve this we created grading policy objects. Teachers can put in their grading policy, naming it and adding many different components such as points per problem and days without late deductions. Once this policy object is created, instructors can apply it or a different policy to an assignment and Autograder takes it from there. As students submit scores in an assignment, the late policy is automatically pollied, deducting points if the submissions are late. The highest score after this becomes the official grade for that student and problem. When a teacher goes to export assignment grades, Autograder goes user by user and finds every grade for every problem they did within that assignment. It can then count out required and extra credit problems and change their point values accordingly. All theses updated scores are then summed into the final assignment score, which is what the export feature putis in the spreadsheet. More details about this feature can be viewed in the technical section of the report,

Canvas Export Architecture

Originally, the grade exportation feature of Autograder simply pulled every grade in existence, with no filters applied, and exported them into a .csv file. Instructors then had to write their own Python scripts to organize and filter all of the grades and group problem grades into full assignment grades before manually entering them into Canvas. The new interface allows an instructor to upload a Canvas .csv file, select a specific assignment to export the grades from, and select a column in the spreadsheet to hold the grades. A custom exportation action will then filter all of the grades by assignment and user and apply a predetermined grading policy to create grades that belong to the whole assignment, as opposed to individual problems. The grades are then inserted into the spreadsheet by matching the associated multipass IDs and the edited spreadsheet can be exported. The edited .csv file is formatted in such a way that it can be immediately uploaded to Canvas without needing any changes.

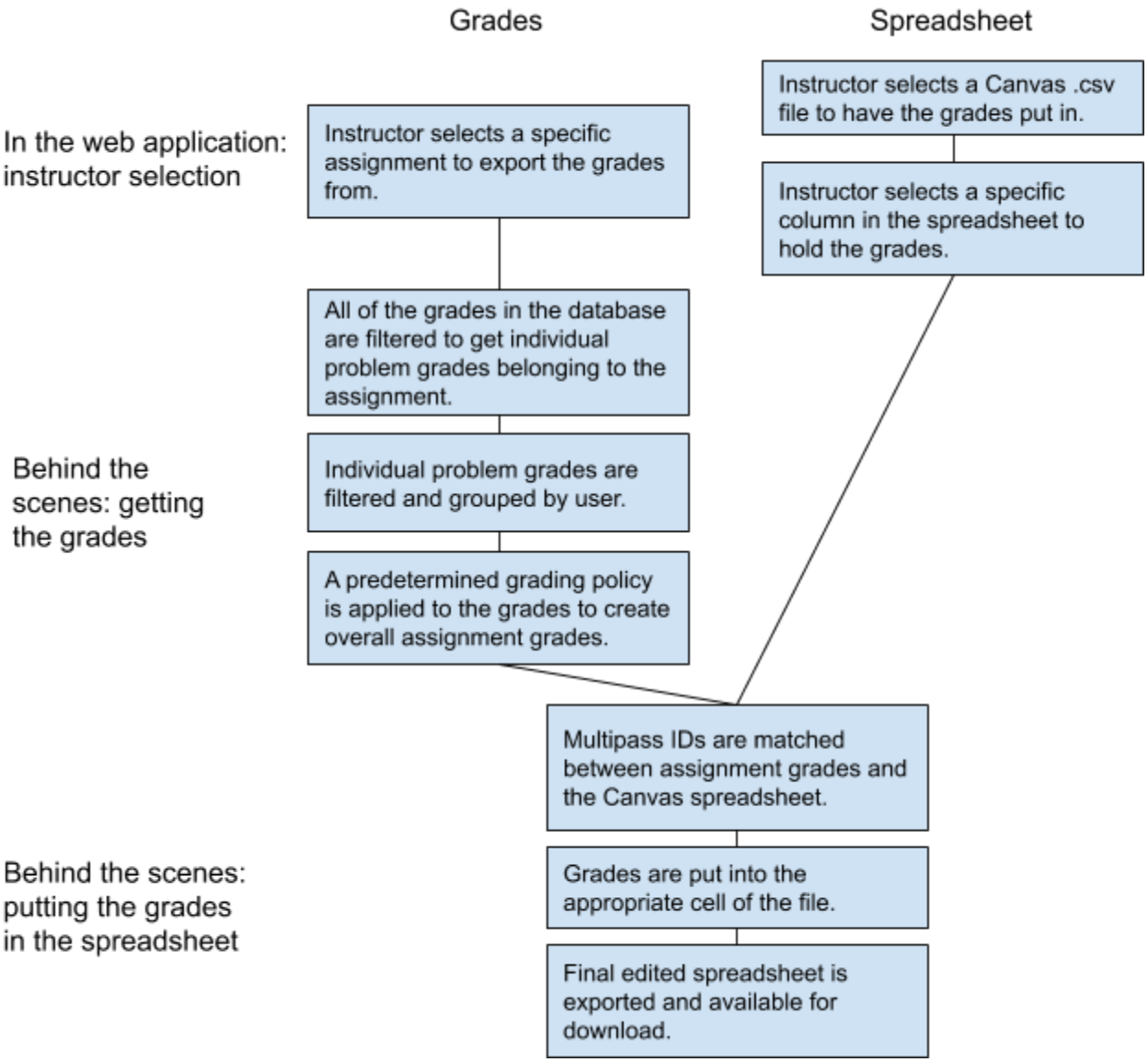

Figure 7: Grade Export Flowchart

Technical Design

File I/O

When a new problem is created, there are a number of fields that the professor can input regarding the File I/O for a problem. First, the professor is asked whether or not the problem has File I/O in it. Next, he/she is asked for the name of the text file, which will be given to the students in the description so they can access the appropriate file. If they were to reference an incorrect file name, they could not be accurately graded. Finally, the professor is asked for the text file contents. In this field, the contents of the test text files will be inputted.When reading from a file, this essentially will be the test cases used to be given to a user to test if they read from the file correctly; when writing to a file it is the expected text file contents that the student should have written to the file when their program is executed (Fig. 8).

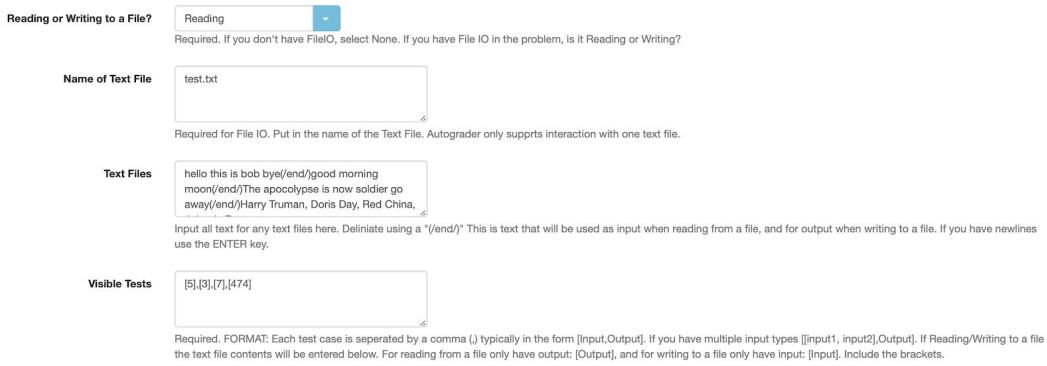

Figure 8: File I/O Problem Creation

File I/O has four parts: reading from a file using a function, writing to a file using a function, reading from a file in main function, and writing to a file in a main function. All four of the different types of file I/O questions run a different handmade GTest. GTest is a unit testing framework for C++ that has been hacked in to work with Ruby so it can be used in a web based environment. GTest doesn't natively support file I/O, so forcing it to work was one of the more difficult parts of this project. Normal problems are run through the *problem_test_runner.rb* file, which determines whether the problem is a function or a full program; from there, the specific child of *cpp_test_runner.rb*, either *cpp_iostream_runner.rb* or *cpp_function_runner.rb* is called (Fig. 9). File I/O works a little differently. For example, if there is a File I/O problem and it is handled as a function, not in the main, it is run exactly the same as a normal problem, except the input is no longer handled by the problem JSON that is created, but rather from the text file contents field itself. If it is run as a full program, or in a function named main, it is handled in a completely different way. In C++, the standard console output is returned to the computer as a string, whereas some of the outputs that reading from a file will have potentially will not be strings. To fix this, the GTest handling reading from a file in a main converts everything to a string and uses string comparison, which can still handle all of the variable types and compare them accurately. In regards to questions where the student writes to a file, this is handled the same way regardless of whether it is a main function or a standard function. It reads the file and

stores it as a stream; from that stream, it takes in a string that holds the contents of the file and compares that to the already inputted text file contents for the problem.

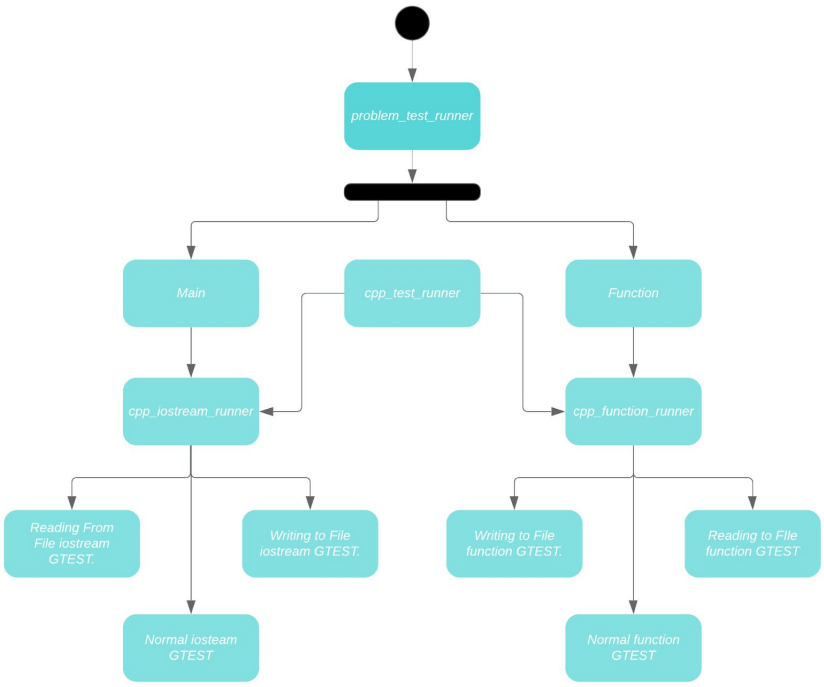

Figure 9: Specific Flow For File I/O

Grading Policy

Originally, Autograder graded only problems, not assignments, and these grades were all out of 10 points no matter what. As explained in the architecture section of this report, this caused many headaches and required a big time commitment from instructors when it was time to grade. Now, instead of applying their policy manually, Autograder can do it for them. First, we made a new model for grading policies. Instructors can create instances of this model in the web app and add the following features:

- Policy name
- Percent deduction per late day
- Weather or not weekends count as late days
- Other days that do not count as late such as holidays and snow days
- How many days past the due date an assignment is still accepted
- How many of the problems in the assignment are required (the rest are then extra credit)
- Points per required problem
- Points per extra credit problem

We talked to all the professors who currently use Autogradert to compile this list of features. The page to create a policy looks like this:

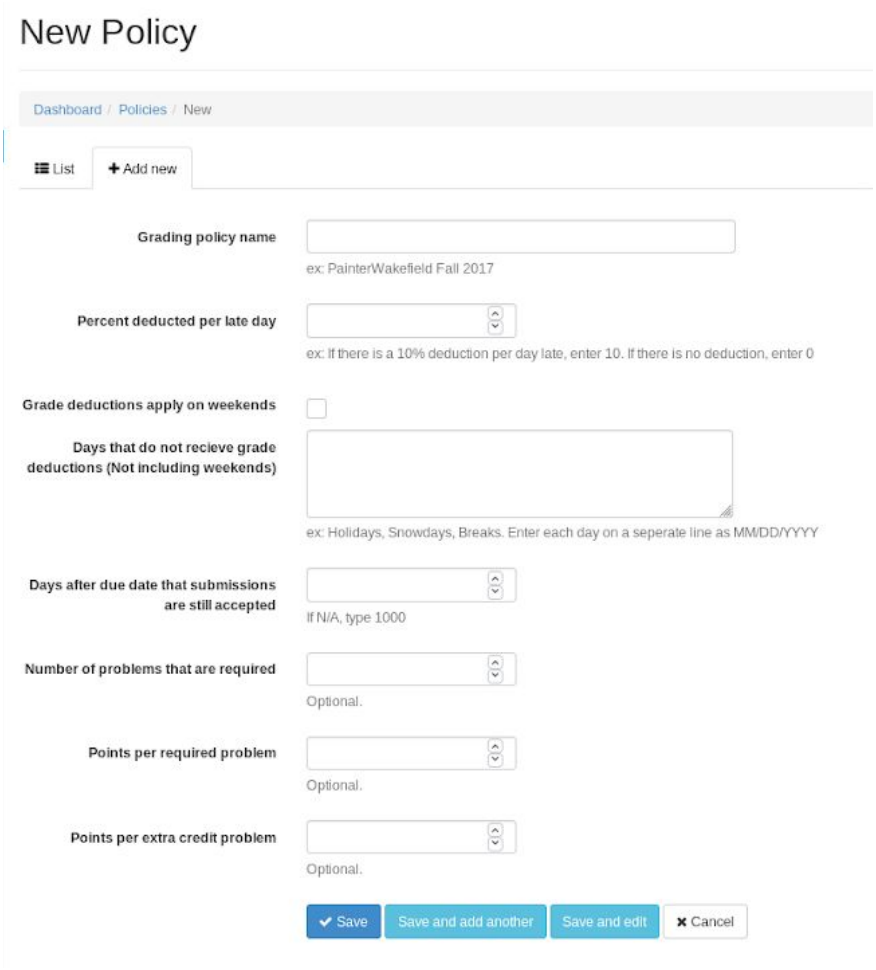

Figure 10: Screenshot of the Create a New Policy page

Once the instructor saves their new policy, everything is stored in a policy database. The schema for that database is as follows:

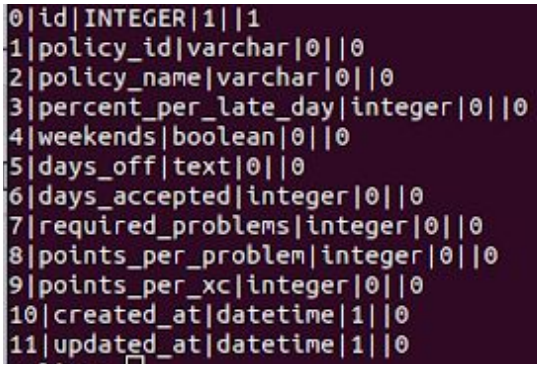

Figure 11: Policy Database Schema

When a teacher goes to create or update an assignment they can add a policy to it. This happens in the create an assignment page and looks like this and the drop down contains a list of every policy that has been created:

Figure 12: Partial screenshot of the Create a New Assignment page

That is all the instructors have to do. The Autograder takes it from there and applies the grading policy as follows:

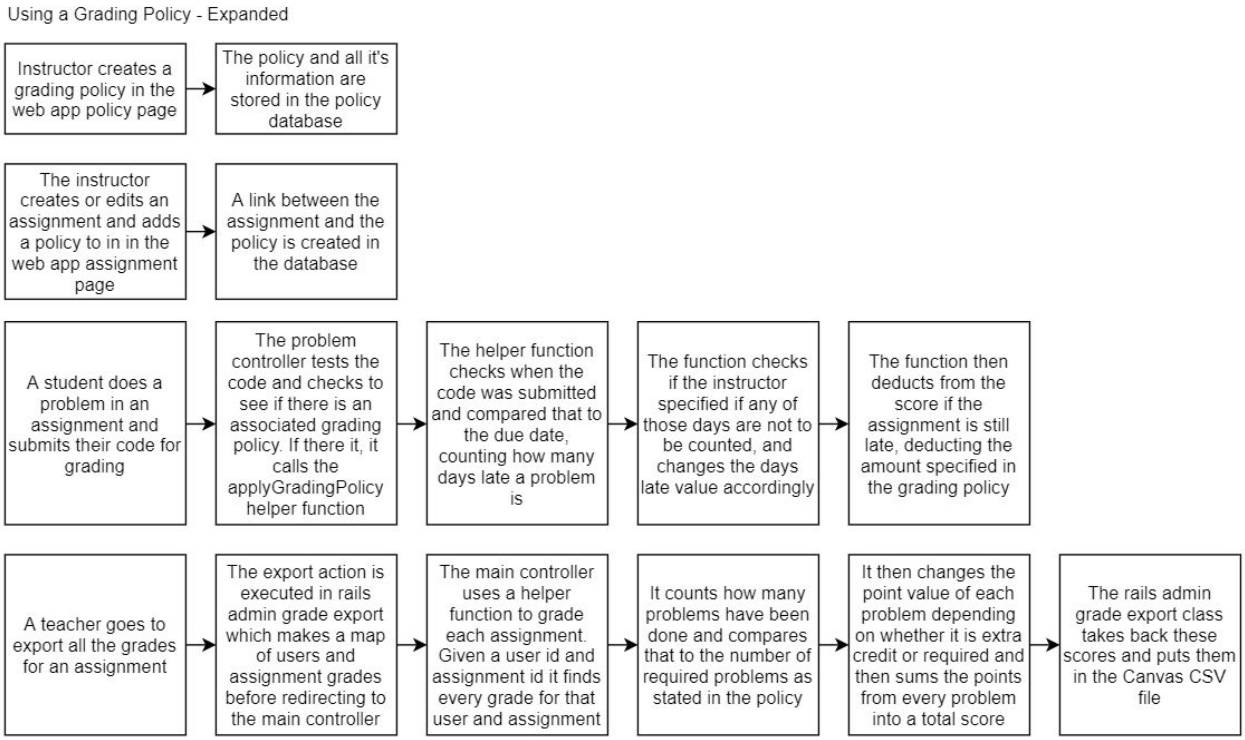

Figure 13: Expanded flow and architecture of the grading policy feature

The first part of the policy that is applied is the late policy. It is applied as students submit grades. This is done because students submit many submissions, and if we applied it later we would have to access and loop through these submissions multiple times. This is difficult and inefficient. Another reason we apply it immediately is because we need to know which submission has the highest score before we proceed with the code. If a student submits an 80% submission before the due date and a 100% submission after the due date, either could be the highest score depending on the late policy. We need to know which is higher so we have to apply the policy right away. The applyGradingPolicy helper function works by checking the current time (when the code is being submitted) and comparing it to the due date. It then counts how many days past the due date the problem is. If it is late (> 0 days past the due date) the helper function parses and checks the list of days that don't have deductions and if a day on that list

applies to this case the days past the due date number shrinks. The same is done if the instructors specified that weekends do not receive deductions. The helper function returns the final score for the problem and the submission with the highest score becomes the official grade.

The rest of the policy is not applied until the teacher exports the assignment grades using the Canvas spreadsheet export feature. The code for this feature in main controller loops through all the users who were assigned that assignment and calls a helper function that takes a user id and assignment id and then find every problem grade submitted for that assignment by that user. Once it has all these grades it counts how many there are and compares that to the number of required problems as specified in the policy. It then sorts the grades from highest to lowest and loops through them, applying the points per required problem to each grade up to the number of required problems. For all problems after that it applies the points per extra credit problem. The list was sorted to ensure that the student's best problems counts towards the required problems and their less good or not attempted problems are counted as extra credit. These new point values are now summed into a total score for the assignment. The export function then handles putting these scores into the spreadsheet.

This was a very large and comprehensive feature that required a lot of planning and work. We had to accommodate many different requests as well as make sure it integrated well with the other features we added such as exporting grades. We worked closely with instructors to make sure it was usable and understandable.

Quality Assurance Plan

For this project, it was essential that we focus on producing quality software. Our updates will be pushed to a live server ready for use in the Fall 2019 semester, so what we created will impact many lives and needed to be fully functional, usable, and maintainable upon the completion of the project. To make this possible, we followed a multi-step plan that included constant communication, testing, and merging to ensure that we produce a quality product as opposed to the bare minimum that would satisfy the project requirements. We designed multiple ways to know that our project is not just done, but done well. These include:

Constant communication
We communicated regularly with the other team working on this web application, our advisor, our client, and most of all, each other. This not only helped us work correctly and efficiently, but also helped us identify potential issues before they could become a problem. It also created an environment conducive to everyone sharing new ideas that could improve the quality of our project. When people feel comfortable sharing, they contribute more ideas more often, and we were able to integrate these good ideas into our project to improve its quality.

A good starting design
Before we started each requirement for this project, we made sure we understood what we were supposed to be doing going in. Much of the time we spent on our project was spent on critically analyzing the existing code so we understood it and knew how to work with it well, to avoid

damaging the existing functionality. We also always had at least some type of design plan before we started making new code. By making sure our designs had good quality, we were much less likely to run into errors and create a poor product.

Regular user feedback
For this project we were uniquely situated because we were both the creators of the software and the users. We have all used this program in our classes and some of us have previously used the instructor side as TAs. This means we could constantly self-reflect and accurately ask "When I use this software, what do I want?" We could also ask each other and our peers for feedback about what could make this application better. By focusing on the user's view and opinion of Autograder, we could prioritize quality over completion and accurately create a quality product that fulfills all user desires.

Code Reviews:
While we were working on the project, we regularly reviewed the code written by other members of our team as well as the members of the other Autograder team. We did this when needed, approximately once a week, in a relaxed and informal way so that everyone felt comfortable identifying and owning up to mistakes. Code reviews helped to ensure the readability of our program.

Constant testing, including:

- **Static and dynamic analysis**
  Static analysis is defined as testing without dynamic execution of the software so defects can be detected at an early stage before they cause issues at compile time. Half of the static analysis was done by the computer, alerting us to potential errors such as spelling and formatting. This helped us improve the readability of our code, which goes hand in hand with maintainability. We also looked at each others' code, which helped with readability, reusability, and efficiency because a fresh pair of eyes can often see things the original writer cannot. If our code could not be understood by each other, we could not hope for it to be understood by future developers, which is a mark of poor quality. Dynamic analysis is testing by running the code. Running the code often keyed us into quality issues almost immediately with such things as compiler errors and revealed things we missed in static analysis. Dynamic analysis identified vulnerabilities in a runtime environment, helped us test how each part of the code interacts, and can reveal false negatives that we identified in the static code analysis. We did static and dynamic analysis almost constantly as we add new elements to the code. We used static analysis as we coded, and once we believed the code could compile without issue we switched to dynamic analysis.

- **Graphical user interface testing**
  The objective of GUI testing is to validate that the graphical user interface is working, makes sense to the user, and fits the project requirements. This was a very important part of testing for our group because many of our requirements were to modify the GUI, and none of our new features could be used without a functioning GUI supporting them. We

did GUI testing after every modification we made to the program. To do this, we compiled our changes and then ran them in localhost:3000 so that we could visually judge whether we did a good job. This is more efficient than conventional unit testing because many different things can go wrong with the GUI and it is easier for a human to spot issues than a test suite. This testing did not suffice for every change we made, but it was certainly essential to determining the overall quality of our product.

- **Incremental Integration and Component testing**
  These types of testing take a bottom up approach, which means we continually tested the application as a new functionality was added. When we built new features they consisted of multiple pieces that were each independent enough to be tested separately. This helped us isolate problems and fix them quicker. We also merged with the other group regularly and then tested the code to see if any new errors were introduced by integrating the student and instructor sides. By doing this incrementally instead of all at once at the end of the project, we avoided large-scale and/or confusing errors and could easily try out additional functionalities we thought would improve the quality of the final product. Merging ensured that each component we added worked with all of the other components in the program.

- **Browser compatibility testing**
  Unlike the other types of testing we did, browser compatibility testing was less complex and less continuous. We did this type of testing once we were done with a new major feature and had completed GUI testing. We then tried to load the program in browsers other than the one we used for successful GUI testing. If it worked, testing is complete, and if it does not work then we altered the code until it worked properly. This type of testing helped to ensure the usability of our program. Had we not implemented this type of testing, our program would pass the requirements but would be low quality.

- **Comparison testing**
  This kind of testing was performed by comparing the application's strengths and weaknesses with its previous versions. This was a good way to test the quality of our product because both the version we started with and the version we created are functional and achieve a purpose, but our job would only be a quality one if the newer version had more strengths and fewer weaknesses than the old version. We did this testing closer to the end of the project, but continually thought about this testing as we were creating new features to guarantee that our work would improve the application. This tested almost every aspect of our project at the same time because it is a holistic view of what we have done.

- **Exploratory testing**
  This is a type of testing that we did at the beginning of every work day. When we logged into the local host, we clicked around and tested many different functionalities with no set plan or direction. This exploratory approach revealed defects in the application; as it is a holistic approach it often revealed new issues that the previous night's more

specialized testing did not. It also helped to ensure the GUI is at peak quality and is navigable and usable for students and teachers.

- **Monkey testing**
  Monkey testing is named for the assumption that if a monkey used the application and entered random inputs, the application would still work and be able to handle any input the monkey put in. We did this testing closer to the end of the project once we had added all of the new functionality that requires user input. If a case we tried caused the program to break, we stopped to address it and then repeated the process until we were confident that the user could use the application no matter what they do. We also heavily focused on testing inputs that included potential SQL injections, as we needed to make sure that users could not change the database information without the proper privileges.

- **Usability testing**
  As its name implies, this tests how user friendly our final application is. It is a test to see if the flow is of sufficient quality that a new user can easily understand the application. We completed this testing closer to the end of the project, because when we were doing things that require modifying the GUI the application at times has to become less user friendly before it can be more so. (Ex: when we designed a new button, sometimes it looks bad and is in a strange place because we are working on its functionality before its appearance). Any concerns that were raised when we did this type of testing were addressed to make the final application the possible highest quality.

- **Acceptance testing**
  This is the very last type of testing we performed. The acceptance test was performed by the client and verifies whether the end to end flow of the system meets the requirements outlined at the beginning of the project. If this test worked, the client will accept our new software and deploy it for use by teachers and instructors next semester. All the testing we do ourselves led up to this final test.

At the end of the day we can feel confident in what we have done because we will know that we have tested and delivered a quality product that will positively impact the lives of countless students and instructors. All of our testing means that we can ensure our product is user friendly as well as functional, and our focus on what the user wants instead of just meeting project requirements will both result in a quality application and help us develop good coding habits we can take into our future careers.

Results

After weeks of hard work, we successfully implemented all six of the main project requirements to our client's satisfaction. They have every additional feature that our client asked for and are ready for the final roll out. We were unable to complete the two stretch goals because we did not have enough time and all of the other requirements took priority. Overall we are proud of what we have done.

Gui Testing Results:
We have run many GUI tests over the course of this project, and we have fixed all issues so that now all of the tests pass. Our graphical user interface looks good and lines up with our added functionalities. We ran tests after every change we made, and the whole team contributed to this type of testing.

- **Incremental Integration Testing Results:**
    - Incremental integration testing has yielded ultimately good results. At certain points of the process it instead yielded mixed results, but because we were doing the testing incrementally that meant we were able to find issues caused by our two teams conflicting code and fix them without too much pain. We were able to merge successfully in the end and pass our integration tests.

- **Component Testing Results:**
    - We have also run all of our component tests successfully or have fixed all of the unsuccessful ones so that they run successfully. This was done by each team member running dynamic analysis of each component they made right after they made it.

- **Monkey Testing Results:**
    - We asked a smart outside student who was not involved in development to attempt to damage Autograder using SQL injection. They were unable to do it. This was the first part of Monkey testing, and we passed. The second part of GUI testing is to put random inputs into the new input fields we created and see if any random combo could create an error. We failed this part of the test initially but then successfully updated our code so it could handle unconventional inputs and it passed.

- **Browser Compatibility Testing Results:**
    - We ran multiple browser compatibility tests on the five most common browsers that students use. Autograder worked on all five, which means that students can use the web app without worrying that their browser will cause extra problems. Autograder works on Firefox, Chrome, Edge, Internet Explorer, and Safari.

- **Comparison Testing Results:**
    - Our ongoing comparison testing passed with flying colors. Our product is more advanced than Autograder has ever been and has many new features that make the site easier to understand and more comprehensive.

- **Useability Testing Results:**
    - We ran usability tests with multiple people over the course of a week. We asked multiple students who were not involved with the project development to try and make new assignments, problems, and policies. Their feedback was instrumental

and we made major changes in response to what they said they had difficulty with. Now we know that our code is as user friendly as it can be.

- **Acceptance Test Results:**
    - We performed an acceptance test with our client after we had finished implementing all of our features. Our client tried to perform every action we added, stress tested the system, and evaluated the flow of the web app to see if it was understandable. Our client was satisfied with our product and the stress tests all passed. We made minor updates based on his feedback and now our final product is ready to roll out for the coming semester.

We are proud of our final product, but it can always be improved even more. If work was to be done on it in the future, some potential ideas for improvement we have include:

- Porting all of Autograder to the latest versions of Ruby, g++, and Rails so that it is as secure as possible
- Improving Python support so that python users can access all the same features as C++ users, especially the file input and output
- Adding a dark mode or other color scheme options to the website as a whole, including red/green colorblind support on all pages of the web app and in the testing results bar
- Ensuring that the site zooms well for users with visual impairments (potentially including a text to speech reader for the problem description)
- Figure out what the static model is and make it work
- Same with hover description
- Spend time going through all of the code and adding proper comments and proper documentation so it can be easily understood by new users when it becomes open source
- Add even more protection against hacking, such as the best protection against SQL injection
- Expand autograder to include languages other than just C++ and Python, potentially including Ruby and Java, maybe even SQL someday (even if Mines does not currently use these languages, they would be good additions to an open source Autograder)
- Convert every .haml file to .erb for maintainability and readability
- Mobile support so that students can actually use Autograder on mobile and do so easily
- Change grades from integers to some type of floating point variable so that decimal grades are supported and not rounded
- Speed up testing and reporting errors and server errors so students don’t have to wait so long
- Add a mass delete button for problems
- Update the code so that if a teacher changes or deletes a policy while an assignment it applies to is ongoing the grades are updated accordingly

Over the course of this project, we learned quite a lot of things. Some of what we learned includes:

- Rails is a powerful and elegant language and is a good choice for making web apps that match a pre-existing format. Trying to use Rails to create entirely new features and things can be difficult, however, especially when one needs to override one of its pre-existing features, such as export. Like all languages, Rails has its place, but it might not always be the best choice for a project if it has many custom requirements.
- The longest part of picking up any project is setting up the environment. This includes every step necessary to take before being able to start working on the actual requirements, including installing IPEs, downloading existing code properly, getting administrator access, and so much more. One needs to plan for this time at the beginning of any project and get as much help as possible from the client where appropriate.
- Documentation is vital because picking up a pre-existing project without proper (or any) documentation makes work exponentially more difficult than if there were documentation. We learned this the hard way when we started the project. We also learned that it is easy to not do documentation, especially when running out of time on a project. It is important not to ignore it, however, and must be done for the project to be maintainable.
- Communication is vital. We benefited from regular client meetings because it prevented us from ever going too far down the wrong track. It is also very important to communicate within a project group because if someone does not ask for help or ask how he/she can contribute, it can be very easy for someone to get lost and fall behind the rest of the group.

Appendix 1

- Autograder Test Environment Install:

AUTOGRADER TEST ENVIRONMENT SETUP

Installation

- Install Ruby. The client is running version 2.5 as of now, this is the only compatible version as of now (at least until the codebase is updated to a later version).
    - If you are using a version manager such as rbenv, make sure the global version is set to 2.5 as this should work.
    - For rbenv, add "$HOME/.rbenv/shims" to your PATH variable. You may want to add this to your .bashrc file so you won't have to read every time you startup your machine.
- Install Rails. Version 5.1.3 seems to work.
- Install/upgrade g++, git, cmake, and python3/python.
- Clone GTest repo and install.
    - Try running sudo apt-get install libgtest-dev as this might work depending on your distro.
    - Repo can be found here if this does not work.
    - Install the libraries and headers to /usr/local/lib and /usr/local/include, or some other location, but remember this location. For Ubuntu:

```
sudo apt-get install cmake # install cmake
cd /usr/src/gtest
sudo cmake CMakeLists.txt
sudo make
 # copy or symlink libgtest.a and libgtest_main.a to your /usr/lib folder
sudo cp *.a /usr/lib
```

- Install latest version of sqlite3, should be able to download from distro.
- Create an account on JetBrains using you student email. Download and install RubyMine.
- Clone the Autograder repo, navigate to the project folder in the terminal.
- Download an empty test database with a .sqlite file type and save in the /db subdirectory. Download the corresponding .yml data file and save into the /config subdirectory.
- Navigate to the project folder and execute the command bundle install. This should install all the gems required for the project, though some will need external dependencies you may not have.
    - When errors pop up, install any dependencies for that gem such as "libpq-dev" for the pg gem.
    - Keep going until all gems will bundle with no errors.
- Cd into the common directory, run the command "make clean", followed by "make".
- At this point you will need to give ruby permission to do the chroot action the autograder uses. Figure out what the executable for ruby actually is with "ls -l /usr/bin/ruby" - e.g., on my system the actual executable is /usr/bin/ruby2.5. Then run:
    "sudo setcap cap_sys_chroot+ep /usr/bin/ruby2.5" (replacing with the path to your executable)

    - You will have to redo this command anytime ruby is updated (if you install updates).
    - If Ruby is installed with RVM or RBENV you may need to look for that executable instead. It may be tricky to find.

RubyMine Setup

- Open the project in RubyMine. Make sure the version control is set to a branch other than the master.
- Create a file called secret_token.rb inside of config/initializers. Write the following code:
    - Rails.application.config.secret_key_base = ‘’
- Open a terminal window in RubyMine and run the command “bundle exec rake secret”. Copy the (long) alphanumeric token it gives you.
- Navigate back to the secret_token file and insert the copied token into the single quotes.
- You’re almost ready to test. Click the Configuration dropdown menu next to the run/debug buttons in the top left of the menu bar of RubyMine. Select Edit Configurations.
    - Click the ‘+’ to add a new config. Select Rails from the dropdown, set the Environment as development. Leave everything else default and hit apply.

Running Locally

- Click the Run button in the top left of the menu bar. Open a browser and navigate to localhost:3000 where a local version of the site will display
- Login with GoogleOAuth, and you should gain access to your development site as a user.

Running Locally as Admin

- Now that you’ve signed in once, your name should appear in the test database supplied. Now cd into your project, and run the command “rails dbconsole”
- Enter the command SELECT* FROM USERS.
- UPDATE users SET role = “admin” WHERE name = “Johnny Appleseed”