CSM Autograder 1: Student-Facing Feature Update

Final Report

June 18th, 2019

Colorado School of Mines Field Session Summer 2019

Advisor: Donna Bodeau
Client: Dr. Christopher Painter-Wakefield, CSM Professor
Team: Jay Harrison, Kellen Parker, Kyle Strayer, Joanna Turner

Table of Contents

Introduction **2**
Product and Client 2
Product Vision 3

Requirements **4**
Functional Requirements 4
Non-Functional Requirements 5

System Architecture **6**
Test, Save, and Submit Button 6
Version Control with the Database 7
Which Version Gets Submitted? 7
Accessing Previous Submissions 7
Autosaving Students' Code 8

Technical Design **10**
Data Flow through the System 10
Benefits of Database vs. File System 11

Quality Assurance Plan **12**
Deployment Testing 12
Code Reviews 13
Merge Meetings 13
Client Meetings/Demonstrations 13

Results **14**

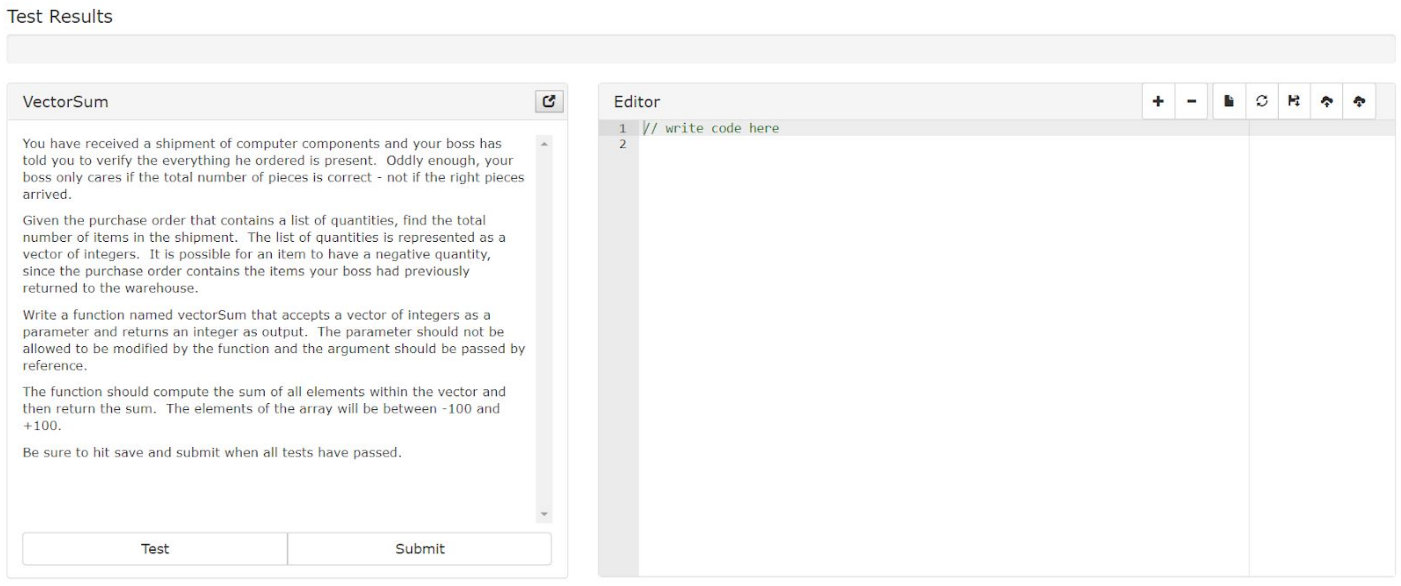
Conclusion **15**
Lessons Learned 15
Reflection and Future Work 16

Appendix **17**

Introduction

Product and Client

The Colorado School of Mines Autograder is a web application that allows Mines' Computer Science students to complete their coding assignments online and receive immediate feedback. It also allows instructors to easily access grades and import them into Canvas. It was created by Dr. Christopher Painter-Wakefield in 2014 specifically to make the assignment and grading process run smoothly for both students and professors. It has been continuously updated over the years with several previous field session groups having worked on the system over the past 3 years. The current system, Autograder 3.0 (*Fig. 1*), is a vast improvement over previous versions but still leaves much to be desired. Many students who use or have used version 3.0 notice that the UI is not overtly intuitive and that they often lose the code they write if not careful. This is not a bug in the system, but rather a lack of user-friendly design. Exiting a page without saving is remarkably easy, as is submitting code without saving or testing code without submitting.



You have received a shipment of computer components and your boss has told you to verify the everything he ordered is present. Oddly enough, your boss only cares if the total number of pieces is correct - not if the right pieces arrived.

Given the purchase order that contains a list of quantities, find the total number of items in the shipment. The list of quantities is represented as a vector of integers. It is possible for an item to have a negative quantity, since the purchase order contains the items your boss had previously returned to the warehouse.

Write a function named vectorSum that accepts a vector of integers as a parameter and returns an integer as output. The parameter should not be allowed to be modified by the function and the argument should be passed by reference.

The function should compute the sum of all elements within the vector and then return the sum. The elements of the array will be between -100 and +100.

Be sure to hit save and submit when all tests have passed.

```
// write code here
```

Figure 1. Autograder Version 3.0

Students can access the site by going to flowers.mines.edu and logging in using their Mines Multipass credentials. Flowers - named after British Computer Scientist Tommy Flowers - is a secure, local server used purely for the Autograder system. It utilizes the Mines Shibboleth authentication process, ensuring only registered students can access the site while on campus or going through the Mines VPN. After logging in, a student can select which course, assignment, and problem they would like to work on.

From there, students can write and edit their code within the provided editor. Testing is made possible using the Test button. This button does only that, however, and only runs the editor code through pre-established tests. The program runs this series of tests and graphically displays the output to the student immediately. A green or red box representing each test shows whether the student passed, however a score is not passed into the gradebook. These tests are purely for the student and formatted such that they can look at an individual test case to see the input, desired output, and actual output. There is no penalty for failing these tests and a student can test their code as many times as necessary.

To actually submit their assignment, a student must hit the submit button to have the latest version of their code submitted for grading. The testing graphic again shows the student how they did and this time the score they receive is recorded for the instructor's review. The process requires a student to strictly follow an order of saving, then testing, and then finally submitting their code. Often times students forget a step in the process, complicating things on both ends of the instructor-student relationship.

Product Vision

The Autograder 4.0 addresses many of the issues that students and professors face when using version 3.0. Our client, Dr. Painter-Wakefield, requested that we take measures to prevent the loss of student code, as it eventually becomes his problem to fix. Most of the updates added to the new system are specifically in place to protect students' code and improve user experience. The Autograder no longer requires students to go through the process of testing, saving, then submitting their work, but instead relies on a new streamlined process. He requested that students have the ability to test, save, and submit their code all in one action. Dr. Painter-Wakefield also requested a saving feature that would prevent lost code in the case of a browser crash/accidental close of the site. On top of this, he wanted students to have full access to previously tested versions of their code with the ability to change between these versions at will.

Overall, these added features should not remove functionality from the existing platform, but extend the capabilities and improve the overall quality of the product. While given full reign over the implementation of these features, Dr. Painter-Wakefield was involved in every aspect of the project to ensure quality. As the Autograder is his project, our main goal was to preserve its functionality and fulfill his vision.

Requirements

Functional Requirements

Test/Save/Submit Button: Our client requested that Autograder 4.0 have one button that saves the current code, runs all tests on it, and stores the grade into the existing database for later access. Combining these features was the primary goal of this project. Since all of the desired functionality already existed, our task was a matter of modifying these underlying components of these features to ensure compatibility with the other requirements. While these features would not change much, we were required to modify them in order to work with the database.

Storing Previously Tested Code: We were tasked with creating a solution that would allow students to switch between previously saved versions of their code. To add version control, a new way of storing each submitted version needed to be created. We were told there was a need to store not only the code, but the metadata surrounding it as well such as submittal time, grade, and all underlying database dependencies. This meant creating a new table in the database to store a new version of the student's code every time it was submitted. In order to prevent duplicate submissions from being submitted, we were also tasked with finding a way to prevent copies of the same code from being submitted as separate copies.

Autosave: On the current iteration of the Autograder, students are required to manually save their code whenever they wish to save their progress. When the 'Save' button is pressed, the current editor code is copied into a solution.cpp file stored within a file system on the production server underneath the student's name and the specific problem. If students forget to save manually, code is lost and unable to be recovered. Our task was to not only create an autosave feature, but remove the reliance on the file system altogether. Autograder 4.0 needed to save the current code to the database whenever changes were registered in the editor. This meant that whenever a student exits the page, the code they wrote earlier needed to be saved within the database and retrieved whenever the student returned.

Stretch Goal: Port to Latest Ruby, G++, Rails Versions: Porting to newer versions ensures that no unforeseen problems pop up in the future due to old versions of software. We were asked to upgrade the underlying frameworks to not only future-proof the software for further development, but also to ensure that the project has the latest security patches. This meant upgrading not only the current Ruby/Ruby on Rails versions, but also all of the existing gems and libraries that the Autograder uses. This managed to be easier than we thought and was completed almost immediately upon starting our project.

Non-Functional Requirements

General User Interface Improvements**:** During the initial meeting with our client, he highlighted many pitfalls to the Autograder 3.0 UI. One area with room to improve was the way in which a student's score is displayed within the problem page. As of Autograder 3.0, students had to access their Grades page and find each problem there. There was no way to see their current grade on the problems page without running tests manually. We were tasked with displaying the grade of the current assignment right above the workspace, as well as reformatting the Grades page to show each assignment under their proper Class and Problem set (APT). Other UI improvements included showing when there is unsaved code in the editor, removing unnecessary buttons, and creating an easy way for students to access their code versions without leaving the problem page.

(Stretch) Migrate Existing Data: Since all saved code is currently located within the file system of the flowers server, it still has to be migrated to the database if students want to access their saved solutions. Due to the fact that this has little to no impact on the actual functionality of the system, migrating existing data is a goal that will be attempted upon the installation of the new system on the production server

Technology and Framework

Ruby on Rails**:** In the time since its inception, the Autograder has been ported onto the web-application framework Ruby on Rails. Rails is defined by its Model-View-Controller (MVC) framework which provides pre-established settings related to database and web access. It encourages usage of common web practices and taking advantage of JavaScript, JSON, HTML, CSS, and several other mainstay industry standards. In order to maintain and update the application, we had to familiarize ourselves with not only the Ruby programming language, but all web-development related languages used in the project

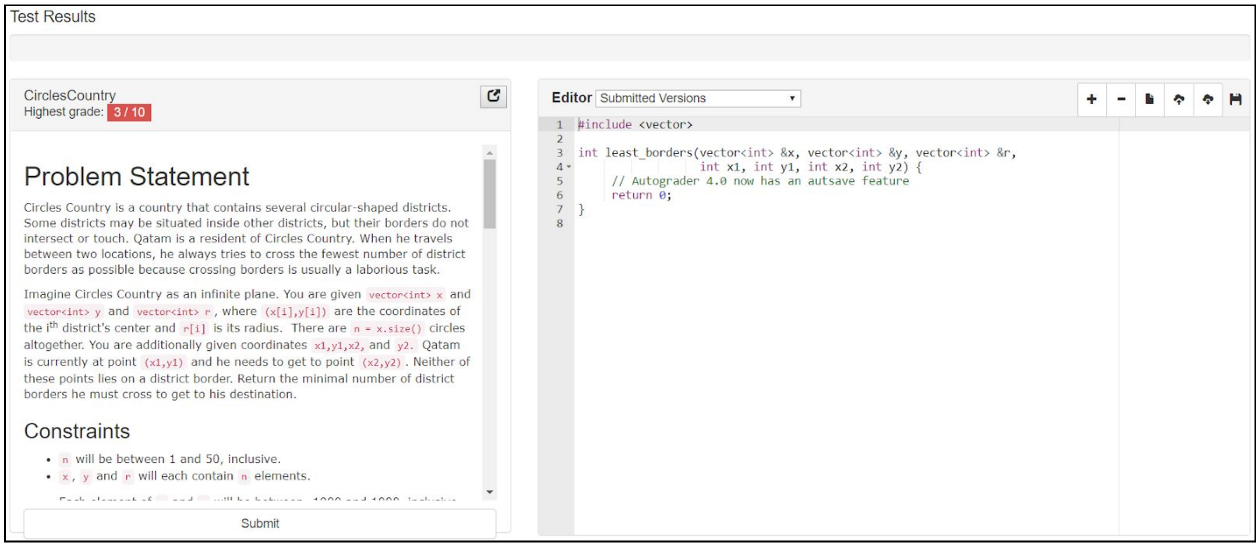
Linux: In order to work on the project, all team members needed to use Linux OS. Linux allowed full use of all Rails features as the command line was necessary for many operations. Windows or Mac systems may be able to edit and maintain the codebase, however setup is far more difficult

RubyMine IDE: Our client recommended a JetBrains product called RubyMine as our IDE. This modern editor allowed us to switch versions of Ruby easily, refactor code with ease, and run a localhost version of the web-application for testing.

System Architecture

Test, Save, and Submit Button

The first things students will notice about the Autograder 4.0 system is that there is only one button that tests, saves, and submits their code. Instead of forcing students to go through the arduous process of testing, saving, and submitting, students can now click one button and the back-end takes care of everything. There is a slight visual difference between version 3.0 (*Fig. 1*) and version 4.0 (*Fig. 2*), but the overall style of the application remains unchanged. In the editor header buttons, there is no longer a Save button. The Test button has been removed and there is now a single Submit button, which, as mentioned, combines the functionality of the old test, save, and submit buttons (*Fig. 3*).



Problem Statement

Circles Country is a country that contains several circular-shaped districts. Some districts may be situated inside other districts, but their borders do not intersect or touch. Qatam is a resident of Circles Country. When he travels between two locations, he always tries to cross the fewest number of district borders as possible because crossing borders is usually a laborious task.

Imagine Circles Country as an infinite plane. You are given `vector<int> x` and `vector<int> y` and `vector<int> r`, where `(x[i],y[i])` are the coordinates of the ith district's center and `r[i]` is its radius. There are `n = x.size()` circles altogether. You are additionally given coordinates `x1,y1,x2,` and `y2.` Qatam is currently at point `(x1,y1)` and he needs to get to point `(x2,y2)`. Neither of these points lies on a district border. Return the minimal number of district borders he must cross to get to his destination.

Constraints

- `n` will be between 1 and 50, inclusive.
- `x`, `y` and `r` will each contain `n` elements.

```
#include <vector>

int least_borders(vector<int> &x, vector<int> &y, vector<int> &r,
                  int x1, int y1, int x2, int y2) {
    // Autograder 4.0 now has an autsave feature
    return 0;
}
```

Figure 2. Autograder 4.0 User Interface

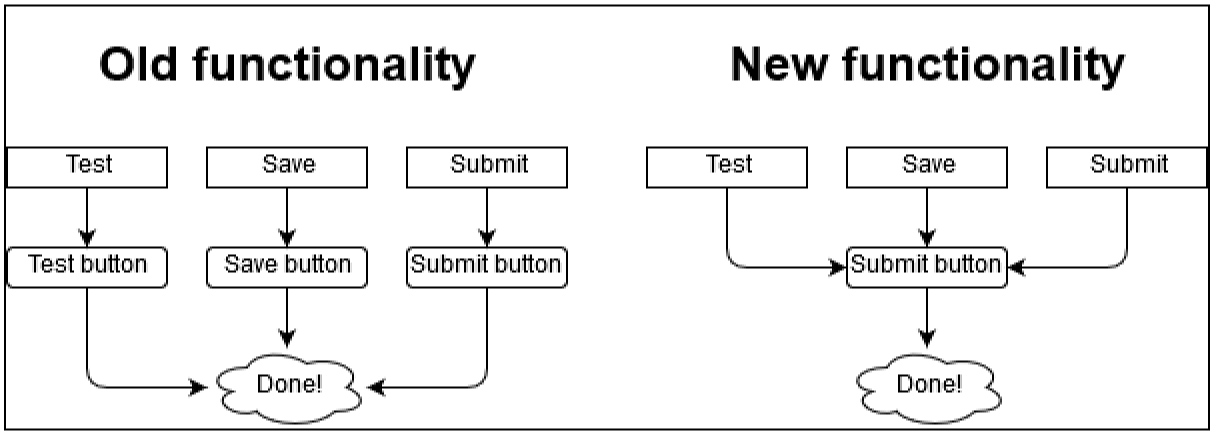


Figure 3. Comparing the old system to the new system

Version Control with the Database

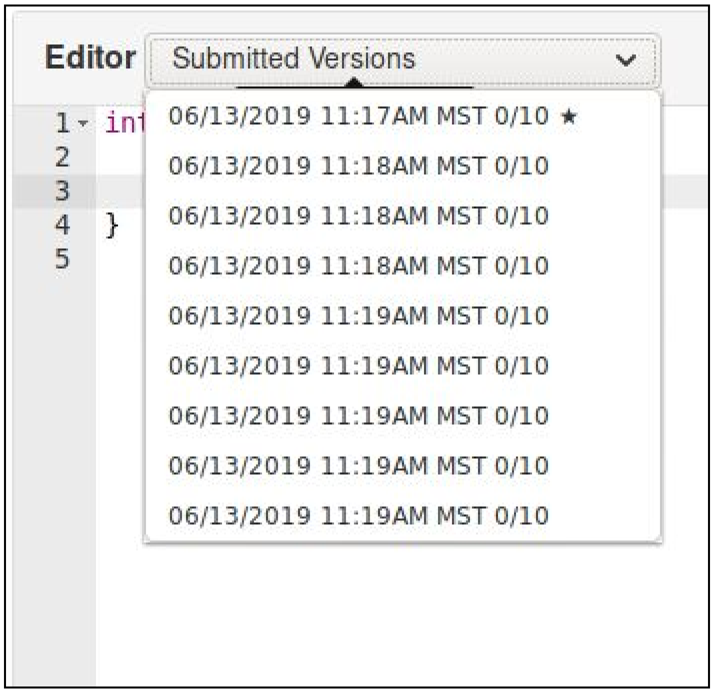
While the previous iteration of the Autograder did not allow instructors to see the actual submitted code, version 4.0 now makes this possible. Version 3.0 stored the student's code in a file system and wrote over the contents of that file every time the student saved. Now the code is archived in a Submissions table within the existing database. Considering this database was already being used for part of the project, all data storage has now been moved over to the database along with submissions. This allows archival of all code submitted within the application as well as increased efficiency without the need for file I/O on every save. Due to the increased access speed, the saving process is also able to eliminate duplicate code by checking the editor code against previously saved versions before it is committed using a single line of Ruby on Rails. Because hundreds of students are likely to use this at the same time, especially near exams and due dates, we came to the conclusion that it would be better to move everything over to this system instead of using the existing code that uses file I/O.

Which Version Gets Submitted?

One important part of having multiple submissions for any given problem is choosing the right version to submit. Students should obviously have their highest grade recorded in the gradebook, but what should happen if they have multiple submissions of an equally high value? The question now becomes which of these higher values should be recorded. With regards to our system, the logical choice decided upon by Dr. Painter-Wakefield and ourselves was to record the earliest, highest grade that is possible. Take for instance a problem with a due-date of Monday night. If a student receives a 10/10 on that assignment on Monday, their following 10/10 submission on Tuesday should not replace the existing grade and be counted late. This system not only ensures that the student gets the highest grade possible, but even allows students to find new solutions after the due date without harming their grade.

Accessing Previous Submissions

In terms of functionality, this is by far the most profound change the Autograder 4.0 update includes. In the open space near the editor header buttons, a version control drop down menu has been added. A timestamp and a test score are stored for each version and a star indicates which version of the code has been submitted for grading. The highest grade will be kept automatically and any later submissions that do not exceed the highest grade will be ignored with regards to the gradebook. The drop down menu (*Fig 4*) displays all submissions in chronological order to the user and they simply click on a submission to load the saved code. This will automatically replace the editor contents with that of the selected version.



```
int
}
```

06/13/2019 11:17AM MST 0/10 ★
06/13/2019 11:18AM MST 0/10
06/13/2019 11:18AM MST 0/10
06/13/2019 11:18AM MST 0/10
06/13/2019 11:19AM MST 0/10
06/13/2019 11:19AM MST 0/10
06/13/2019 11:19AM MST 0/10
06/13/2019 11:19AM MST 0/10
06/13/2019 11:19AM MST 0/10

Figure 4. Drop-Down Box for Version Selection

Autosaving Students' Code

At the start of the project, we wanted to address the issue of losing student code by implementing a save-on-close feature. We ran into a problem in that internet browsers intentionally limit the power that web developers have over preventing users from exiting a site, so it was difficult to guarantee the code was going to be saved. After extensive frustration and time commitment, we realized that the best method was to use a timed autosave feature instead. Autograder 4.0 automatically detects changes in the text editor, and sets a five second timer until it saves the code. Every time another change is detected, the timer restarts. In theory, this always prevents the loss of student code-unless they solved the entire problem and closed the browser in less than 5 seconds. As this is unlikely, the autosave feature effectively saves all code the student enters into the text editor. Icons were pulled from the Glyphicon library to represent 'Unsaved' and 'Saved' states of the user's code (*Fig. 5)*.

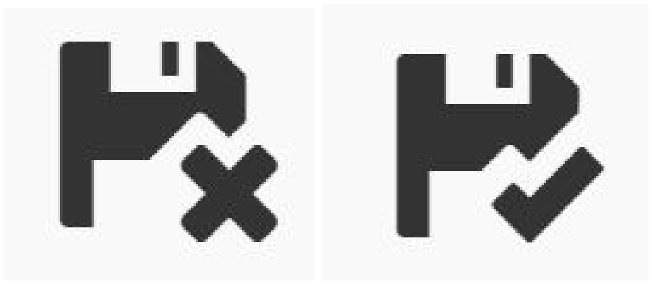


Figure 5. Autosave State Icons

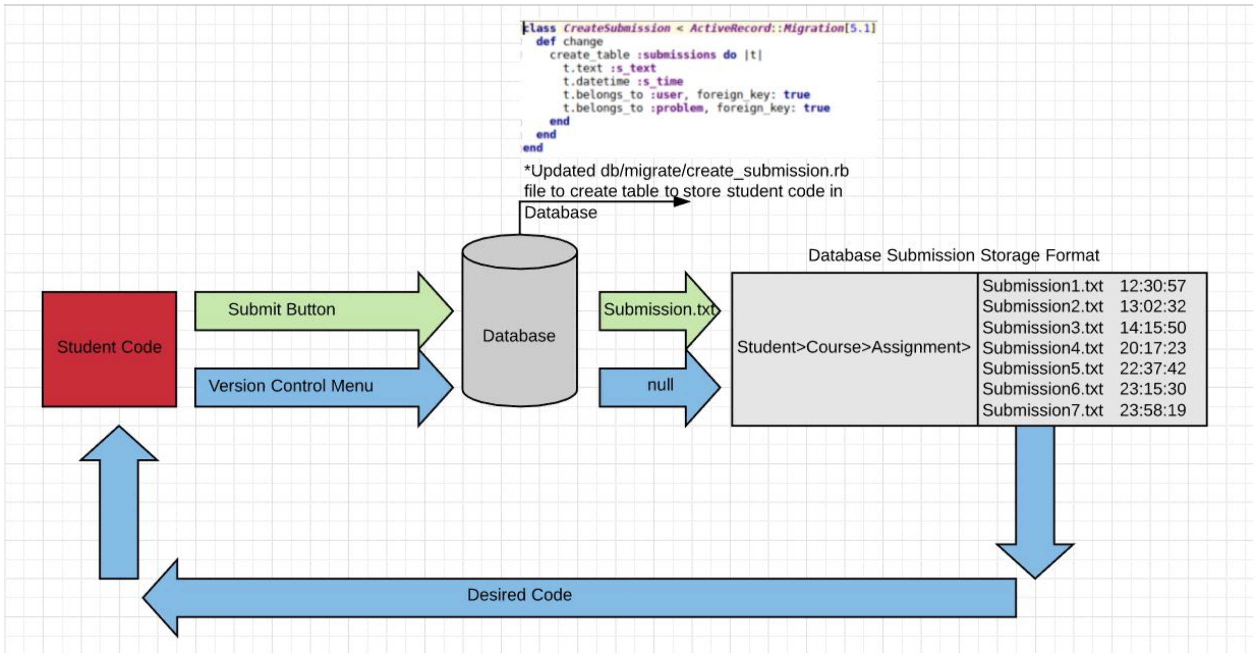
The Glyphicon library was used in the design of the updated user interface. It contains over 250 different glyphs, or icons, to convey a slew of different messages to the user. The Glyphicons don’t do anything aside from display. As the Autograder already uses certain Glyphicon components, icons were chosen from this library to represent the new features without disrupting users’ familiarity with the site.

Overall, the autosave feature is one of the most subtle yet powerful upgrades to the application. Students no longer have to manually save their code and risk losing everything. This new system works quietly in the background to ensure code safety and prevent stress. While it is not as prominent as the version control or major UI tweaks, it is just as important to the end-product as everything else within this update.

Technical Design

Data Flow through the System

Now that every user submission is being stored within the database as opposed to a file system, the database is being utilized much more than ever before. Not only is it receiving a myriad of submissions, but it is also being queried approximately every 5 seconds per user. We added two tables to handle this new data, a Submissions table and an Autosave States table. Each one would handle the respective data for each problem and user. An Object-relational Mapping plugin called ActiveRecord was used to communicate with the database through the Rails application. Commands were entered within the command line and auto-generated ruby files were created (*Fig. 6*). These files are run as a database migration, altering the schema and making additions or changes to the database.



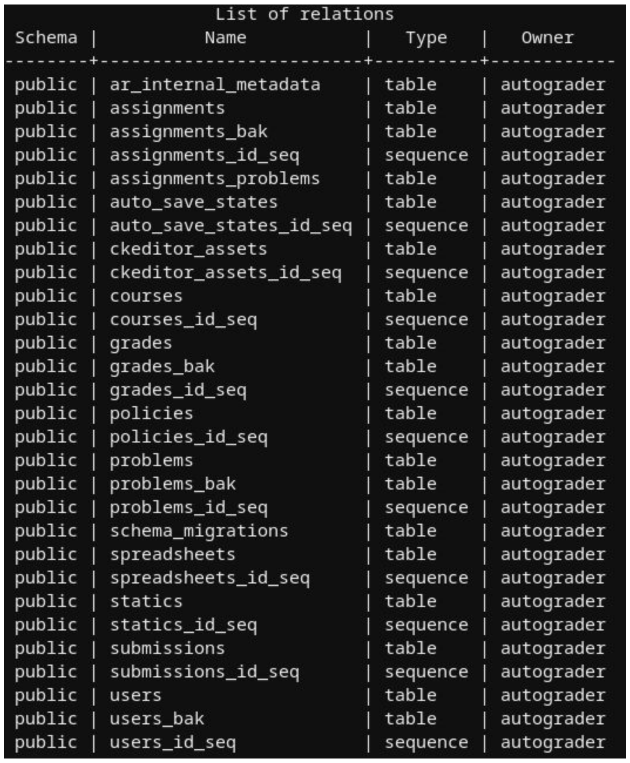
```ruby
class CreateSubmission < ActiveRecord::Migration[5.1]
  def change
    create_table :submissions do |t|
      t.text :s_text
      t.datetime :s_time
      t.belongs_to :user, foreign_key: true
      t.belongs_to :problem, foreign_key: true
    end
  end
end
```

Figure 6. Database Flow Diagram

When this feature was in its infancy, we were concerned that network traffic would be an issue. Some professors use the site to administer exams or quizzes. With the possibility of up to ~400 students using the Autograder at the same time, we were concerned that so many users querying the database would affect performance. However, we have been assured by our client that the Flowers server is capable of handling this load. At this time it appears that the site is running faster with the further implementation of the database. With that being said, however, this scenario is difficult to simulate and better performance cannot be confirmed/denied until more rigorous testing is performed upon implementation in the Fall.

Benefits of Database vs. File System

File systems and databases each have their own advantages and disadvantages. File systems are better when you have a small amount of arbitrary, possibly unrelated data stored in separate files. This is because they are less complex, and offer more redundancy than databases. Databases, on the other hand, are more efficient when you have large amounts of data that are related to each other in some manner. This is because they provide data consistency via a normalized template for each datum. They are more flexible than file systems when it comes to accessing data, since they use the Structured Query Language, or SQL. In our case, since our client wanted to store multiple submissions - which would be related to each other in that they share a common student and problem - we elected to build off of the existing database, which had everything needed except for the Autosave State and Submission tables (*Fig 7)*. Since file I/O blocks CPU operations while it is running, we decided that it is safer to use a database, especially given that it runs on a single server. On top of this, since Autograder runs on a Linux server there is something called an inode limit - that is, a limit to the number of files one can store on the server, regardless of remaining space in the file system.



```
                  List of relations
 Schema |          Name           |   Type   |   Owner
--------+-------------------------+----------+------------
 public | ar_internal_metadata    | table    | autograder
 public | assignments             | table    | autograder
 public | assignments_bak         | table    | autograder
 public | assignments_id_seq      | sequence | autograder
 public | assignments_problems    | table    | autograder
 public | auto_save_states        | table    | autograder
 public | auto_save_states_id_seq | sequence | autograder
 public | ckeditor_assets         | table    | autograder
 public | ckeditor_assets_id_seq  | sequence | autograder
 public | courses                 | table    | autograder
 public | courses_id_seq          | sequence | autograder
 public | grades                  | table    | autograder
 public | grades_bak              | table    | autograder
 public | grades_id_seq           | sequence | autograder
 public | policies                | table    | autograder
 public | policies_id_seq         | sequence | autograder
 public | problems                | table    | autograder
 public | problems_bak            | table    | autograder
 public | problems_id_seq         | sequence | autograder
 public | schema_migrations       | table    | autograder
 public | spreadsheets            | table    | autograder
 public | spreadsheets_id_seq     | sequence | autograder
 public | statics                 | table    | autograder
 public | statics_id_seq          | sequence | autograder
 public | submissions             | table    | autograder
 public | submissions_id_seq      | sequence | autograder
 public | users                   | table    | autograder
 public | users_bak               | table    | autograder
 public | users_id_seq            | sequence | autograder
```

Figure 7. Database Schema

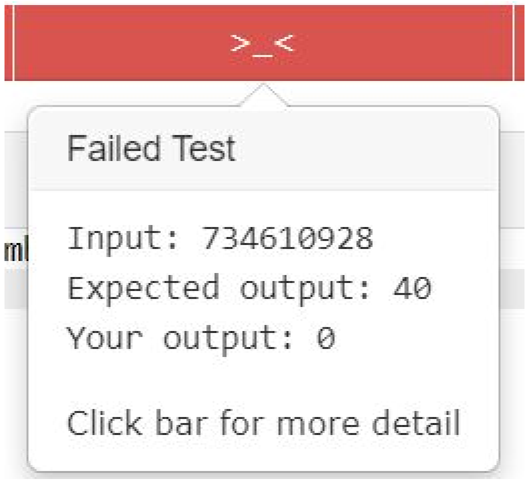
Quality Assurance Plan

There are countless moving parts in the Autograder system, and if one of them is out of place, the entire system can be rendered useless. Due to the fact that student grades are dependent on the functionality of the system, we have an ethical responsibility to ensure this software is not only functional but of the highest quality possible. While automated testing would be nice for our product, there was not enough time to adequately do so. For our purposes, we stuck to thorough, manual testing.

Deployment Testing

Before deployment, Dr. Painter-Wakefield regularly tests his builds and updates on a test server accessed through mauchly.mines.edu. For all intents and purposes, it is a ‘mirror image’ of the production server. We deployed to Mauchly near the end of week 4, leaving roughly two weeks to search for any possible errors that students/professors may encounter while using the new system.

The first Mauchly deployment presented slight errors concerning database migrations, but a reset of the database schema as well as another round of migrations fixed the issue. At this time the UI had not yet been finished and the version control system was not working fully. Based on feedback and our own test results, we adjusted the header button UI and implemented version control fully for the week 5 deployment. Upon deployment, we discovered that testing output was not being reported to the user in the usual fashion (*Fig. 8*). Instead of outputting the users results next to “Your output”, nothing was displayed. After consultation with Autograder Group 2, this issue was resolved and Mauchly was updated. All other bugs stemmed from unfinished UI and are either fixed or in the process of being fixed right now.



```
Input: 734610928
Expected output: 40
Your output: 0
```

Figure 8. Test Result Display

Code Reviews

Though we didn’t dedicate massive amounts of time to actually setting up code reviews, we’ve had plenty of opportunities to be critical of one another. Occasionally, we disagreed on what was the best way to approach a problem. Though it was difficult, this process pushed us to compete with one another and create the best possible solutions. Since there are so many interdependent parts of the Autograder system, a seemingly harmless change can wreak havoc elsewhere in the system. This naturally caused us to be in a constant state of reviewing and (constructively) criticizing one another’s work. As a result, the code is generally clean, well commented, and easy to follow once you have a grasp on how the system works.

Merge Meetings

Every week or two, we met with the other Autograder team in order to merge our code. This repeated process ensured that we always worked on the same versions of the code and making incremental changes. Luckily, these merge meetings went generally well, with very minimal merge conflicts. Thankfully, the previous field session groups did a good job of compartmentalizing the system such that the student-facing team and professor-facing team are rarely working on the same files. As we neared the end of the six-week session, the frequency of the merge meetings increased to ensure there were no last minute hangups.

Client Meetings/Demonstrations

As per the Agile process, we met with our client to discuss progress on the project a minimum of once a week. These weekly meetings gave us a perfect opportunity to demonstrate how update implementation was going. We ran into many small issues in this project, some of which forced us to change directions slightly. As mentioned earlier, the seamless switch from the save-on-close to the autosave feature went so due to the fact that we were in such close communication with our client. Due to the open line of communication, we were able to quickly resolve issues that would typically have been a hangup on a project like this. These meetings have also allowed us to tweak the UI the way the client wants. These changes are simple, but have a big effect on user experience; so it was important to get it right.

Results

Overall, we managed to meet the primary objectives as stated by our client. The user experience has been revamped and seems to be more intuitive. Throughout the project we adhered to over-arching stylistic components that were already in place. Every addition to the student view was deliberate and designed to enhance the user experience without the user having to put much thought into it.

As previously stated, our main goal was to refine the already working Autograder 3.0 system and streamline functionality on the student-facing side of the application. Combining the Test, Save, and Submit functions into one button as well as implementing an autosave feature has greatly increased the site reliability on the student-end. By adding the ability to revert back to previously submitted code versions, students are now more free to take chances and solve problems in ways they might normally not attempt. The version control system we developed allows students easy access to their code versions without having to worry about the back-end at all.

With regards to new features to prevent code loss, we were forced to take a different path than the one we originally planned. At first, we thought that the best way to go about this was to implement a save-on-close feature, but it quickly proved to be difficult and unreliable. This was due to the fact that internet browsers intentionally limit the power web developers have over web page visitors, making a save-on-close feature much more difficult than originally thought. Instead, we implemented the timed autosave feature that has turned out to be better than a save-on-close ever could have been. At the start of week 5, we deployed to the Mauchly test server. By leaving ample time for testing and debugging, we have ensured that we found any unseen issues in the Autograder 4.0 system. The autosave feature is one of the most profound changes in the Autograder 4.0 and fixed the most frustrating issue with Autograder 3.0.

There are still many potential features that would still be added to the Autograder system (examples being custom student tests, improved Python support, improved error display, etcetera). The only reason we did not get around to adding these features was the time limitation. Our group and previous ones did a good job of following the OCP, and therefore future Field Session groups should be able to improve this system with relative ease.

Conclusion

Lessons Learned

As a group, we learned a lot while working on this project both technically and personally. In regards to the technical skills, our team had next to no web development and database experience at the beginning of this project. Through intensive internet research and the help of our client, we have learned at least basic skills in both categories. While this might have been out of necessity for completing the project, we still came out better-off at the end. In no particular order, the project took advantage of Ruby on Rails, HTML, SQL, JavaScript, and CSS. There was a steep learning curve for every group member, however we now have experience working with these industry standard languages and frameworks.

In regards to personal development, we found the old adage "If it ain't broke, don't fix it!" to be quite applicable. The Autograder was given to us as a fully developed, working product. Throughout our time working with it, we had to take caution not to damage any of the working parts that already existed. At times this was easy as we never had to touch the user authentication process or the way in which Autograder handles problem data. Other times, we had to directly modify existing code for our purposes. In the case of the database, we made several considerable additions while taking care to 'link' our new tables to the existing ones in the 'correct' way. When working with any existing code base, it is always a nerve-wracking task to modify existing code, however it was necessary for the completion of our project. Overall, however, we found many things we could leave alone throughout the project, and we took that chance at every opportunity.

Something we learned early was that it is always a good idea to leave time for possible errors. Procrastination is an unfortunate habit that we all fall into, but for this project we could not have delivered a great product without diligence. Everything we added to the Autograder required multiple revisions before it 'worked'. Without ample time to debug and refine, we would have delivered a buggy, inferior product at the end. Instead, we gave ourselves plenty of time to find and fix as much as we could to end up with a product we are all proud of. As a result, we believe we have a very informative and user-friendly UI and a system that takes multiple steps to prevent loss of student code.

Probably the biggest lesson we learned had to do with communication. For the first time, two groups were assigned to work on the CSM Autograder. We knew from the beginning that we would need to collaborate with this team throughout the project, however we did not immediately do so. Both teams worked separately from each other for roughly 3-4 weeks before the Mauchly deployment, and when the server was deployed we realized there were issues caused by 2 teams working apart from each other. By the last week and a half of development, we worked daily with Autograder Group 2 to find bugs, fix them, and modify the product to meet our clients needs. If we had tried to do this sooner, we might have prevented a few headaches. In the end though, we realized how important constant communication was and we finished strong with the other group.

Reflection and Future Work

At this point, we have fulfilled the major tasks our client asked us to complete and we can all say that we are proud of the work that we have done. Throughout the 6 week process, we have incorporated partner-programming, Agile and SCRUM techniques, and bettered ourselves as programmers and team-players. Overall, we treated this course and project as a learning experience and we have been rewarded.

The Autograder is in a state that is 95% complete as it still needs to be deployed onto the production server, flowers.mines.edu. This will be handled by Dr. Painter-Wakefield in the coming weeks or months before the start of the Fall 2019 semester. Representatives from both teams may be asked to assist in the deployment and testing of the application upon Dr. Painter-Wakefield's request. Right now, testing indicates that the system is in a good place technically with only minor dependencies requiring updates on school servers.

One last issue that has yet to be resolved is whether existing save files stored on the flowers server can be migrated to the new database format. Our team did not have enough time to create a solution to this problem, however one may be developed in the near future by students or Dr. Painter-Wakefield himself.

Appendix

Figure 4 Source

https://www.glyphicons.com/

Development Environment Setup Instructions for Future Developers

https://docs.google.com/document/d/1XwFGZvVWsTtzSq2EwzrSezqX0nKwzG8peHX29_Q8jQo/edit?usp=sharing