MQP GFP 1201

Git Analytics Tool

A Major Qualifying Project Report:

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

degree of Bachelor of Science

by

_____

Justin Chines

_____

Andrew Iovanna

_____

Thanaporn Patikorn

_____

Anjali Venkatesh

Date: March 19, 2015

Approved:_____

Professor Gary F. Pollice, Major Advisor

Abstract

This MQP was created for professors in the computer science department – specifically those teaching the software engineering course. Students and teams are evaluated on how they distribute work and function as a unit. Grading for this course can be long and potentially inaccurate as it requires investigating the repository history of each team. Our project identifies work habit and distribution patterns from Git commits, giving the professor more time to administer the course and provide feedback to students.

ACKNOWLEDGEMENTS

We would like to thank our advisor, Professor Gary F. Pollice, for guiding us through our project and providing us with his knowledge, expertise, and guidance at every step along the way.

Additionally, we would like to thank our friend Gabriel Morell-Pacheco for giving us design advice and allowing us to tap into his significant Django expertise during the initial stages of the project.

Finally we would like to thank all our friends and classmates who volunteered to participate in our usability study and providing us with valuable feedback and suggestions for improvement. Without all their support, this project would not have been possible.

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

The goal of this MQP is to simplify student evaluation and feedback in WPI computer science courses, specifically software engineering. Professors can gain valuable insight on individual contributions and team performance by looking at data stored within version control software (VCS), improving the quality of feedback and saving time for faculty.

Software engineering is a unique computer science course at WPI; instead of working individually or in small groups, students work in teams of 15 to create a module for WPI Suite. WPI Suite is an open-source software package created by students throughout multiple offerings of the software engineering course. While the goal of other courses is to solve problems in a specific technical realm (operating systems, networking, object-oriented design), software engineering's goal is to teach the "soft skills" required to work in the software development industry: teamwork, critical thinking, communication, etc. The course also teaches a history of software development patterns and trends, as well as an overview of those in use today.

As a course oriented towards providing practice with industry standards, students learn to use professional software tools. Among these is version control software, which is used by groups of software developers to coordinate and combine code contributions.

Version control software like Git and Subversion (SVN) are popular among software developers. These tools store current and previous versions of all files, and simplify the process of developing software in groups. Courses with large group projects, such as the software engineering course offered at WPI, benefit from VCS to combine code contributions from student teams. The course instructor is able to access teams' Git repositories and inspect their code.

Evaluating individual and group learning outcomes for software engineering is more complex than for other classes, as they are not fully observable. A combination of teaching assistant input, individual reflection, and peer-grading influence evaluation. Code contributions to a team's Git repository are also considered. Our major qualifying project (MQP) automates this analysis by analyzing team repositories and reporting on the contribution habits of each team member.

Git provides information on contributions that helps assess individual students and teams. Tools exist that can visualize this data, but are geared more towards managers and code

quality inspectors. Such tools provide useful data for student evaluation, but took significant set-up time, and presented so much information that it was not intuitive to find only what was useful for our project. Using them in software engineering would likely add time and complication to the professor's job.

This MQP provides multiple features for the professor in a quick and easy-to-use website. Our application shows basic information on student commit habits within a team. Beyond this, the professor can define custom metrics to identify trends, such as students who commit on a certain day, students who delete code instead of writing new code, etc. Our application can raise red flags for the professor by identifying students who do not contribute, or who display bad commit habits. It can also identify students who should be recognized for contributing regularly or significantly. Strong contributors can be rewarded, weak contributors can be encouraged, and students with negative habits can be taught how to improve. Software engineering emphasizes distribution of work — our tool can make it clear how well teams are achieving this and other course outcomes.

Our team worked closely with a professor of this course at WPI to identify what information is useful to meet these ends. The end product is an intuitive Web application that can quickly visualize and encapsulate whatever data the professor is looking for. With the adoption of our MQP, students will learn more from the software engineering course, and the professor will have more time to spend interacting directly with students. Our project is accessible at www.sourcecontrol.me.

This paper provides an overview of version control systems, a discussion of the work we put into creating our application, and a review of our accomplishments.

2 BACKGROUND

2.1 SOFTWARE ENGINEERING

Our MQP aims to improve student evaluation and ease workload on the professor for WPI's software engineering course. The course places students in teams of up to 15 students each, challenging them to create a functioning software module in the span of a 7 week term. Development is done in Java, a popular object-oriented language. Software engineering introduces students to software development methodologies, for instance, Scrum and Kanban. The goal is for students to improve their communication and problem-solving skills through the challenge of a team setting with demanding goals. Students also gain practical skills (familiarity with design patterns, presenting, writing) that give them an advantage in the workplace.

Students are graded not just on their code contributions, but also their interactions with teammates, leadership, and contribution to the team. This presents one of the most complicated grading schemes of any computer science course at WPI. The professor has to evaluate students and teams on seemingly intangible goals: adherence to development practices, distribution of work, etc.

The professor currently uses a mix of student assistant input, reading through thousands of journal entries, presentations, student paycheck distribution, and code reading to evaluate student and team performance. Student assistants, referred to as 'coaches' for this course, work closely with their assigned teams, helping them through the difficulties of the course and providing feedback. Every day, students are requested to write a journal entry, detailing what they contributed to the team and what they learned.

Other courses in WPI's computer science department require students to work in groups, but the emphasis of these courses is on technical skills and not on inter-personal skills developed by working in a team. Courses like Operating Systems, Networks, Object Oriented Analysis and Design, and even introductory courses can all benefit from the use of our MQP to tease out what students are working hard versus those that might be relying too heavily on other group members. One barrier to this is that students might not be using Git on their own, and might not react well to having it added as a requirement for the course.

2.2 Source Control Overview

Version Control Systems are an integral part of software development for professionals, hobbyists, and students.[1] Git, SVN, and other tools help track changes to the codebase, creating a full history of the project's development. This creates a change list that can be used to view the history of a given file or to rollback to previous versions if the need arises. Each change or "commit" is associated with an author, who can add a message explaining his or her commit. This makes collaborating on large, complex software projects easy, as it becomes known who is responsible for contributing which parts of the project, and the commit message can often clarify basic information needed for continuing development.

VCS supports branching, which allows for development of an individual feature to be done on a copy of the codebase separate from the main codebase. This largely eliminates the risk of impeding the work of other contributors while creating a new feature or fixing a complicated bug, as it can be done in isolation from other development work. When coding is completed, the "feature" branch can be merged into the main branch to incorporate changes that were made. Such merges would be complicated, time-consuming, and error-prone to do by hand. VCS software handles this merging for the user.

Git was created by the developers of the Linux kernel as a tool to aid development. Other examples of version control software are Mercurial and Perforce. Git and Subversion are by far the most popular, used by 33% and 31% of developers respectively.[2] The software engineering course at WPI that is the primary customer for our project uses Git for version control.

2.3 Data Visualization

Data visualization was essential to our product since the large amount of raw data gathered by our program would be difficult to interpret on its own. Graphs, charts, and other data visualizations played a key role in representing data in a quick and easy-to-understand manner, as well as in helping people detect contribution levels and trends.

The "lie factor" is a metric that describes the relation between the size of the effect of change shown in the data versus the size of the same effect shown in a graph[3]. The further away the lie factor is from 1, the more a graph is exaggerating our underplaying variation in data, though this often leads to more eye-catching visuals and is frequently used in order to make a

point. We made a conscious decision to not underplay or exaggerate any changes in data. We made the decision to follow other data visualization principles such as keeping Edward Tufte's data-ink ratio (the proportion of pixels that are used to present actual data compared to the total number of pixels used in the entire display) in line and bar charts to a reasonable minimum, and eliminating all unnecessary visual elements[4].

2.4 EXISTING TOOLS

Several tools exist that visualize Git repository data. While our project focuses on data for professors' use, our team benefited from using and inspecting similar tools that exist for developers and managers.

GitInspector (Figure 1) is a tool created by students from two Swedish universities to provide analysis of Git repositories used for student projects. It is a small Python project that visualizes much of the same data that we do; including commits, insertions, and deletions per author, and contribution history over time. GitInspector provides a "sort by" feature on authors that makes it quick and easy to identify strong contributors by number of commits, insertions, deletions, and other data. It can also sort through a project to provide data on certain types of files (.py, .c, etc.), which could prove useful in a product as large and diverse as WPI Suite, the project that is worked on in software engineering.

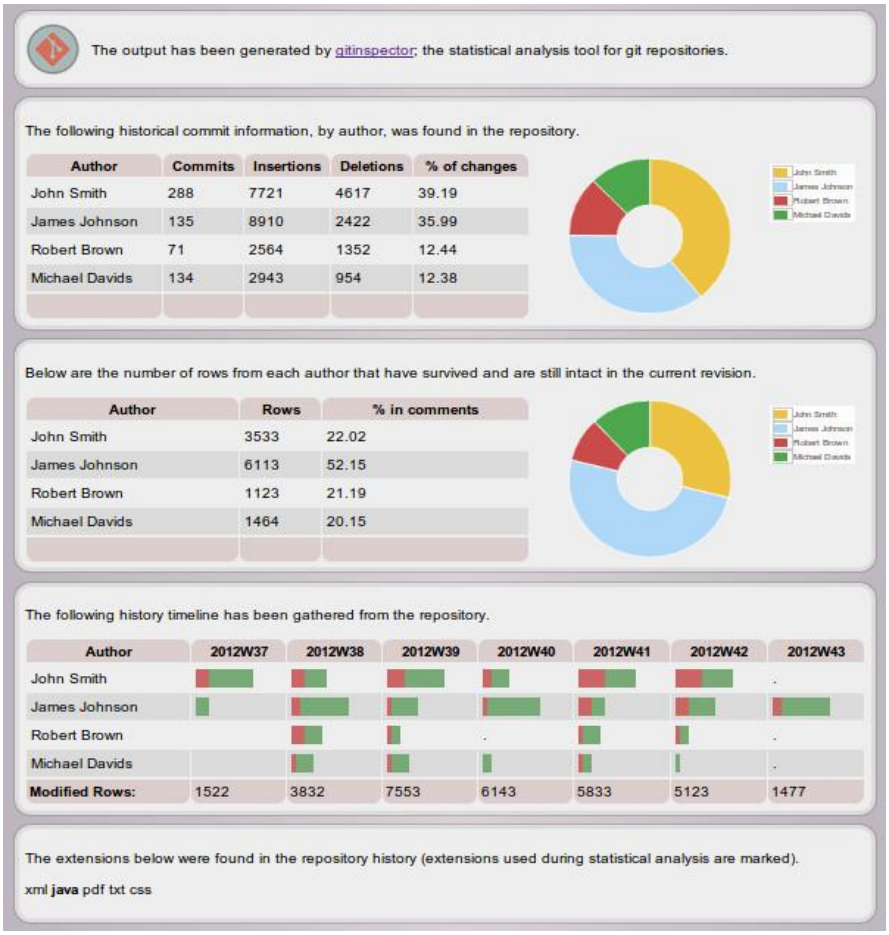

Figure 1: GitInspector Output

Another tool similar to ours is SonarQube (Figure 2), which can be used as a web app, desktop app, or Eclipse plugin to analyze code quality and improvement over time. The data presented is of interest to managers and developers.

Figure 2: SonarQube Data Display

SonarQube suffers from information overload, presenting a lot of information on the initial screen, not all of which are relevant to our goals. The landing page shows general info: lines of code, files, functions, tests, etc. Getting more detailed information is possible by simply clicking on the title of the section of interest. Other features include: interactive graphs, support for over 20 languages, code quality, bugs, classes (for object-oriented languages), duplication, documentation, and complexity.

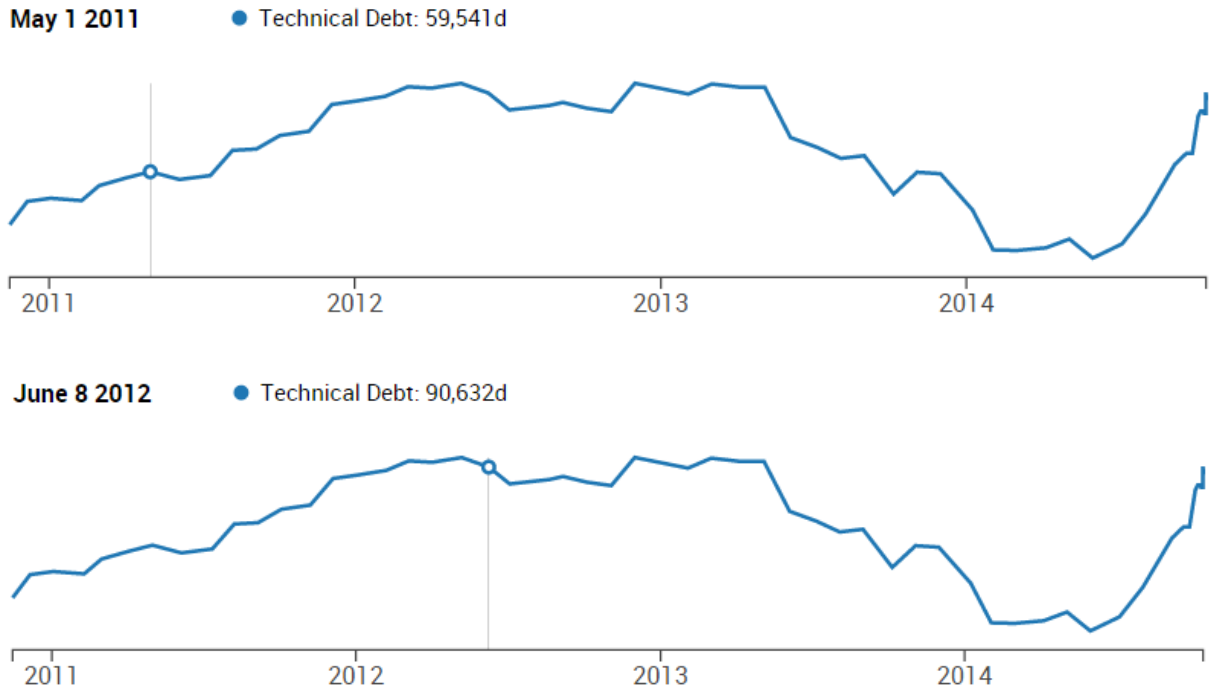

Figure 3: SonarQube Interactive Line Graph

2.5 WEB DEVELOPMENT

The Web has become one of the most ubiquitous computing platforms for communication and activities. This increased use has gone hand-in-hand with the development of HTML, CSS, and JavaScript to further develop the Web as a platform for commonly used applications. The popularity and accessibility of the Web, coupled with the technologies used to develop for it, make it the best choice for our application.

There are many frameworks that can prevent developers from reinventing the wheel, save development time, and provide features such as secure user login, asynchronous job queues and session management. Our application uses Django, a Python-based Web framework, and

Foundation, a CSS library with strong grid-support. Django is described as an “MTV” framework by its creators[5] – consisting of “models”, “templates”, which are what the user actually sees, and “views” describing what data the user will see. Using these frameworks allows our team members to spend more time implementing features unique to our application.

3 METHODOLOGY

3.1 REQUIREMENTS AND TOOLS

We began the project by identifying the requirements of a finished product. This process included discussions with the professor, use of similar tools, and brainstorming as a team.

Using similar software made it clear that one of the main benefits our tool could provide is speed and ease-of-use. Some products required upwards of ten minutes set-up time, and all of them had an overwhelming number of features. These tools have clear user interfaces, which incorporate important raw data and graphs on the same page. The design of our information pages (Repositories, Users, Reports) looks similar to what we saw in these tools.

Our team performed a round of user story creation after input from the advisor. These user stories helped clarify broad desires of the professor ("I want to use this for all the software engineering teams") into specific goals for our product ("I want the product to remember each of my repositories"). This makes it easy for the professor to check our progress, and to provide feedback on how well implemented the stories are. While not all of our original stories are in the final product, those that are were clarified and improved through this process. The original list of stories can be found in Appendix B.

The team developed a list of specifications based on conversations with the professor, using similar tools, and through our own experiences in the course. This helped focus the product on specific objects (users, commits, time) that the product would focus on displaying. Professor input created the general goals of the product, while using similar software showed approaches to achieving those goals (graph types, website design, metrics). The result of this process is a list of specifications that guided the design of our project. As with the user stories, not all of the specifications were met, as the goals of the project changed over time. The original list can be find in Appendix C.

After talking with the professor, using similar software, and creating user stories and specifications, the team chose tools to use for the project. We decided to use Django to create an MTV backend to our project. Having the specification sheet and user stories ready at this step allowed our team to check that Django and other tools were sufficient for our goals. We also

used Foundation, a front-end library, for our styling. Foundation's simple design fit well with our goal of creating an easy-to-use product.

The most difficult tool decision was deciding which Javascript library to use for creating graphs. After experimenting with d3.js, we decided on Flot, which is a simple charting library. While it does not provide the charting power and flexibility of d3, it was easier to implement with the custom query language, which could produce a wide range of data for charting.

3.2 GOALS

After researching existing repository analysis tools and various implementation options, we began designing our product, a website we named SourceControl.me. We set goals to meet those requirements and made a series of implementation decisions that gave rise to the final product.

The goals for the project were as follows:

- Create a website to analyze Git repositories that is easier to use than other similar tools.
- Provide meaningful data visualizations alongside repository data.
- Create an extensible product that can easily be built-upon to add new features, such calculation of changes per class and similar language-specific features.
- Streamline the website to provide information relevant to the instructor and project managers for evaluating software engineering students.
- Allow the user to generate customizable reports based on some formalized queries.

3.3 HIGH LEVEL DESIGN

Project Structure

The SourceControl.me directory structure follows Django convention and its three sub-packages or "applications" (in Django terminology) deal with repository management, data visualizations and customizable reports. The first contains code responsible for adding, editing and removing repositories, and presenting data specific to the added repositories. The second is

responsible for setting up API endpoints for every chart we generate, while the third is responsible for the creation, deletion and editing of customizable reports associated with a repository, and the queries that are contained in each report.

The Front-End

Our product uses a responsive CSS framework called Foundation by Zurb, which uses a 12-column grid system. Ben Schneiderman's Eight Golden Rules of Interface Design[6] (Appendix F) have been kept in mind while designing and improving upon the interface. The dominant color scheme comprises of grays, blues and black. To the extent possible, colors such as red and green have been avoided in order to keep the product colorblind-friendly. We have aimed for consistency in terms of terminology, layouts and color schemes in the different views.

There are 11 unique views on our site; however, navigating between different views is made very easy by a universal drop-down menu on the top bar. It helps the user navigate between the home page and various repositories and reports. We also have quick links to the previous view on most views (Figure 4).

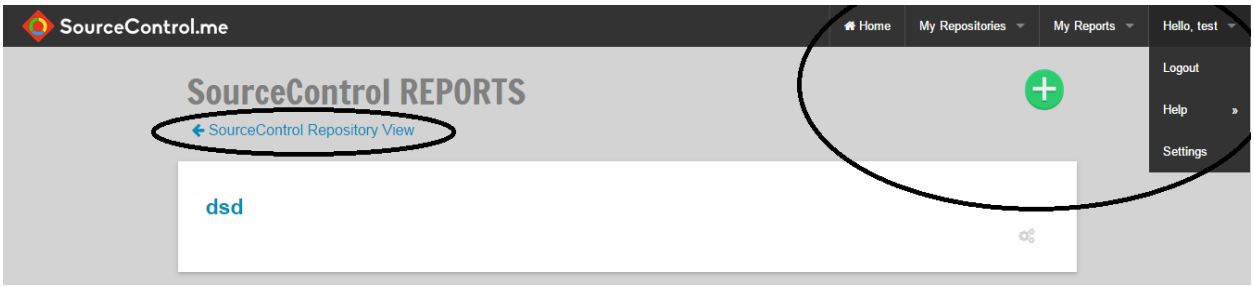

Figure 4: Navigation bar and quick link to previous view to ease navigation.

There are three categories of views, not including the landing page for a logged out user—list views for lists of repositories and reports, detail views for repositories, committers and reports, and form views for login, signup and adding reports and repositories. The information is grouped and presented in the form of cards in all these views. This is because cards make grouping of information easier, adapt well to every different screen sizes and impart a sleek and modern look to the interface (Figure 5).

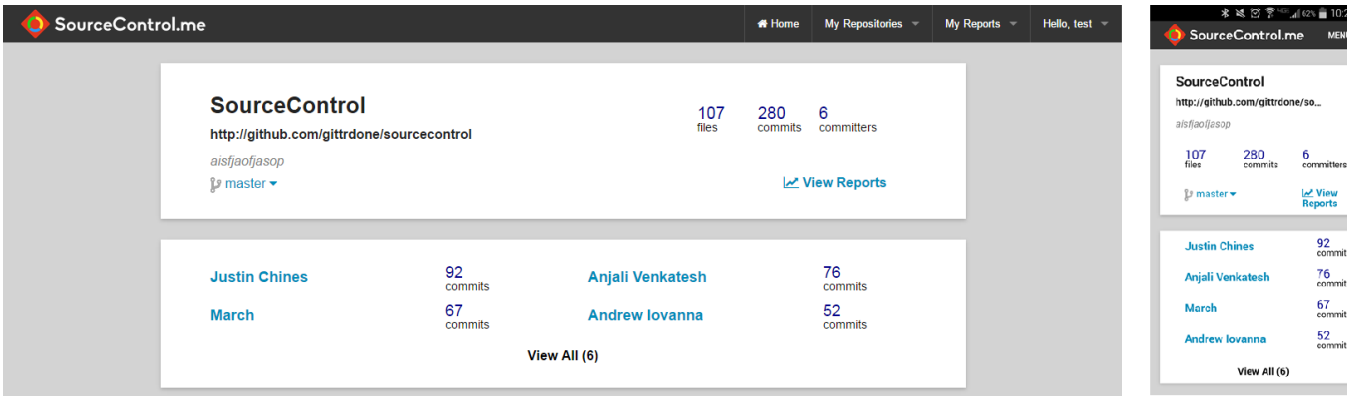

Figure 5: Cards displaying different groups of information and on different screen sizes (desktop and mobile)

Model Design

Our user model, SourceControlUser, extends Django's default User model, allowing each user to store an associated list of repositories. Django's provide User model gives all of the required basics: username, email, password, and comes with authentication functionality. Using the provided model saved a lot of time and simplicity in development.

Branches are stored independently of repositories in a many-to-many relationship. This is because one branch might be common between a forked and original repository. The branch will only be stored once, saving space on the server and time during repository cloning. This is especially common in the software engineering course, as the teams fork their repositories from the WPI-Suite Master repo.

Reports are stored independently of repositories for a similar reason, as the professor might make the same report across the repositories for each team. This also applies to the relationship between queries and reports, as some of the requested information is likely to be common between reports.

Different models are used in reference to user objects and Git objects. Users have UserGitStores which store information about what a user associates with a Git repository, such as a friendly name and description. In the backend, there is a GitStore that stores information about the actual Git repositories. Separating these concerns allows information about repositories to be easily shared among users, without private information leaking.

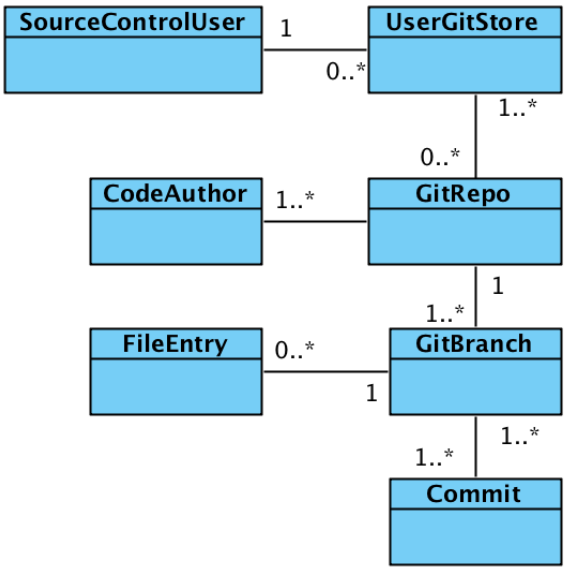

Figure 6: Model Design

Speed Consideration

Adding a repository can take considerable time depending on the number of commits in its history. Our website was originally hosted on Heroku[12], but performed poorly with large repositories. A repository with around 6000 commits takes about 8 and a half minutes to process entirely, on average. A repository with only 1 commit however will take a little less than a second. To minimize the time needed, we moved the application onto the same server as the database. Adding repositories takes much longer with increased latency between the application and the database during processing.

The benefit of processing repositories when we first clone them is that generating reports is very fast. Additionally, when a repository is updated, we only have to process the new changes that have been made since the last time it was checked. For example, that repository with approximately 6000 commits takes only about 4 seconds to update, on average.

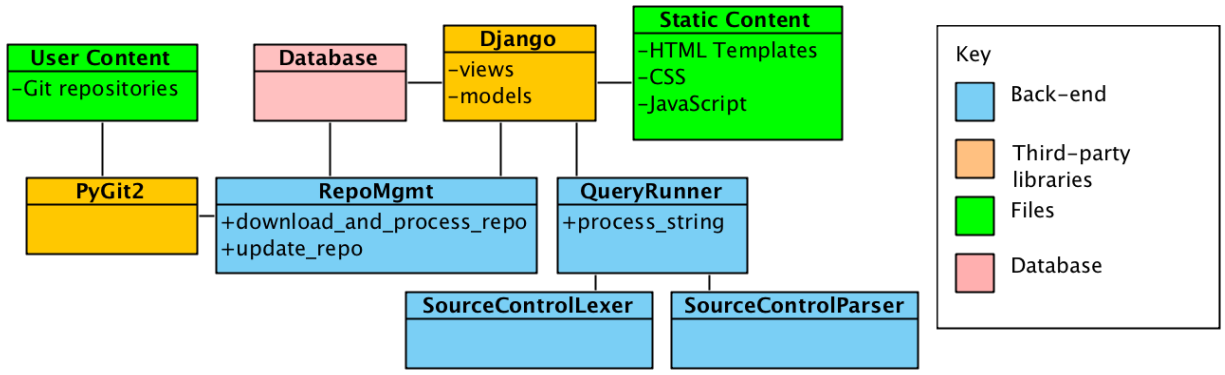

Figure 7: High-Level Design

3.4 IMPLEMENTATION

Login / Sign-up

Implementation began with creating landing page (Figure 8), login page (Figure 9), and sign-up page (Figure 10). We first made a base HTML file that specified elements and styling common to all pages (top navigation bar and central content pane), before adding specific pages that inherited from the base page. Views, written in Python, were added to serve specified pages and process data. For instance, the login view uses Django's built-in user authentication method. The user model is an extension of the one provided by Django.

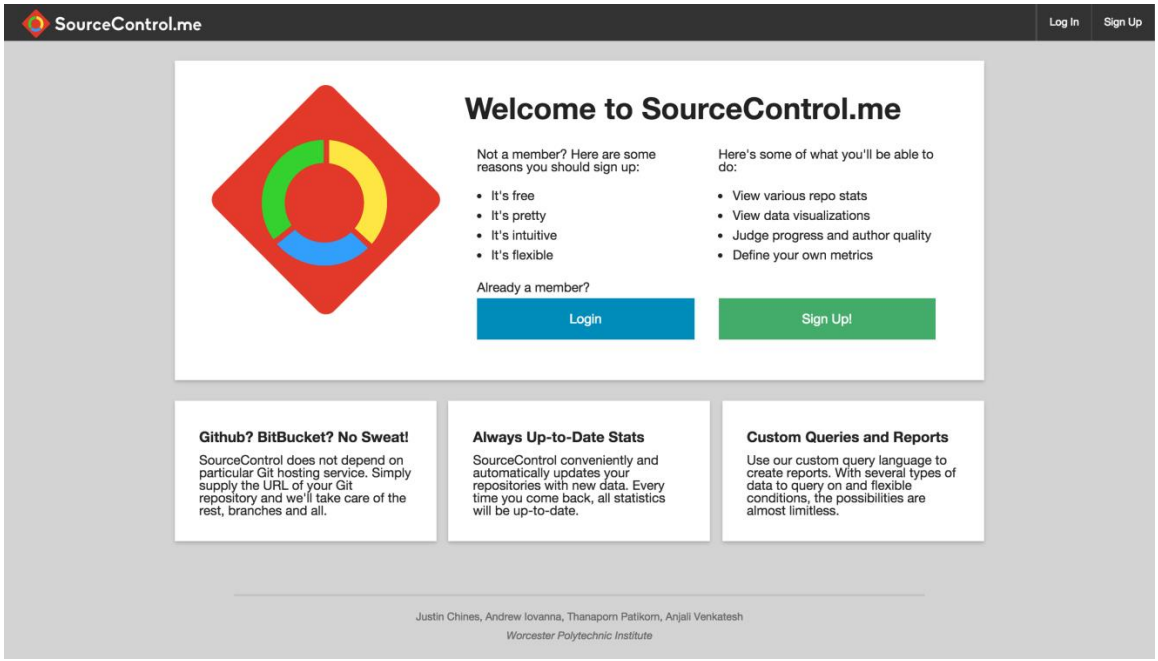

Figure 8: Landing Page (Logged Out)

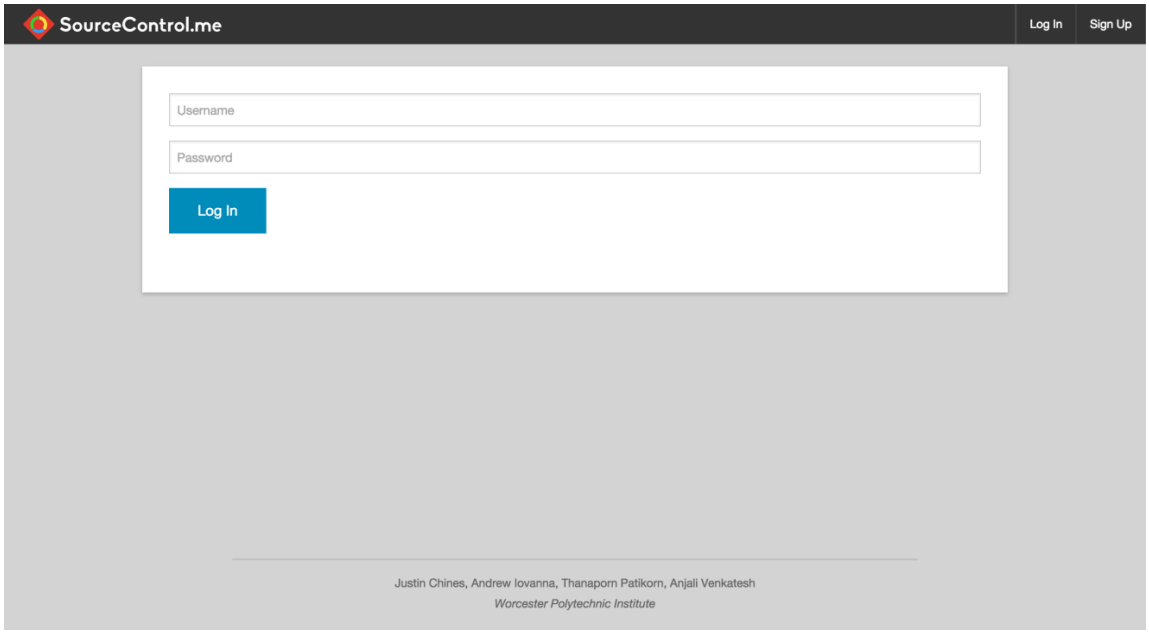

Figure 9: Login Page

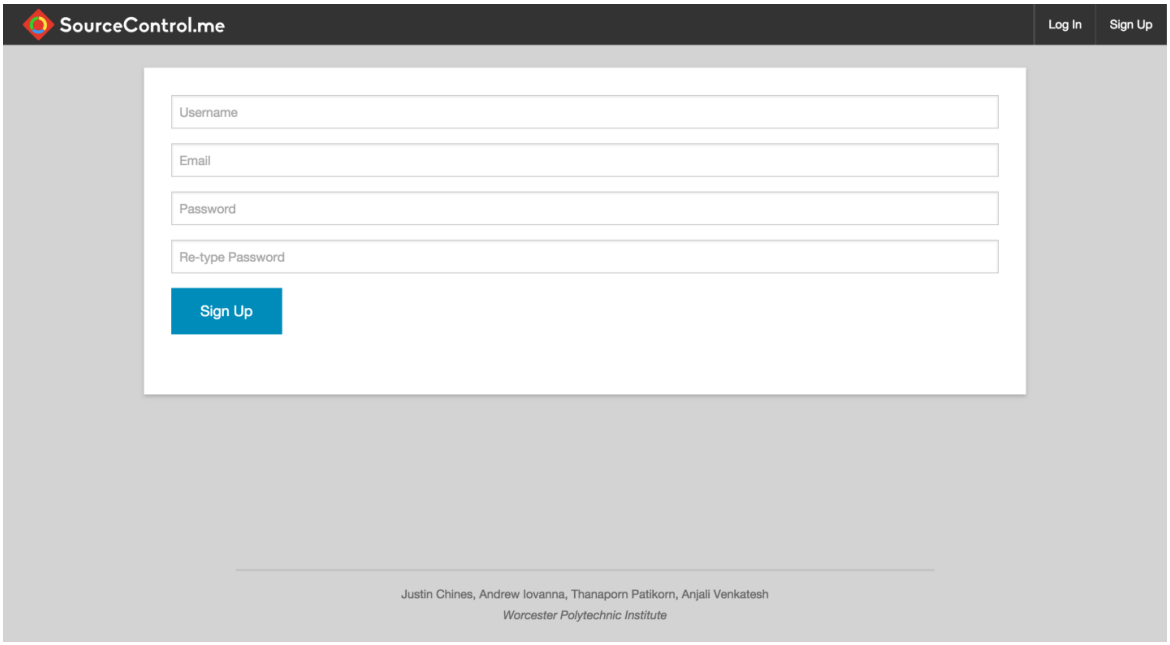

Figure 10: Sign-Up Page

Repository Page / Graphs

Following account creation, the next goal was to add some basic usability to the product in the form of adding a Git repository and having it display basic information. Again, we created HTML files built off of the base template, views to process data, and models to store the data

(described in Section 3.3.2). We used pygit2 to clone and retrieve Git repositories on the server and to retrieve information from them. Celery is a library to implement asynchronous job queues with support for scheduling, which supports Django out of the box[7]. Using Celery, the product asynchronously clones the repository, allowing the user to do other tasks on the site while the download finishes. The view stores repository information, as well as each commit, in the database.

With a repository added and relevant information stored in the database, we added a page to visualize this data. This prompted the addition of two new HTML templates, one for viewing a list of all added repositories (Figure 11), and another for viewing details about a specific repository. The repository list view also contains a big '+' icon which brings up the form to add a new repository (Figure 12). The detailed view (Figure 13) shows the number of files, commits, and authors on the repository at the top, as well as a dropdown to switch between branches. There is a second panel that shows all of the contributors and their number of commits. The final panel shows two graphs: a pie chart detailing the levels of contribution of each author, and a bar chart displaying the number of commits over the last week. At this stage, the product was minimally usable. It displayed the same information that was easily available on Github, the most commonly used site for hosting Git repositories. Adding our last feature, customized reports, made our product unique and useful to the professor.

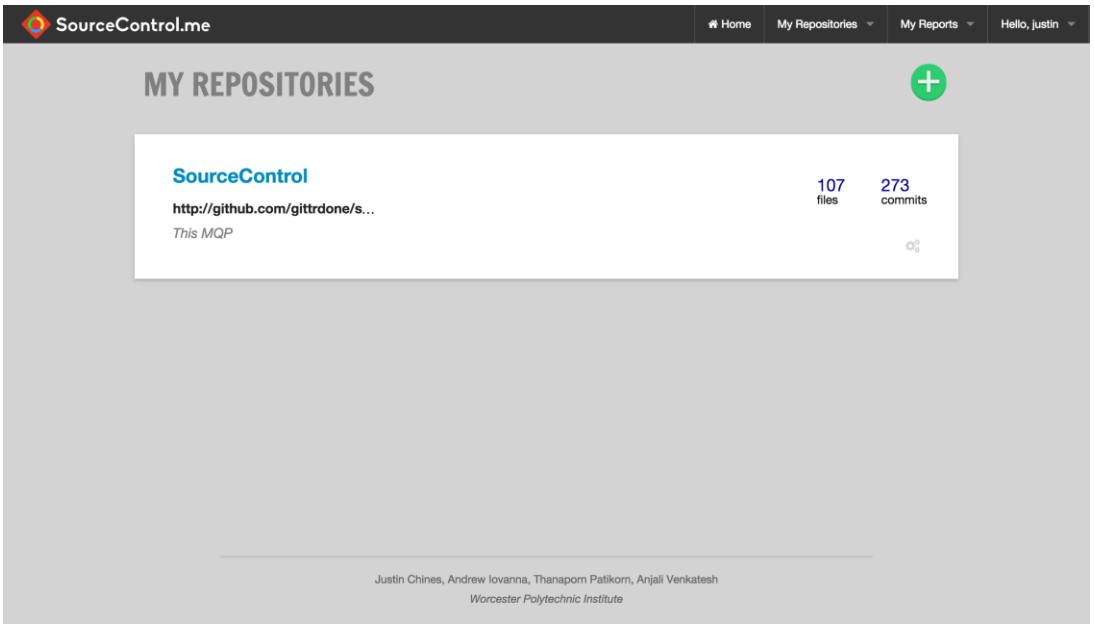

Figure 11: Repositories List View with One Repository

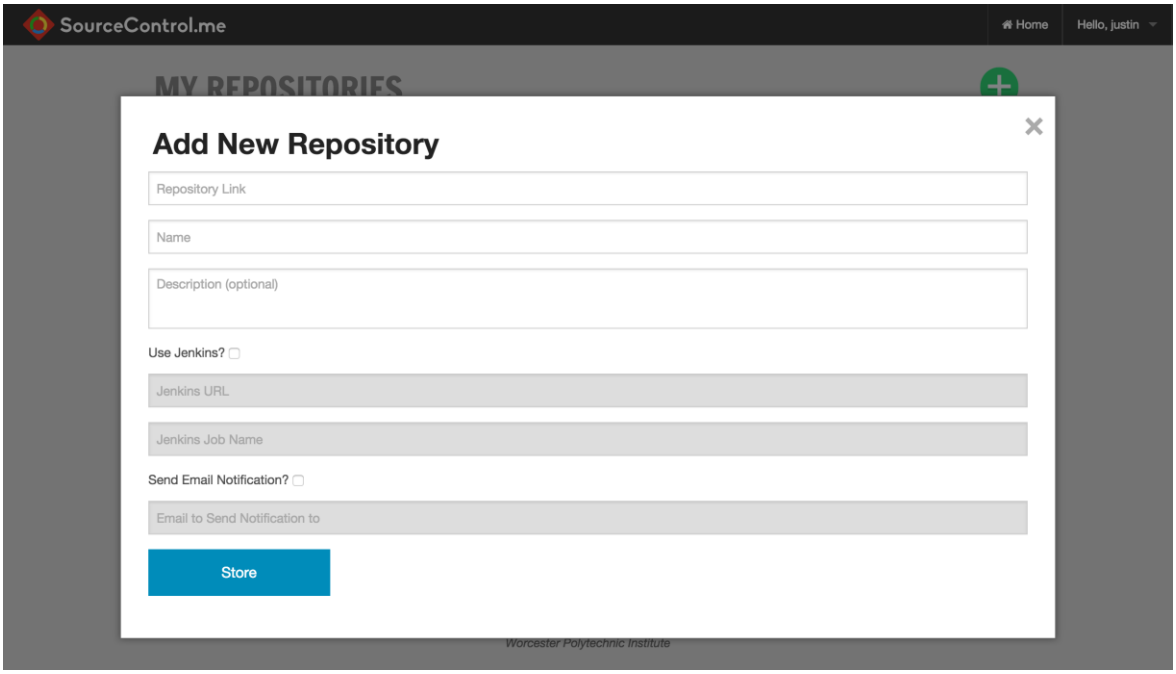

Figure 12: Form to Add a Repository to Track

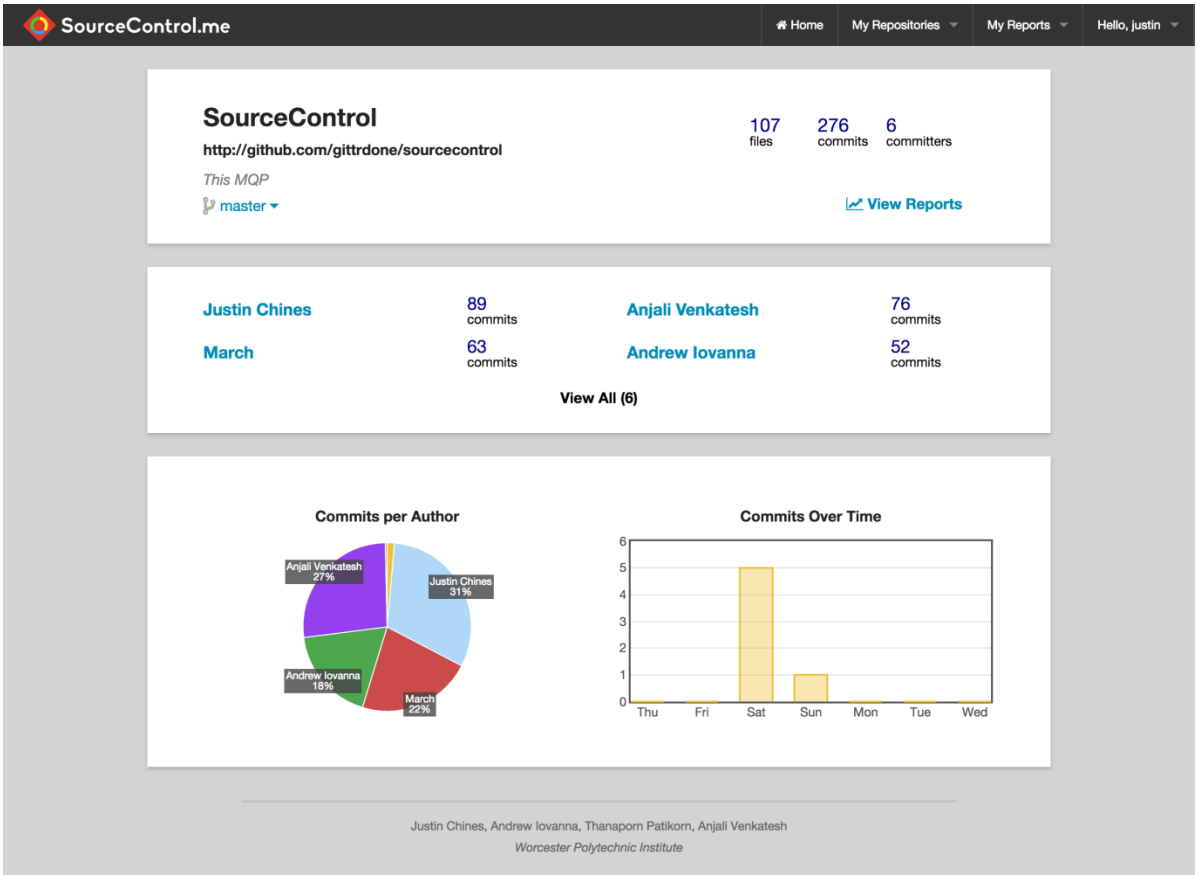

Figure 13: Detail View Showing Information for the Repository

User Page

We created a user page (Figure 14) to help the professor learn more about individual students. When viewing the repository of a team, the professor might want to know more about a certain student. We made each student name a link to a page that gave more details about their contributions. This page displays that users number of commits for the project, a pie chart showing the number of lines deleted vs. added, commits for the last week, and the number of lines added per commit. Knowing how many lines the student added and deleted can help determine if he is contributing all of his week's work in one commit (a bad practice), and if he is dragging the team backwards by deleting or editing too much code. A bar chart of commits per day for the previous week can show whether the student commits a little bit every day, or all-at-once on the night the code is due.

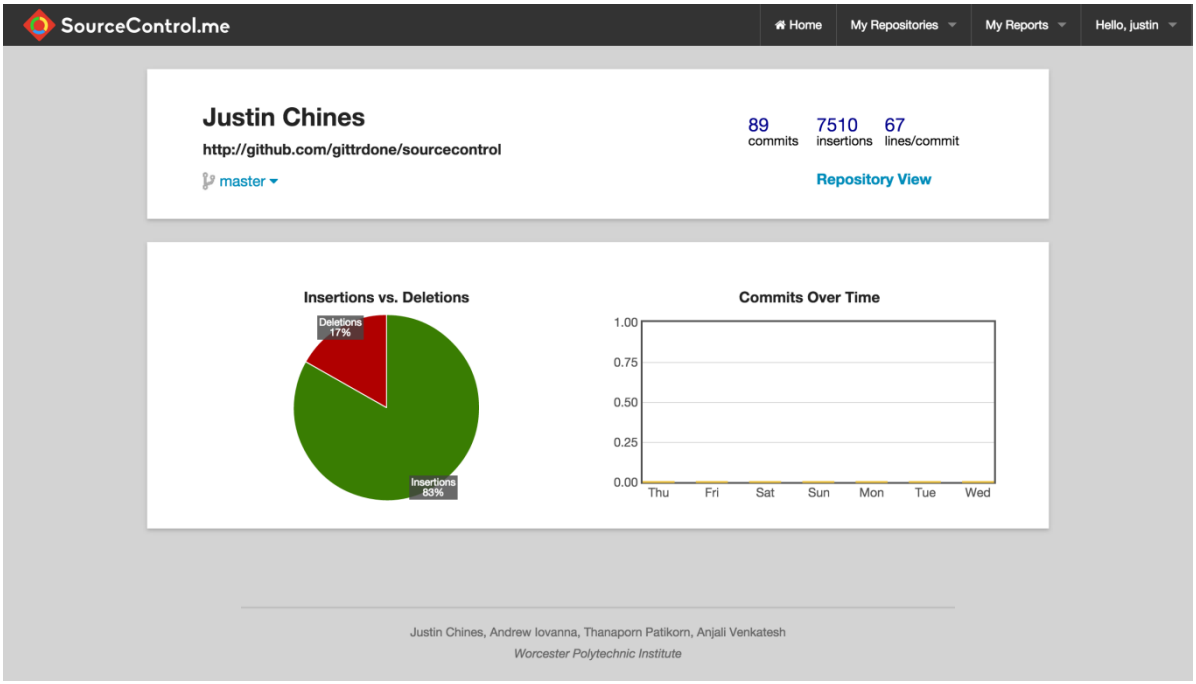

Figure 14: Detailed User Page

Reports / Queries

Our initial plan did not include creating custom reports. Instead we planned on identifying metrics that can indicate a weak or strong performer — committing less than ten times over the project, for example — and including this information on the reports page. When it became clear that this was not the best approach, our advisor suggested letting him customize these metrics.

Creating customized reports required three new pages, both very similar to the pages for repositories. One is a form view in order to create a report consisting of up to 10 queries (Figure 15), the second lists all reports for a repository (Figure 16), while the third is a detailed view of one report (Figure 17). A view was created to process the queries made in each report and store them individually.

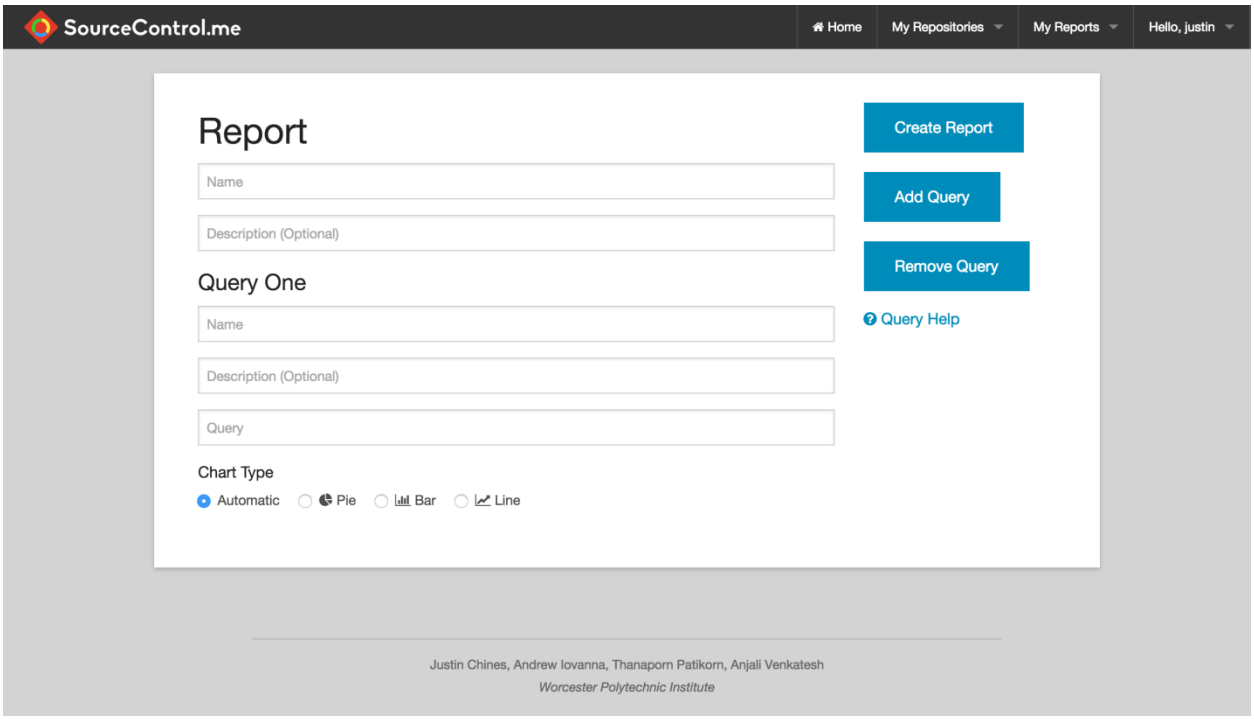

Figure 15: Adding a Report with Queries

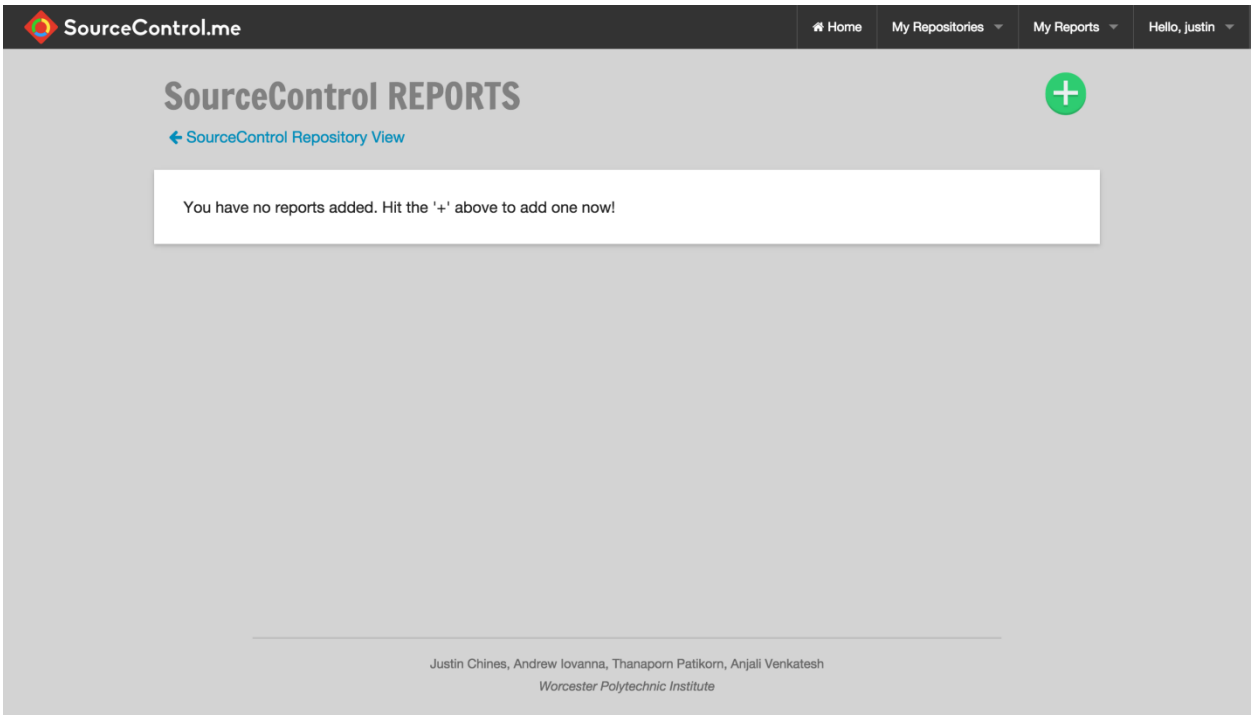

Figure 16: Reports list view, with no reports added

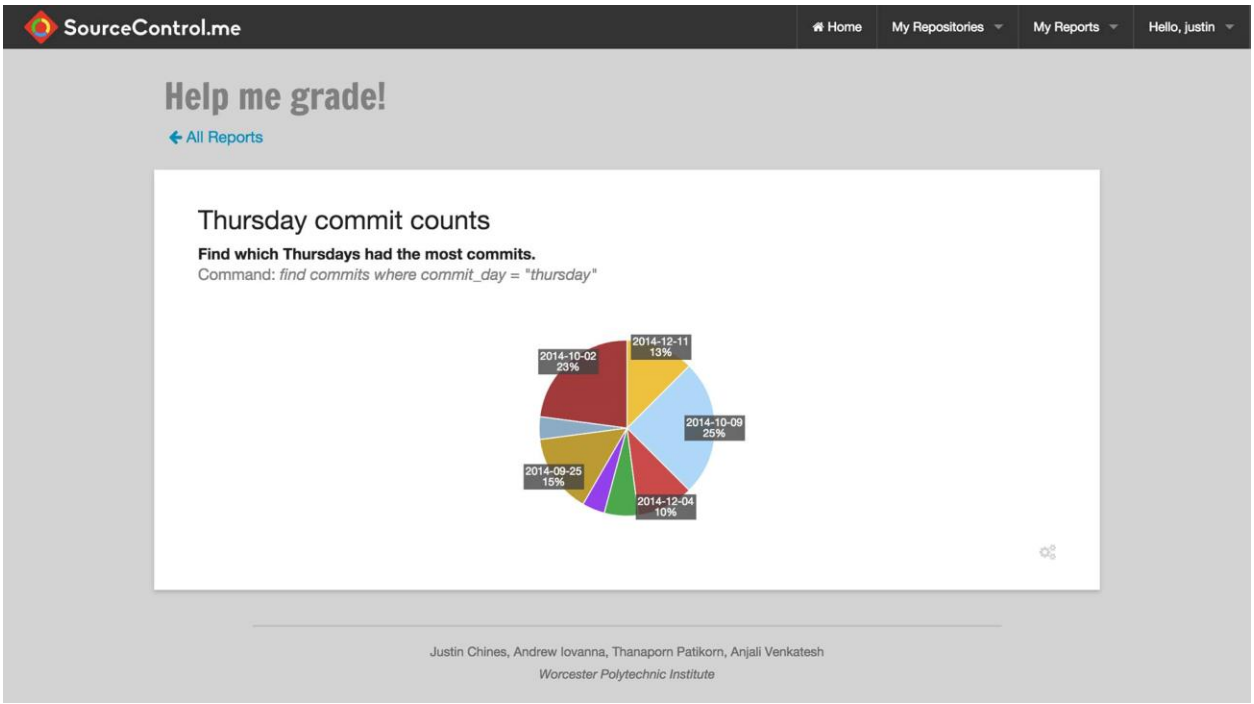

Figure 17: A Report with One Query

Mobile Access / Re-sizing Browser

With all of the features in place, the team turned its attention to increasing the usability of the site. We edited the styling so that the site is clean and intuitive on mobile devices. It is hard to predict all of the scenarios that the professor might use our website, so it might be useful for him to access it on the go, while that certainly is not the main use case. We added changes to make our site display well on browser windows of any size (Figure 18), which might come in handy as the professor might be using half of his screen space on a student grade book.

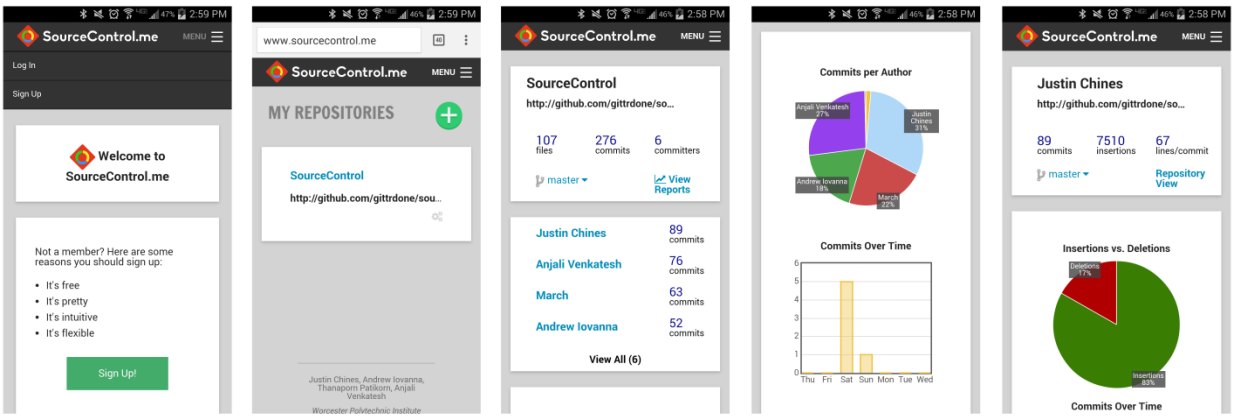

Figure 18: Mobile View of SourceControl

Code Quality

Once all our planned features were added to the product, we refactored the code in order to fit Django's standards, such as replacing all anchor tags with URL template tags. We also significantly cleaned up the CSS, moving all repeated inline styles into the style sheet. The email notification sent when a repository is finished adding were also improved from an aesthetics standpoint.

3.5 Custom Query Language

Our MQP had the scope for a custom query language that allows the professors to specify their own metrics for student evaluation. While the professors will always need certain statistics (contributions per student, for example), the majority of evaluation criteria will be created with

their judgment. For example, the professors could choose to identify strong contributors as students on a team that have contributed more than 10 commits in the last week; a list of such students will be returned by our query language. Our language allows flexibility going forward without editing the code.

The process of developing the query language began by specifying what words would be important in the language. The professor will want to know about Users (students) and Commits, and he will also care about the Time of commits. Another important piece of information is Additions and Deletions (lines of code) within commits.

The query language was created with ANTLR, a tool for generating parsers. Parsers convert computer languages from the form people use to a form that software can understand. Writing parsers is no easy task, but ANTLR simplifies the process by generating the parser entirely. The result of the parser is used to pull information from our database. The grammar can be found in Appendix E.

3.6 CHARTING LIBRARIES

In order to graph the data we collected, we decided to use a Javascript charting framework. There were several Javascript charting frameworks researched: D3[8], Flot[9], FusionCharts[10], Google Charts[11], et cetera. With the help of these frameworks, our development tasks were reduced to setting up API endpoints for the raw data, and styling the output graphs to be visually pleasing and useful.

A powerful Javascript charting framework we came across was D3. D3 could create all common graph types, as well as animations. However, D3 came with a caveat: the code required to generate D3 charts was more complex than that of most other frameworks, and it had a very steep learning curve. Dimple (dimple.js), an extension of D3 simplified the creation of basic graphs. However, the team felt that it sacrificed aesthetics for simplicity. It also required that data is saved locally in .tsv files, which added complication to development.

When we compared the features, documentation, and support of different charting frameworks, Flot fared very well. It made attractive graphs and required relatively uncomplicated

code. Once an API endpoint was set up for the data to be displayed, displaying the chart was a straightforward process. There were plenty of example snippets on the Flot website, and it was also well-documented. All these features made it a strong candidate for our visualizations.

3.7 USABILITY FEEDBACK

Early meetings with the advisor proved that the site was not as easy-to-use as the team had hoped. Per professor suggestion, the team conducted usability studies to iron out user interface issues and to fix usability problems. Feedback gathered from these interactions influenced the design and functionality of our product, helping us achieve our goal of creating easy-to-use software.

The goal was to gather detailed, qualitative feedback on our project, areas that need improvement, and ideas for content. In-person, informal interactions with friends were identified as the best option, as the format allows for observation and conversation. A team member can take notes on tasks that the user struggles through, and can then have a dialogue about why the task was difficult and any potential fixes. Users of software are often very opinionated, and will share their thoughts on why a button was on a different part of the screen than expected, for example.

An online survey approach was also considered. A survey would have reached a much larger audience, and required less work, but at the sacrifice of detailed feedback. Also, surveys do not encourage subjects to interact with our product in as much depth. They are also not the ideal platform to gather the qualitative feedback that our team was looking for. The in-person conversation was conducted according to Appendix A - User Study Procedure.

The results follow no formal analysis methods. The task completion times are reviewed to identify which tasks take longer than others, which could be indicative of the need for improvements. These are compared with the qualitative data - if a task took disproportionately long, and the tester noted that subjects struggled with the task, then it is clear that changes need to be made. In this case, feedback on the task in question are reviewed to generate ideas for improvements.

4 RESULTS

Input was gathered from six friends of team members on the usability of our project. These studies show that some tasks were simple (creating an account and adding a repository) while others were not straightforward (finding out repository information, creating a report). Users made their sourcecontrol.me accounts and added repositories with no struggles or hesitation, indicating that our labels for these actions were sufficiently clear.

Most users' first instinct was to click on the repository name to find out more information, which is a good sign. Two users, however, did not find this intuitive, instead looking around for some time before using the navigation pane. This shows that it is helpful to have navigation in the active area of the screen and in the navbar at the top, and that perhaps our active navigation options could be a bit clearer. Users indicated in-person that the waiting message for uploading repository information was clear and helpful.

The most confusing task was creating a report with one query; only one user completed it seamlessly. Two other users struggled at first without having a clear understanding of what a query/report is in the context of our site, but were able to use the help documentation to complete the task. Two more users were so unclear on the difference between a report and a query that the task took significant time and explanation from a team member. This shows that our page for adding a report with queries could use some work to make the difference more clear; perhaps by adding short descriptions of each on the page, and by making it clear that a query is a component of a report. On the other hand, the end-user for this product will already know what reports and queries are, so it is possible that the page is ideal as it stands.

Throughout the course of the interactions, users pointed out minor issues that slipped past our focus during development. Multiple users tried clicking on the highlighted sections of the pie chart on a repository page, expecting it to take them to the page for the highlighted user. It was also noted that the pie chart on a user page does not have highlighting, which is inconsistent with the pie chart on a repository page.

Users also submitted their own ideas for improving the site. One suggestion was to have each query be a set of dropdowns, so the user would select an action (Find, Get, Count), an object

type (Commits, Users), etc. Another suggested removing the navigation to return to the repository page from the user page, as it is inconsistent with how other pages handle navigation.

Another benefit of the usability study was that users noticed bugs that we had not. When opening the query help page, the navbar will say that the user is logged out, even though they are not. The other bug is the branch dropdown breaking when a repository has only one branch.

5 FUTURE WORK / CONCLUSION

Moving forward with the product that we have developed, there are some areas with room for improvement. There is greater scope for more advanced data visualizations, as well as expanding the query language. Programming language-specific features such as test detection and calculating changes per class could also be added. While none of these are critical to the functionality of the product, since the project is open source and available for Github, future contributors can easily add to it.

Currently, our product displays simple graphs for commit data in the form of pie, bar and line charts. Future contributors could devise more advanced graphs depicting tree or network-based relationships between coders, commits et cetera. Advanced frameworks like d3.js can also be used to create animated info-graphics, which could show how the repository or committers' contributions have morphed over time.

Improving and expanding the query language would allow for new data types to be displayed. If language-specific support is added, the query language could return results for Classes, functions, or tests. It can perhaps be improved to return results that compare the teams competing in the course to help evaluate team performance. It is otherwise hard to predict expansions to the query language.

Another potential addition we discussed was language-specific features such as calculating changes per class and function or method to detect any abnormal commit activity. Detection of unit tests which follow naming conventions such as those of JUnit and PyUnit can also be added as a feature to check the number and distribution of tests across the repositories.

6 REFERENCES

1. "A Visual Guide to Version Control." (2014) Retrieved from http://betterexplained.com/articles/a-visual-guide-to-version-control/

2. Skerrett, Ian. "Eclipse Community Survey 2014 Results." (2014) Retrieved from http://eclipse.dzone.com/articles/eclipse-community-survey-2014

3. Tufte E.R. (1983) The Visual Display of Quantitative Information, Graphics Press, Cheshire, Page 57

4. Tufte E.R. (1983) The Visual Display of Quantitative Information, Graphics Press, Cheshire, Pages 96-105

5. Django Documentation (2014) Retrieved from https://docs.djangoproject.com/en/1.7/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names

6. Washington University. Retrieved from http://faculty.washington.edu/jtenenbg/courses/360/f04/sessions/schneidermanGoldenRules.html

7. Celery (2013) Retrieved from http://www.celeryproject.org/

8. D3 (2013) Retrieved from http://d3js.org/

9. Flotcharts (2014) Retrieved from http://www.flotcharts.org/

10. Fusioncharts (2015) Retrieved from http://www.fusioncharts.com/

11. Google Charts (2015) Retrieved from https://developers.google.com/chart/

12. Heroku (2015) Retrieved from https://www.heroku.com/

7 APPENDICES

A – USABILITY FEEDBACK

The team interviewed friends that were familiar with the software engineering course to get feedback on the usability of our website.

(Purpose of Project): Our team is developing a website for use in a computer science class at WPI. A professor using it can submit git repositories to 'follow' and learn information that can help them evaluate and provide feedback to students. Information from these repos can be used to identify good and bad contributors, non-productive development patterns, and can illuminate team workflow inefficiencies.

-I am going to take you to our website. I will then ask you to accomplish certain steps. When you have finished that step, I will tell you to stop.

-First up, I will be asking you to make an account. (Bring them to sourcecontrol.me and begin timer).

(On finish of each task, ask "What did you find difficult or confusing about this task, if anything?" Stop timer.) Take notes as they perform the task… like if they click the wrong button. Then take notes as they talk about the task.

Note time, restart timer, and give next task.

List of tasks:
-Create a user account
-Upload a Git repo (provide them a link to a Git repo, lest they waste their time looking for a hosted repository)
-Find out more information about this first Git repo (we want them to click the name and go to the detail view)

-Add two more git repos (again, provide links)

-On the first repo you made, create a report with one query that returns users with more than 10 commits

-Logout

B - User Stories

User Story

Basic Functionality

- As a project manager, I want to see stats for each contributor in a project.
- As a project manager, I want to see stats of the project.
- As a project manager, I want to use built-in metrics to determine the strengths and weaknesses of the team.
- As a project manager, I want to easily switch between several different repositories that I manage.
- As a student, I want to see stats of my project and myself.

Repository

- As a project manager, I want SourceControl.me to remember each of my repositories.
- As a user, I want SourceControl.me to remember my identity e.g. manager
- As a user, I want to use SourceControl.me on the repository stored locally in my machine without having to upload anything.
- As a user, I want to use SourceControl.me on a project stored on an online repository using desktop devices or mobile devices.
- As a user, I want to view the most up-to-date data whenever I use SourceControl.me, or at least have a "forced update" function that I can use.

Data Display

- As a user, I want to be able to use my own metrics on a repository.
- As a user, I want SourceControl.me to remember my custom metrics.
- As a user, I want SourceControl.me to point out interesting information, e.g. a student/contributor that break the built twice a day, instead of just flooding me with all the information.
- As a user, I want SourceControl.me to show data in an organized manner.
- As a user, I want to see not only numbers representing data, but also visualization of data so that I can easily understand the data.

- As a user, I want SourceControl.me to have a search function so that I can find specific data or metrics easily.

C - SPEC SHEET

Description of Product

- Allowing software engineering professors to see information related to the projects:
    - raw statistics
    - built-in metrics
    - keep track of branches
    - allows for custom metrics
- Creating visualization of the information above
- Web-based interface (online) and (possible) desktop interface (online/offline)
- (possible) Jenkins, SonarQube, and other tools integration
- (above and beyond) suggest metrics/evaluation methods/visualizations based on project data, view history, and other users (online)

List of “raw” statistics

- A number of errors per line of code and overtime
- A number comments per line of code and per function
- A list of commits overtime
- An amount of commit comments per commit
- Changes in lines of codes per commit
- Test cases and code coverage
- Proportion of commits per team member
- Bad habits (too many one line changes, committing an entire feature at once, etc.)

List of built-in metrics

- Code quality and integrity
- Characteristics of contributors e.g. usually work at night
- Role of contributors e.g. testers

Requirements

- Git version control system
- (non-IE) Web browsers
- JavaScript
- Internet access for online interface
- Access to specified projects

D – ANTLR Grammar

```
grammar SourceControl;

options {
  language=Python2;
}

command: find | get | count;

find: 'find' query;
get: 'get' getList 'from' query;
count: 'count' (getList 'from')? query;

query: dataType ('where' condList)?;

dataType: 'users' | 'branches' | 'commits' | 'files';
cond: attrName comparator value;
condList: cond (',' cond)*;
getList: attrName (',' attrName)*;
value: INT | STRING | vlist;
attrName: ATTR_NAME;
valueName: STRING;
vlist: '[' value (',' value)* ']';
comparator: '>' | '<' | '=' | '!=' | 'in' | 'not in' | 'contains';

INT: [0-9]+;
STRING: '"' [ _\-a-zA-Z0-9]+ '"';
ATTR_NAME: [_a-zA-Z0-9]+;
WS: [ \r\t\n]+ -> skip;
```

E – Ben Schneiderman's Eight Golden Rules of Interface Design[6]

1: Strive for consistency.
Consistent sequences of actions should be required in similar situations; identical terminology should be used in prompts, menus, and help screens; and consistent commands should be employed throughout.

2: Enable frequent users to use shortcuts.
As the frequency of use increases, so do the user's desires to reduce the number of interactions and to increase the pace of interaction. Abbreviations, function keys, hidden commands, and macro facilities are very helpful to an expert user.

3: Offer informative feedback.
For every operator action, there should be some system feedback. For frequent and minor actions, the response can be modest, while for infrequent and major actions, the response should be more substantial.

4: Design dialog to yield closure.
Sequences of actions should be organized into groups with a beginning, middle, and end. The informative feedback at the completion of a group of actions gives the operators the satisfaction of accomplishment, a sense of relief, the signal to drop contingency plans and options from their minds, and an indication that the way is clear to prepare for the next group of actions.

5: Offer simple error handling.
As much as possible, design the system so the user cannot make a serious error. If an error is made, the system should be able to detect the error and offer simple, comprehensible mechanisms for handling the error.

6: Permit easy reversal of actions.
This feature relieves anxiety, since the user knows that errors can be undone; it thus encourages exploration of unfamiliar options. The units of reversibility may be a single action, a data entry, or a complete group of actions.

7: Support internal locus of control.
Experienced operators strongly desire the sense that they are in charge of the system and that the system responds to their actions. Design the system to make users the initiators of actions rather than the responders.

8: Reduce short-term memory load.
The limitation of human information processing in short-term memory requires that displays be kept simple, multiple page displays be consolidated, window-motion frequency be reduced, and sufficient training time be allotted for codes, mnemonics, and sequences of actions.