Collaborative Editor Environments
for Player Programs

Sonny Thai, Dr. Alvar Saenz-Otero

May 2012

SSL #11-12
Collaborative Editor Environments
for Player Programs

Sonny Thai, Dr. Alvar Saenz-Otero

May 2012

SSL #11-12

This work is based on the unaltered text of the thesis by Sonny Thai submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.
Collaborative Editor Environments
for Player Programs

by
Sonny Thai

Submitted to the
Department of Electrical Engineering and Computer Science
May 20, 2012
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Player programming competitions are becoming a more popular way of introducing students to programming as well as allowing more experienced coders to hone their skills. In these competitions, teams of individuals must work together to produce a software project that will compete against other teams’ projects in a well defined test. Collaboration is crucial in the overall learning experience and in determining the quality of the project. Although there are no standardized set of collaborative tools that exist for player programming IDEs, this paper documents the steps taken to investigate the ideal collaborative tools for editing player programs. These tools are a combination of frontend and backend features. Initial testing shows positive feedback from users of older systems. Future work consists of larger scale testing to perfect the standardized collaborative toolset.

Thesis Supervisor: Alvar Saenz Otero, PhD
Title: MIT SPHERES Lead Scientist
Acknowledgements

This work was performed primarily under contract NNH11CC26C with DARPA as part of the Zero Robotics program. The author gratefully thanks the sponsors for their generous support that enabled this research. The author also thanks Dr. Alvar Saenz-Otero and the SPHERES team for support and feedback.
## Contents

1 Introduction ........................................... 8  
   1.1 Collaborative Editing ................................... 8  
   1.2 Player Programs .................................... 8  
   1.3 Problem Statement .................................. 10  

2 Background ........................................... 11  
   2.1 SPHERES ........................................ 11  
   2.2 Zero Robotics ....................................... 11  
   2.3 Initial Standing of the Zero Robotics IDE .......... 13  
   2.4 Initial Standing on the Collaborative Tools for the Zero Robotics IDE 18  

3 Backend Design ........................................ 21  
   3.1 Locking Editor .................................... 21  
   3.2 Merging Editor .................................... 23  
   3.3 Real-time Editor .................................. 25  
   3.4 Comparison ....................................... 27  
   3.5 Decision .......................................... 28  

4 Frontend Design ....................................... 31  
   4.1 User Interface Principles ............................ 31  
   4.2 Principles for Player Programs ....................... 33  
   4.3 New Features ...................................... 34  
   4.4 Summary of Designs ................................ 39  

5 Implementation ....................................... 41  
   5.1 Backend Implementation ............................. 41  
   5.2 Frontend Implementation ............................. 44  

6 Testing ............................................... 52  
   6.1 Clicktracking System ................................. 52  
   6.2 User Testing ...................................... 54  
   6.3 Survey Results ................................... 62  
   6.4 Clicktracking Results ............................... 63  
   6.5 Conclusion ...................................... 64
List of Figures

1  Zero Robotics Projects Tab .............................................. 13
2  Zero Robotics Editor View ............................................. 15
3  Global Variable and Procedure Return Types ................... 15
4  Zero Robotics Compiler .................................................. 16
5  Simulation Settings Screen ............................................. 17
6  Global Locks ............................................................... 19
7  Locked Project ............................................................. 19
8  Comparison of Three Different Backends ......................... 28
9  Original Zero Robotics UML Diagram .............................. 42
10 Updated Zero Robotics UML Diagram ............................... 43
11 Revision Status Display ................................................. 45
12 Commit Comment ........................................................... 46
13 Status Marker Descriptions ............................................ 46
14 Status Markers ............................................................. 47
15 Individual Merge Messages .............................................. 48
16 Merge Content ............................................................. 49
17 Updated Menu Bar ........................................................ 50
18 Merge All .................................................................... 51
19 Clicktracking SQL Structure ........................................... 53
20 Survey Responses ........................................................ 62
1 Introduction

1.1 Collaborative Editing

Software systems are getting more and more complex every day. In order for software teams to keep up with past development speeds, companies must hire more developers as well as find innovative ways for them to collaborate. Problems such as concurrency control and code integration must be dealt with so that teams can contribute to the same project in parallel. This has given rise to much thought about collaborative programming, where software teams have a methodology of designing, implementing, and integrating software for a single system. Several modern solutions such as Gclipse[1] and Cola[2] rely on real-time editing to deal with concurrency issues as well as provide instantaneous information. While they certainly provide useful properties, in this thesis I investigate whether or not they are the ideal editing environments for player programs.

1.2 Player Programs

Besides complex software systems being developed in the private sector, there are other types of software projects that require a great deal of team collaboration. Competitions such as Battlecode[3], held yearly at MIT, allow teams of developers to design and develop a "player program" to compete in a computer game. The details of the game are different every year, but the challenge is always the same. Teams must combine battle strategy, software engineering and artificial intelligence to construct a player program to best their opponents. In doing so, teams must learn and incorporate path finding, distributed algorithms, and network communication in their players. Even though the grand prize is the main driving force for the contestants,
in the end, every player gains something from the competition. It could be as simple as being the first time they applied their software skills to an open competition to gaining experience working with a team on a large scale software project to learning swarm AI and A* path finding techniques. Competitions such as Battlcode are great ways to encourage students to go above and beyond their basic class assignments to create pieces of software that reflect their true ability, and to hone their skills even further.

It is clear that a great deal of collaboration must be put into developing these player programs, but there is an insufficient amount of tools to allow developers to do so. At first one may try to apply results from existing collaborative editing research papers to the IDEs of these player programs, but taking a step back, one can see that player programs are a unique type of project. While all complex software projects go through iterative phases to continually improve themselves, player programs iterate much faster and must remain in a "compilable" state. In order for a contestant of a player program project to assess their player, they are usually provided with a simulation tool that can run a match between two players at any time the contestant wants. For this simulation tool to be of use, however, the contestant’s project must remain "compilable". For example, if a contestant wants to see how changing the value of a certain constant in their player program affects the outcome of the match, they should be able to continually change the constant, re-compile their project, and run the simulation tool. They should be able to do this as if they were the only person editing the project, even if their team is also working on another piece of the project. With this in mind, applying the real-time component of emerging IDEs may not necessarily be the best enhancement for developers working on player programs.
1.3 Problem Statement

Developing player programs is a great way to get groups of students interested in programming at an early age, but the lack of a standardized collaborative environment for these player programs greatly hinders the students abilities to work and learn together. In this thesis, I will research the concepts and tools that will provide the ideal collaborative tools for working on player programs. I will then implement and test my vision of the ideal editor. Lastly I will analyze and conclude my findings for potential future work.
2 Background

2.1 SPHERES

My thesis will be based on work performed on Zero Robotics, which is one of the programs under SPHERES. The MIT Space Systems Laboratory developed the SPHERES (Synchronized Position Hold Engage and Reorient Experimental Satellites) laboratory environment to provide DARPA, NASA, and other researchers with a long term, replenishable, and upgradable testbed for the validation of high risk metrology, control, and autonomy technologies for use in formation flight and autonomous docking, rendezvous and reconfiguration algorithms. These technologies are critical to the operation of distributed satellite and docking missions such as Terrestrial Planet Finder and Orbital Express.[4]

2.2 Zero Robotics

Zero Robotics is a robotics programming competition that utilizes the SPHERES satellites inside the International Space Station. It was created in 2009 by the MIT Space Systems Laboratory (SSL) and astronaut Greg Chamitoff with the goal of opening research on the International Space Station to large groups of secondary school students[5]. The goals of Zero Robotics include building lifelong skills in science, technology, engineering, and math. Much like Battlecode, Zero Robotics allows a team to develop a player program that controls a SPHERE satellite. Each year the game objective of Zero Robotics is different, but the topics involved are the same. Students must apply their knowledge of mathematics, physics, and programming to design and develop a program to meet the game challenge. Currently students can access the full Zero Robotics IDE in a web browser. There is also a simulation environment and
tool provided to the students that allow them to see how their code will work in a real match environment. Most of the competitions in the Zero Robotics tournament are software competitions. They are held to continually eliminate teams to reach the final competition. The finals are then held aboard the ISS and the finalists code are run on physical SPHERES.

With several rounds of competitions to submit players for, Zero Robotics contestants must quickly iterate through versions of their projects in order to reach their best project. Also, with the supplied simulation tools, it is important that the projects remain in a "compilable" state. With these requirements, Zero Robotics projects fit the description of a "player program". Since a single Zero Robotics team may consist of up to 20 students, it is very important for the collaborative tools provided by the IDE to be sufficient. In addition to a team having up to 20 students, the 2011 Zero Robotics tournament also had a concept of alliances. After the second competition, the top 54 teams formed 18 alliances of three teams each. Going forward from there, each alliance was responsible for submitting one player program for future competitions. With a single team comprised of up to 60 students in three different geographic regions, it is even more important that the collaborative tools be sufficient. My thesis will explore the ideal editing environment for developing player programs. Although the collaborative enhancements I will be working on will be targeting the existing Zero Robotics IDE, the lessons learned should be applicable for player program editors in general.
2.3 Initial Standing of the Zero Robotics IDE

To better understand the editor I will be starting from, in this section I will provide an overview of the Zero Robotics IDE. In the projects tab, users can browse their own projects, browse projects that are shared by teammates, and create new projects. They can also access specific revisions of projects and choose to share their projects with teammates. A detailed guide of the main actions available in the projects tab is shown in figure 1.

Figure 1: a. Create a new project  
          b. Access projects and revisions  
          c. View or set shared projects

The more interesting features lie in the ZR IDE tab. Here we have all the tools that an editor requires to be sufficient for developing player programs. First, there is a generic text editor in the middle of the screen. Currently the SPHERES API is only accessible through the C programming language, and so users must only program in
C in this text editor. Users are only allowed one class file per project, but they are allowed to create as many procedures and global variables within the project as they like (so long as the whole project is within a size limit). Variable types are limited to signed/unsigned chars, ints, and floats. Procedure return types and arguments also have the same restriction, as shown in figure 3.
Figure 2: This is the view of an empty project in the Zero Robotics IDE.

Figure 3: When adding a new procedure, in addition to void, there are three valid return types. These types also represent possible global variable types.
Once the project is ready, the user can then compile it to find out any warnings or errors that exist. A code usage percentage is also shown to the user to remind them to stay within the project size limit. After the project has been successfully compiled, the user can finally run some tests, or simulations, with it. A comprehensive popup allows the user to set game constants as well as set a particular opponent to run the simulation with. With the feedback from the simulation, the user can refine their project and repeat the process to their satisfaction.

Figure 4: The code is compiled server side and any warnings or errors are sent to the user
Figure 5: Users can simulate with custom settings to more effectively test their code.

Along with a game manual, the ZR IDE is all a user needs to develop a fully functional Zero Robotics player. However, this task is too much for an individual alone. Teams of high school students along with a mentor spend months perfecting
every line of code in a project. Collaboration is an integral part in developing a Zero Robotics player program and the IDE must provide the necessary tools to make it possible.

2.4 Initial Standing on the Collaborative Tools for the Zero Robotics IDE

When the Zero Robotics 2011 tournament was officially released in September, the collaborative tools provided to contestants could be considered the bare minimum. Students were allowed to share their projects with their team, but in order for anyone to edit a shared project, a global lock on the project had to be obtained. This meant that there could only be a single person actively working on a specific project at any given time. Non-lock holders could not even run simulations with the existing project code). All they were able to do was refresh the page to view the latest version of the project. With alliances of up to 60 students, these limitations were clearly a great bottleneck in the development process.
Figure 6: Global locks handle concurrency issues, but can greatly hinder efficiency.

Figure 7: Once a project is locked by one user, no one else can edit the project.

Throughout the 2011 tournament several enhancements were made to the Zero Robotics IDE to assist project collaboration. The enhancements were limited to small changes in order to avoid forcing a new IDE upon the users, but they improve on at-
tributes that the ideal editor should certainly take into account. First was the ability to view the latest version of a procedure by simply clicking on it. This improved the feedback system, since users normally had to refresh the whole page to ensure that they were viewing the latest version of a project. Next was the ability to simulate a project even if a lock was not held. With this, users without the lock could still be very productive. They could gather useful data for the team by seeing how the latest version of a project fared against standard players (that were provided) or other projects the team created. Although the existence of a global lock still greatly slowed down progress, this feature was a big step towards unconstrained editing. Last was the addition of a chat room system where each project was its own room. With this, users could see who else was viewing the project in real-time and communicate with them about the project. Instead of having to coordinate the use of another chat system or communication tool, users were provided one that automatically placed them in the same room as others viewing the same project. This addition was a great enhancement in providing real-time feedback.

Although the Zero Robotics IDE took a huge step forward with these enhancements, it was still far from being considered an ideal collaborative editing environment. In the next two sections I discuss my designs for turning this into a reality.
3 Backend Design

The main issue with collaborative editing is currency control. When researching different types of editors, I came up with three potential ideal editors for Zero Robotics that approach this problem differently. They all solve the problem, but still have their own strengths and weaknesses. They are a locking editor, a merging editor, and a real-time editor.

3.1 Locking Editor

A locking editor is a simple, yet effective way of dealing with concurrency control. In order for a user to edit a project, that user must hold the projects lock. There is only one lock per project, and so this naturally limits one editor of a project at any given time. With these conditions, concurrency control is no longer a possible problem since there cannot be concurrent updates to a project.

One major strength of a locking editor is that locking is a concept that first-time programmers will easily grasp. A simple metaphor is that there is only one computer to work on. Multiple teammates can look on the screen (read access), but there can only be one person actively working on the project (write access). While locking in general does pose some more complicated situations, such as deadlock and livelock, this should not be a problem since users will not be able to obtain more than one lock at a time in the Zero Robotics IDE.

Although a locking editor solves the problem of concurrency control, another problem that arises is that the way it solves the problem is also its biggest weakness. By not allowing multiple writers at once, users cannot step over each other, but this also
greatly limits efficiency. The previous metaphor becomes a reality. Even though there may be multiple computers for a team to work with, the concept of locking limits them to one user editing a given project at a time.

The limitation of a single writer may instantly deter most from choosing a locking editor, but there are many enhancements that can be made to improve the efficiency of a simple global lock design.

The granularity of locks in the ideal locking editor should be flexible or depend on the applications that are being developed in it. In the case of Zero Robotics it makes most sense to have element level locking, where an element is a procedure or global variable. One could even wish for the granularity to extend to line-level locking as well, since the majority of users are high school students that may not have much experience with procedures. Either way, the properties of a locking editor remain the same. While one user has a lock on an element, the user is the only one allowed to make modifications to it. However, other users are still able to view the element while it is being modified.

Zero Robotics currently also has a notion of "stealing" a lock. This means that the stealer obtains the lock and the original holder no longer has the lock, with no modifications to either users' view of the project. This can be useful if a user forgets to release the lock, but can also be dangerous if users are not careful. Work may be lost if the original owner of a lock does not realize that they have lost it. In the ideal locking editor, each element will have a buffer associated with it. When a user obtains a lock on an element, they "own" the buffer for that element. Changes they type will stream into the buffer and when they save, the buffer will be cleared. With
this system, if user A steals a lock from user B, user A will receive the buffer and user A’s screen will be updated to reflect those changes. The potential work lost from stealing locks is now gone. With a locking editor, complex concurrency problems are eliminated in a simple way.

3.2 Merging Editor

A merging editor would resemble many of the Version Control Systems (VCS) that are in use today. Systems such as Subversion[6], CVS[7], and Git[8] use a Copy-Modify-Merge protocol where users first copy a version of a project from a repository into their local workspace. After modifications to the project are made, the user must merge in changes from the latest version of the project and resolve any conflicts that may have arose from other users updating the project. When the merging step is complete, this ensures that the user’s version of the project has the latest changes and finally the user is allowed to submit the latest version of the project.

The ideal merging editor for Zero Robotics would work in a similar fashion. When users view a project, they immediately check out a copy of the project. They are free to do anything with the project, as if they are the only ones editing it. They can edit and save the project as well as compile and run the simulation tool on it. If their version of the project is up to date, they are allowed to commit new revisions of the project. However, if there are conflicts between the latest version of the project and the current view of the project, the user may have to perform manual conflict resolution in order to merge and commit a new revision of their code.

The major advantage of a merging editor is that a single project can have multiple
concurrent editors that do not interfere with each other. A team can optimize one section of their code, test another section, and add a new section all at once. For development of a player program, this is a great plus because time is often an important factor of the competition. By being able to edit in parallel, development speeds are greatly increased, iteration time is reduced, and ultimately the end result is better.

While a merging editor can bring numerous advantages to Zero Robotics, there are important flaws to consider as well. The concept of personal copies, latest revisions, reference revisions, and merging may be complicated enough to scare away high school students as well as any first-time programmer. Before a new user can jump into Zero Robotics and get their hands dirty with some initial development, they will need some basic understanding of the versioning system. Having to learn this as well as general programming, more advanced mathematics and physics, and the ZR API may be a daunting task for most high school students. A second flaw is the merging of conflicts. This is a process that requires human intervention to sort out code manually. Because multiple users are able to edit the same project as if it were their own, there are situations where the editor simply cannot detect the correct final state without human input. This requires a human to spend their time tidying up existing code instead of developing new code. Even in the professional world, some developers dread the unavoidable moments when they must manually merge their work.

Since the major problems with a merging editor are its lack of simplicity and its time consuming merging mechanism, the ideal merging editor has features to solve these problems, as opposed to further increasing efficiency.

To deal with the problem of simplicity, the user interface of the ideal merging
editor must be very intuitive. Enough information must be provided to the user so that they do not need to actively search for it on their own. There must also be a limit of the information shown, to keep the interface simple and uncluttered. To solve the merging problem, a clean, intuitive merging user interface will also suffice. With a strong backend design such as a merging editor, efficiency is automatically gained and the major design decisions lie in the front-end. Another great benefit is that since Copy-Modify-Merge systems are in great use today in the workplace, users will be able to easily adapt into and from other popular VCS.

3.3 Real-time Editor

A real-time editor is an elegant solution to concurrency control. Instead of creating addition tools to restrict access or sync revisions, a real-time editor provides full access and perfect synchronization with all users viewing a given project. In a real-time editor, right as one user makes a change to the project, that change is instantly propagated to all other users so that everyone sees the change and everyones views are consistent with each other. The change can be the addition of an element, deletion of an element, or even as small as one character change in an element. It will be as if several users are connected to the same computer, but they each have their own keyboard, mouse, and screen. They are allowed to individually navigate through the project and make edits, but since they are on the same computer, there is only one state of the project that they all see.

With this perfect synchronization, concurrency control is no longer a problem. Even if two users attempt to edit the same element at the same time, with granular enough timing or a clever scheme, we can determine which user attempted the edit
first. If it turns out that the second edit overwrote the first edit, this will be apparent to both users because they both would see this immediately. If both users are simply typing text in the editor, they would quickly write over each others’ text, but they would also immediately see it and be able to resolve the problems that arise.

The real strength of a real-time editor is the ability to deal with the problem of concurrency without adding any overhead to the users. They do not need to worry about managing locks or merging in code. Editing a shared project with multiple teammates viewing the project is as similar as editing an unshared project. Users are constantly provided with information about their current collaborators and their edits. This type of editor closely resembles emerging editors discussed in the introduction. With a real-time editor, users have access to a natural, highly responsive interface just as if they were physically working in the same room together.

However, a significant drawback of a real-time editor is that a given project may not be in a compilable state for a long duration. If there are several editors on a single project, it is possible that they are all editing their own procedures. If the first user that finished their changes wanted to test their code through a simulation, the user would have to wait until the project reached a compilable state. Since several users are typing text into their procedures, this may not happen because any syntax error would break the compilation. It may be possible that the first user must wait until the last user has finished their changes in order to test their code.

While the ideal real-time editor would not have to add any features to deal with concurrency control, it would need some to address these issues of compilability. Since it is not trivial to determine specific states of a project once it is under a real-time
editor design, these features would have to be highly complex. A project may be un-
compilable because of a certain section of code that has been edited by several users in
several different points in time. A user friendly feature would have to be designed to
consider this and allow the user to compile with a great amount of flexibility. Similar
to the merging editor, a real-time editor has efficiency built into the backend, and an
ideal version only needs to improve upon the front end features.

3.4 Comparison

While these three editors sufficiently tackle the problem of concurrency, they each
have their own strengths and flaws. A locking editor is a simple solution that solves
many issues, but requires users to perform additional steps to actually obtain a lock
for the elements they wish to modify. It also restricts the number of concurrent edi-
tors of an element to one at a time, which can greatly hinder productivity. A merging
editor may seem to be very unrestrictive at first, but manual conflict resolution can be
a dreadful task in the software world. Understanding the design of a merging editor
may also be non-trivial for a beginner programmer. A real-time editor is an elegant
solution, but it may leave the project in an uncompileable state for long periods of
time. Multiple users will be able to edit in parallel, but only test serially. The chart
below summarizes the strengths and weaknesses of each editor.
3.5 Decision

Without the ability to allow multiple users to edit the same element at the same time, a locking editor is not a true collaborative editor. The existence of a global lock would hinder development speeds since this would only allow one editor at a time. As the locks become more granular, it is possible for users to edit different parts of the same element, but the overhead required to manage these locks also increases. In order to make edits to an element, a user must lock a section of code or the signature of the element. If users forgot to release locks on parts they were working on, this can slow down other teammates that are waiting for the lock to be released. Deletion of elements can become a nightmare if other users have locks on and are continually editing the element. Overall, the necessary work required for the user does not make the ideal locking editor the best overall collaborative editor.

It is evident that real-time editors have a great number of advantages over traditional editors through their emergence in the workplace. However, due to their lack of compilability, they are not the best collaborative editor for developing player
programs. Being able to compile a project, run simulations with it, and evaluate it is a critical part of player program development. This is how they are tested and just as it is with any other software project, testing is very important. This is especially true for player programs because of how involved testing is in the development process and how iterative the process is. Without compilability, development on a real-time editor can even be slower than a locking editor. A user will only be able to simulate their code if the whole project is in a compilable state. This may be difficult to achieve if multiple teammates are all editing different parts of a project at once. Even once the last user has finished their changes, others may have moved on to work on new sections. Any features to improve compilability would add significant overhead to the user. With only the active view of the project, it is difficult to see how a user could test only their changes. All in all, the lack of compilability of a real-time editor greatly limits its performance when working with player programs.

I decided to proceed with a merging editor design, but also incorporate real-time features wherever possible. According to figure 2, the major flaws of the merging editor are its complexity, its need for manual merging, and the work required to update a project.

The complexity can be especially daunting for a program such as Zero Robotics because the user base may contain first time programmers. However, with the use of effective tutorials or a tooltip system, this can be converted into a perk. Learning how to use one of the most popular VCS can certainly be helpful in the future. I plan on creating tests to evaluate how new users adjust to the system, but leave it to future work to create mini lessons on how to use it.
Manual merging is often feared because the interface it is presented in is confusing, or even worse, it is in a command prompt. This can be solved if an intuitive interface is provided to the users. Conflicts must be clearly noted, and the options provided must be clearly visible and flexible. Most importantly, the user must not be overwhelmed with information or available actions. I will develop my own interface for manual merging that will take all of these points into consideration.

There are several parts considered in the work required to update a project. First, the user must be alerted that there is a new revision of the project. This must occur in a non-obtrusive fashion. The user should then be able to find out which elements have been changed in the new revision. Lastly, it should be up to the user to choose to accept them or not. Keeping all of this in mind, a friendly user interface can turn the process of updating a project into a simple task. I will also include these features in my collaborative editor implementation.

With the problems of the ideal merging editor solved, the best backend design for editing player programs is a merging editor.
4 Frontend Design

4.1 User Interface Principles

When developing the front end of any application, it is important to follow a set of user interface design principles or heuristics. Some go by standard heuristics such as Nielsen’s Top 10, while others follow their own guidelines. Whichever set the developer chooses to use, I believe there are four overarching UI principles[9]: learnability, visibility, efficiency, and errors.

Learnability describes how long it would take and how effortless it would be for a user who has never seen the interface before to become familiar with it. Complex systems clearly require more complex user interfaces than simple systems, but a front end developer following good learnability principles can design a complex user interface as learnable as a simple one. Wizards and tutorials are a great way to for a first time user to be stepped through the most important features. Metaphors and affordances allow completely new tools to be as intuitive as an everyday task. But the most important part of learnability is keeping the memory required of the user low. If the user does not need to remember having used the interface in the past in order to effectively use it, then the user interface is highly learnable.

Visibility is all about what information is shown to the user. The interface can show possible actions the user can take, the state of the interface, or provide feedback from a user request. A visible user interface provides as much information as possible to the user without overwhelming the user. It must keep a balance between showing too little information and appearing too cluttered. If a certain tool is not immediately accessible on the screen, it should at least be easily found within a few clicks. This
is often achieved through menu bars and other dropdowns. A perfectly visible user interface would be able to predict the user’s next action, and immediately bring forth the necessary tools as well as hide irrelevant information. Although this may not be achievable for user interfaces today, there are still many ways to bring about high visibility.

An efficient user interface is one that allows the user to complete their task in a reasonable amount of time. The tools provided in one must each have a unique purpose and never be repetitive to use. If a certain feature must be used multiple times in order to complete a single task, the developer should reconsider the use cases of the feature and possibly improve it to better fit those use cases. Another metric for efficiency is the accessibility of each feature. Properties such as size and location and heuristics such as Fitts’s law help determine how easily and quickly a user can access the feature. Keyboard shortcuts can also improve upon this property. Just as in the visibility section, predictive logic can improve efficiency. An example is the well known autocomplete feature. By suggesting common responses or previous responses, a user will not have to type out the full text of a query. At the bare minimum, a truly efficient user interface has tools to speed up repetitive tasks and grants easy access to those tools.

Errors are an often forgotten yet very crucial part of a user interface. They cause can originate from the user (ex. leaving a required field blank) or come from the application itself (ex. server is down). In either case, errors will occur in all user interfaces and it is important to handle them appropriately. Instead of letting the system fail or alerting incomprehensible error codes, proper error handlers should explain the error in laymen’s terms to the user. In the case of small errors, the in-
terface should suggest ways to avoid the error in the future. If a more pressing error occurs, constructive methods of resolving the error should be presented. A step beyond error handling is error prevention. Instead of developing ways to fix the error, it may be possible to stop the error from ever occurring. An example is disabling a form’s submit button until all required fields are filled out. With this approach, a user interface does not force the user deal with an annoying popup. The user will automatically realize a part of the form is not complete and avoid the error altogether.

While I will incorporate all of these principles in my frontend design, I will focus on the ones that will have the most impact on player programs: learnability, feedback, and efficiency.

4.2 Principles for Player Programs

Learnability is the most important principle of a user interface that is designed for developing player programs. This is mainly due to the short duration that a player program competition lasts. They often run for only a single month up to a few months. In this time, participants must learn a brand new API, develop strategies, and iterate countless to perfect their player. The time and focus needed to learn a new editor can greatly slow down a team’s progress. Learnability is especially important for Zero Robotics because of the nature of the user base. Zero Robotics teams are often composed of high school students with little to no background of programming. In addition to learning the other topics (physics and advanced mathematics) required to create a functional Zero Robotics player, students may be encountering ”if” statements for the first time. Any early frustrations with programming concepts or the programming interface can easily turn away students for life.
Feedback is the next most important principle. The main distinction between editing a normal project and editing a game player project is that one would expect a player project to be edited simultaneously by multiple teammates more often. Because of this, the state of the project is more constantly changing and it is important that the users know about these state changes. A user may have added, changed, or deleted elements that are crucial in another user’s current work. It is important to acknowledge the latest changes and decide whether the current development work is affected by those changes. It is possible that the newest developments make the current work obsolete, in which case it is much more helpful to learn about it sooner rather than later. With enough feedback from the system, users will continually be up-to-date on the state of their projects and will feel confident that their current development will actually improve the projects.

Efficiency is also crucial for player programs. Because of the short duration of a player program competition, it is important to provide the necessary tools to make the users’ development process as efficient as possible.

4.3 New Features

In order to make efficient use of the new backend system, a whole suite of new front-end tools will be required. In this section I will describe the new features being added as well as which principles they improve.

The first new feature is a revision status display. With the new merging backend editor, users will have personal working copies of a project called "drafts" which will
reference specific revision numbers of a project. Any draft must reference exactly one existing revision number of a project (henceforth known as a reference revision) and a user may only have one draft per project. This will allow future tools to understand where this draft exists in the revision ladder. Another important revision to take into account is the "latest revision". This is the most up-to-date revision of a given project. Knowing a user’s draft’s reference revision number as well as the latest revision number will better help the user understand the state of the project, and this is exactly what the revision status display is for. Because of the importance of the information is displays, it will be located in an open and clearly visible area on the interface. It should also update in real-time to provide as effective feedback as possible. Whenever a user updates their reference revision, the status should update accordingly. Whenever another user commits a new latest revision of the project, the status should also update immediately.

In the existing version of Zero Robotics, every "save", "compile", and "simulate" action creates a new revision of the project. In the new merging editor, "saves" will only save to a user’s draft. A new "commit" action will used for creating the latest revisions of a project. Due to the necessity of a commit action, its button should be placed in a highly visible area such as the menu bar. The new system of allowing users to commit latest revisions on their own without holding a lock can pose new problems. For example, if ”User A” is unaware or forgets to notice that ”User B” has created a new latest revision, ”User A” may commit their own latest revision without pulling in the latest changes. This can potentially wipe out all traces of ”User B”’s changes in the latest revision. Another version of this problem can occur even if two users are highly attentive to the revision status display. If they both commit new revisions within the same second, one of their commits will inevitably be overwritten.
by the other one. In order to prevent this scenario, I have chosen to perform a check on all commit actions. A commit will only be accepted if the user’s draft references the latest revision. This is to ensure that the user has the latest copy of the project and has received all changes from commits. With this check, the two aforementioned scenarios would no longer be possible. The latest commit would fail because their draft no longer references the latest revision. The user must update their draft to receive the changes from the latest revision before they are able to commit their own. This fix is a good addition to the error preventative methods of the new user interface.

As described above, users will need a tool to update their draft to bring in changes from the latest revision as well as update their reference revision. I have chosen to combine both of these into a single action. That is, in order to update a draft’s reference revision to the latest revision, the user must also review the changes brought about by the latest revision. The user may very well decide to reject all changes presented to them (potentially overwriting other user content), but they will be knowingly making that decision. In order to present all of the changes brought about by the latest revision and provide the flexibility to allow the user to accept/reject each one individually, a well designed pop up interface is needed. It must be learnable, visible, and efficient. More details about this interface is described in section 5.2. Upon successfully deciding which changes to keep and which to discard, the user’s reference revision will be updated and reflected in the revision status display. The user will now be able to commit their own changes.

With a tool to update a draft to the latest revision, it would be helpful to know in advance which elements are actually out of date. A user may be focused on one section of the project when the user notices that a new latest revision of the project
has been committed. If the changes are in a separate section, the user may decide to ignore them for now to concentrate on their current work. However, if the changes are within the user’s section, it may be worth grabbing the changes now rather than later. To provide this useful feedback, I envision a status marker next to everything that can change within a project. For Zero Robotics, this means a marker next to each element. Instead of having just an on/off mode for this marker, I have decided to allow it to display different states with different colors. Rather than having to understand several different symbols, users will be able to grasp the same knowledge with different colors. I have chosen colors because the human eye can much more easily distinguish color than shape.

There are three main statuses that an element can be in. The first is the up-to-date status. This means that the draft version of the element exactly matches the latest version of the element. There is no requirement that the reference revision match the latest revision. For a global variable, exactly matching means that the sign, the type, the length, the initial value, and of course the name all match. In the remainder of this paper I will refer to all of these as the signature of the variable. For a procedure, exactly matching means that the return type and all aspects of all arguments must match. These will also be known as the procedure’s signature. In addition, the procedure’s text content must match. To show an that an element is up-to-date, I will provide a green marker next to the element. I chose green because it is often used as a sign in user interfaces to describe that all things are going well, such as a green check mark. The next obvious status is a out-of-date status. This will mean that the draft version of the element does not match the latest version of the element, as well as the reference revision does not match the latest revision. The point of this status is to show that there is a later version of the element that has some changes to it. The
change can be as small as one character difference in the editor to the whole element being deleted. For this status I will use a red marker. Red is often used in world as a warning or cautious color, such as in stop and danger signs. An appropriate warning is warranted for elements that have a red marker since it means that the elements have changes to them and an update should be performed before continuing to work on them. The last status is an edited status. It means that the draft version of the element does not match the latest version of the element, but the reference revision does match the latest revision. Instead of showing that another user has committed changes to the element, this status designates that the user has uncommitted work. This will be a good visual reminder for users who are new to a merging revision system. These users often keep their work in the draft, committing only once when everything is finished. The status marker will continually remind them to commit once they have something working, which is generally a better practice in the workplace. I have chosen blue to represent the edited status. Although blue does not have a consistent meaning in user interfaces, it is the third additive color and should not get confused for the other green or red statuses. Just like the revision status display, the status markers should update in real-time to provide valuable feedback to the user.

When red status markers appear, users know to update their project to grab the changes from the latest revision. However, a user may be unwilling to be brought away from the user’s current task to update their project if they see that there are many changes to deal with. To fix this problem, I will also add individual element updating. Users will be able to update single elements when they see red status markers appear. Although the reference revision will not update until the user performs an update on the whole project, users can view the changes that a single element has undergone without having to deal with the other changes of the latest revision.
The last user feature is another common VCS tool, reverting. With the new merging backend, each draft has a well defined reference revision. Reverting elements in a draft will bring them back to their reference revision state. This is a great example of providing tools for error handling. If a user changes an element several times while testing, but then would like to remove all of the changes, reverting the element will do just that.

4.4 Summary of Designs

To summarize the list of design decisions, below I describe each change or new feature that is to be added to the system.

Drafts: Users will no longer directly edit revisions of a project. Instead, they will work on personal drafts of projects. These drafts will start off from a specific revision of a project, known as a reference revision. Users will be able to update their reference revision to point to later versions by updating their drafts.

Locks: Since users only edit their personal drafts, locks are no longer required. The notion of locking will be completely removed from the entire system.

Revision Status Display: This display will show a draft’s reference revision as well as the project’s latest revision numbers. It will automatically update in real-time and will be easy to locate.

Commit Button: The commit button will be used to create the latest revision of a project. It will take a copy of the user’s draft and append it to the project’s revision list, creating a new latest revision. In order to use it, a user’s draft’s reference revision
must match the current latest revision.

Status Markers: Each element in a project will have its own status marker to display its state. It can be up-to-date with the latest revision (green) or out-of-date. When it is out of date, there are two sub-states. If the reference revision matches the latest revision, then it is in the edited state (blue). If the reference does not match, then it is in the missing changes state (red). These status markers will also automatically update in real-time and be easy to see.

Element Merging: When elements are missing changes, users will be able to perform a merge on them. There are many ways in which an element can be missing changes, and the actions to merge them are specific to each case. The details are discussed in section 5.2.

Conflict Merging: Conflicts occur when the user has made changes to the content of a procedure but the latest revision also has different content. Resolving the conflict is not a simple task and will require a well-designed interface to get the job done.

Merge All: Rather than merge individual elements at a time when a draft is out of date, users can use the merge all tool to update all elements that are out-of-date at once. Upon successfully merging all, the reference revision of the draft will update to point to the latest revision of the project.

Reverting: Users will be able to revert elements to bring them back to their reference revision state. All properties will be changed back.
5 Implementation

With a strong set of backend and frontend goals, I will delve into the implementation details.

5.1 Backend Implementation

In order to turn a locking editor into a merging editor, several changes need to be made. Locking needs to be removed, personal drafts need to be created, drafts need to link to a reference revision, and the server must keep records of the latest revisions of projects. Fortunately, with some clever logic, only one table in the database needs to be modified to add one field in order to accommodate these changes. Shown in figures 9 and 10 are the before and after UML diagrams describing the parts of the database that handle projects and their components.
Figure 9: This UML diagram shows only the tables related to projects in the original database.

There are individual entities for each of the element types, as well as a project itself. Arguments are tied specifically to the procedure that they were created for. To associate a project to its elements, a revision table is used. There are then two association tables to associate specific procedures and global variables to each revision. With this setup, minimal space is required as the number of revisions grow. Instead of duplicating whole projects for small changes in revisions, we only need to create a
revision entry and the appropriate associations. For locking, the system keeps track of a ProjectLock for each project. When users acquire locks on their projects an entry is inserted into the table. When users release their locks, the entry can be deleted. In order to turn this locking system into a merging system, we simply need to add a boolean field to the Revision table.

Figure 10: This is the updated database. Locks are no longer required and a boolean field is added to the Revision table.

The ProjectLock table is also no longer needed, but I have chosen to keep it for
potential future use. Instead of creating a new table to handle user drafts of projects, we can cleverly reuse the Revision table. With this system, a revision is a user’s personal draft if the draft field is equal to true. If it is false, then it is a committed project revision. The checkedInBy field refers to the user who committed a revision if the revision is a draft and it refers to the user who owns the draft otherwise. All of the other fields remain consistent with each other and so this dual use is valid. To find out the latest revision of a project, one only needs to query for the largest revisionNumber with the correct projectId.

5.2 Frontend Implementation

In this section I will describe the new frontend features that were developed to accompany the new backend system as well as the challenges I came across when doing so. To recap, the list of features discussed in section 4.3 were:

- Revision Status Display
- Commit Button
- Status Markers
- Individual Merging
- Merge All
- Element Reverting

The first feature was the revision status display. This was a simple feature, but it was important to present it in a highly visible section of the interface because of the usefulness of its information. In order to allow it to automatically update itself without any user action and in as close to real-time as possible, I used a recursive
AJAX call. This would poll the server for information on the latest revision status. On a successful callback, the client would queue up the same AJAX call to be executed in 10 seconds. Although the updates would not occur in real-time, a 10 second delay is acceptable.

Figure 11: The top right section of the IDE was a clear, open spot that users could refer to for useful information. I chose to place the revision status display there with bright red text.

The next feature was a commit button. Since I anticipated a high usage rate for this button, it made the most sense to place it in the menu bar alongside the other most commonly used tools. As an addition to the previously defined commit button, I also added a required comment to enforce good programming practices.
Figure 12: Providing a description of the changes involved in a commit is an example of good programming practice. This will allow other users to see what has changed from one revision to the next just by reading the comments.

The next features were the ones that deal with individual elements. Figure 13 concisely summarizes how status markers were determined for elements.

<table>
<thead>
<tr>
<th>Description</th>
<th>Green</th>
<th>Blue</th>
<th>Red</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements match Reference matches</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elements match Reference does not match</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elements do not match Reference matches</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Elements do not match Reference does not match</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Figure 13: It is important that the different statuses have clear states that will not overlap with each other.
However, since elements can differ in many ways, individual element merging quickly grew complex. Global variables can differ in sign, type, length, or initial value. Procedures can differ in content, return sign, return type, or any part of any of the arguments they contain. To simplify the choices users had to make, I treated the sum of the changes in a signature as a single change. With this policy, the number of possible differences was greatly reduced. Figure 15 describes the messages that are displayed to the user for each type of change. Each message is answered with the same Yes/No popup to keep consistency across the interface.
To perform the merging action on each individual element, new buttons were placed next to existing edit buttons. I also enabled clicking on the status marker itself to perform the merge. To signal that a new element exists in the latest revision, red status markers appear next to the corresponding "New Element" button. Once clicked, the elements will be automatically added into the draft. Although this may not be immediately clear to new users, it is a consistent and concise way to provide the feature. For content changes in procedures, conflicts can easily become complex. I have developed a dedicated popup interface to handle these types of element merges.

One of the most difficult parts of using a merging editor is resolving conflicts. The biggest problem is that the user resolving the conflict may not immediately understand the new latest changes and how to merge them into the user’s own changes. Another important factor is that user interfaces are often cluttered with information, making them hard to understand. I have designed an interface that only contains the necessities to keep the interface simple. It clearly presents the options available to the user and even allows for some flexibility. A detailed example is shown in figure 16.

Figure 15: These are example messages when a user tries to merge an element under a certain type of change. Adding does not display a message because the elements are automatically added to the draft. There is a separate popup for content changes in procedures because of possible conflicts.

<table>
<thead>
<tr>
<th>Added</th>
<th>Procedure</th>
<th>Global Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>
| Deleted | "This procedure has been deleted. Do you want to delete this procedure?" | "This variable has been deleted. Do you want to delete this variable?"
| Signature Change | "Do you want to turn your version of the procedure: int getDistance() to the latest version of the procedure: float getDistance()?" | "Do you want to turn your version of the variable: int currentDistance to the latest version of the variable: float currentDistance?" |
| Content Change | Separate popup | N/A |

<table>
<thead>
<tr>
<th>Added</th>
<th>Procedure</th>
<th>Global Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>
| Deleted | "This procedure has been deleted. Do you want to delete this procedure?" | "This variable has been deleted. Do you want to delete this variable?"
| Signature Change | "Do you want to turn your version of the procedure: int getDistance() to the latest version of the procedure: float getDistance()?" | "Do you want to turn your version of the variable: int currentDistance to the latest version of the variable: float currentDistance?" |
| Content Change | Separate popup | N/A |
Figure 16: To help simplify the content merging process, the interface is as simple as possible.

In this popup, conflicts are clearly denoted with large red sections. When clicked, the three areas on the right: ”My Revision”, ”Parent Revision”, and ”Latest Revision” display their own versions of the conflicted region. The user can clearly compare the sections on the right and decide which one to pick. Once the appropriate ”Use” button is clicked, the corresponding text is transferred over to the conflict box and is now highlighted green to represent a resolved conflict. For added flexibility, I have made the green area an editable textbox so that the user can edit right inside the popup. Users may not want any individual version of the element, but instead want a mixed copy. The textbox will allow the user to freely add content to the region while all three versions are clearly visible on the right.
To determine conflicts, I used Unix’s diff3 utility. When given two copies of a file and their ancestor version, ”diff3 -m” is able to produce a merged output that contains both sets of changes with clearly marked conflict regions. I was then able to take that merged output and parse it into sections of the same code and sections of differing code to feed into my ”Merge Content” interface.

Finally, I added a feature to merge multiple elements at once. This button would also update the reference revision. Similar to the commit button, I expect it to be a commonly used feature.

Figure 17: Two new features: ’Commit’ and ’Merge All’ are placed in the menu bar.

As discussed before, the interface for merging multiple elements must be learnable, visible, and efficient. Similar to the ”Merge Content” interface, I plan to achieve learnability and visibility by keeping the interface as simple as possible while still achieving its goals. For efficiency, I have set default actions for each of the individual merging actions to choose the most likely response. However, the user is free to override the defaults if they choose to.
Figure 18: The Merge All interface clearly presents each change from the latest revision. Defaults are set, but the user can ultimately choose to accept or discard any and all changes.

With an initial implementation of the new backend and frontend systems, I turn my focus towards testing.
6 Testing

In order to test the new collaborative editor, I created a suite of eight tests for users to find out how intuitive the new features were. I then had three undergraduate students who were familiar with the Zero Robotics IDE take the tests and report their findings through a survey. To also collect objective data, I implemented a clicktracking system to be run in the background.

6.1 Clicktracking System

Whenever a user clicked on one of the new features, a post was sent to an offsite PHP server describing the type of click it was. The server would parse it and insert an appropriate record into a MySQL database. Once all the tests were complete, I would then be able to analyze quantitative results of the new collaborative editor.

For commits, I compared the number of successful commits versus unsuccessful commits. An unsuccessful commit can be attributed to an unsaved project or having a reference revision out of date. If the tools provided for committing were truly intuitive, I would expect only a few unsuccessful commits compared to successful ones in the long run. For all of the merging popup interfaces, I compared their success rate as well. In all cases, the user is given an accept option, a decline option, and a cancel option. A fourth ”X” button was also available on the topright of each popup that would essentially act as a cancel. I treated the accept and decline options as successful responses and the cancel or ”x” as unsuccessful responses. Figure 19 shows the database structure created to determine these rates.
Figure 19: The user field is used to aggregate data on a per user basis. Combined with the timestamp, it estimates the duration the user spends on each popup. The other fields are used to determine what the user clicked on in order to determine the success rates of each feature.
6.2 User Testing

In this section I will describe each of the eight tests created and their objectives. I will then analyze the results and feedback from three users. For each test, users were given a project that was in a specific state and a set of instructions to follow. After finishing the test, users were to rate the test as one of the following:

1. Confusing/had problems and had to skip it
2. Confusing/had problems but was able to finish it
3. Very straightforward/no problems

**Project 1 (Committing)**
Project state:
- empty project
Steps to achieve:
1) Add a procedure `int spin(int duration)` to the project.
2) Add a global variable `int duration` to the project.
3) Commit the project with comment 'Added spin'.
4) Save.

In project 1, users are given an empty project. They are asked to add two simple elements and then commit the project.

In this simple test, all three users described the test as "very straightforward", showing that there is nothing unexpected with the commit tool.
Project 2 (Adding latest elements)

Project state:
  Void spin()
  Void revolve(int duration)
  Int duration

Steps to achieve:
  1) The latest version of this project has some new elements (variables or procedures) added to it. Update the project to bring in those new elements.
  2) Save.

In project 2, users initially have an empty project, but are told that the latest version of the project has new elements. Without being told how, they are told to bring in the new elements into their draft.

Two users believed the task was straightforward and one user rated it a 2. All three mentioned that the terms "updating" in the provided testing steps and "merging" in the user interface could have been more consistent. For future tests, this will be important to take into consideration. It is interesting to note that all three users had updated revisions as a final result. This means that they used the "Merge All" button instead of clicking on the red status lights next to the "New Element" buttons.
In project 3, users are presented with a draft with several elements already in it. There is a later version of the project and the users must update their drafts without being told how.

Again, two users believed the task was straightforward and one rated it a 2. In the comments, the user who rated it a 2 describes that the term "update" was the confusing part.
In project 4, users have procedures that conflict with the latest revision. They are tasked to resolve the project by accepting the latest revision’s content as well as signature, but told not to edit the procedure through the text editor or 'edit procedure' button that existed in the locking editor.

Two users had no trouble and one had to skip the test. In the comments, the instructions provided were confusing, especially the part about updating without the user of 'edit procedure'. It is possible that this user did not have significant experience with the Zero Robotics editor to know about the 'edit procedure' button. Another user commented that the size of the scrolling windows provided glitchy results on their screen size. It will be important to run thorough tests of the new collaborative editor with different screen sizes.
Project 5 tests how intuitive the textboxes in the "Merge Conflict" interface are. Users are provided with a conflict where all three versions have errors in them. Each of them have different errors fixed, and the sum of them provide an error-free message. Users are told to approach the problem with intuition and are told to not edit the procedure through the normal text editor.

Each of the ratings received one response and only one project achieved the expected result. One user encountered trouble with the merging tools and one simply chose to use their draft revision’s contents. The user that managed to end up with the correct resulted commented that it was not very clear that the green textboxes were editable. This is a great lesson and shows that more steps will need to be taken to indicate the flexibility of the conflict resolution interface.
In project 6 the state of the project is out of date. Users are asked to update their project and to commit a new change. The original intention of this test was to see if users remembered to update their reference revision before attempting to commit a new latest revision. However, since the previous tests show that users have always been using the "Merge All" tool to update their project and reference revision at the same time, it turns out this test may have been poorly constructed.

All three users had no problems with this test.
The state of project 7 is very similar to project 6. The draft is out of date and the user is asked to update the project. However, they are told to not delete any elements. This will test how easy the “Merge All” interface is to use when it comes to choosing which changes to accept and reject.

Two users had no problems while one user commented about a potential bug in the interface. An provided suggestion in the survey was to include a legend describing the coloring scheme. Although my intention was to provide an interface intuitive enough to not require a legend, it seems that additional information may still help some users.
Project 8 combines several of the previous tests. Users are asked to intuitively resolve a conflict in order to commit their own latest revision.

One user successfully completed the test and rated it straightforward. The other two users reported encountering more bugs with the system. More investigation will be required to resolve the bugs that users are encountering.
6.3 Survey Results

After completing all of the tests, the users had some additional survey questions to answer. First they were asked how they felt about each of the new elements on a scale from "would strongly like to change/remove it" to "liked it and it should stay the way it is". Figure 20 summarizes the results.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Would like to change/remove</th>
<th>Had problems using</th>
<th>Did not notice</th>
<th>Liked, but could be improved</th>
<th>Liked, and should remain the same</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status lights</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Commit button</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Merging with &quot;Merge&quot; button</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Merging with status markers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Merge All</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Merge Conflict</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Textboxes inside Merge Conflict</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Revision status display</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Reverting</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Figure 20: Responses in the first two columns are high indicators of areas that need to be improved upon.

From this we can see that the textboxes inside the merge conflict popup need to be redesigned. The merge conflict window itself may also need to be fixed since two out of three users had trouble using it. The merge all window could use some improvements and the individual merge buttons need to be clearer.
When asked "Colored graphics were added to the interface to provide intuitive information without labels. Which sentence best describes how you felt about them in general?", a variety of answers were received. Out of the four possible responses:

1. They were confusing, and still are.
2. They were confusing at first, but I understand what they mean now.
3. They were confusing at first, but I understand what they mean now.
4. They were intuitive and should remain the same.

all were chosen except the first. This shows that although the users were eventually able to understand their meaning, it could be very helpful to provide a legend whenever colors are used to signify different meanings.

Lastly, when asked to provide extra feedback, one user suggested that an "on-screen instruction set would be great for beginners". This would certainly solve the problem of users not noticing certain features (refer to figure 20) and is a great learnability feature to look into.

### 6.4 Clicktracking Results

After all users finished the testing processing, the clickingtracking results were aggregated. There were 9 successful commits and 0 unsuccessful commits. With such a detailed step-by-step instruction set provided to the users, this result is not too surprising. However, it is reassuring that the users always had updated references and saved projects before attempting to commit.

The merge all popup had 17 successes and 8 cancels. The merge content popup had 10 successes and 4 cancels. Although there were more successes in both cases,
the number of cancels are not what one would expect from a learnable and efficient interface. However, these results are consistent with the survey results describing the trouble the users had with the interfaces.

Lastly, it is interesting to see that the individual merging tools were hardly used at all. This may be due to a lack of need to use them during these tests. There was no reason to use them when a merge all would achieve the same task in a shorter amount of time. In order to effectively evaluate the usefulness of the individual merging tools, they must be tested during real development phases.

6.5 Conclusion

After the design and development that occurred throughout this thesis, the Zero Robotics editor has taken a great step in perfecting its collaborative tools. The lessons learned can also be applied to the development of player programs in general. A copy-modify-merge editor is preferable over a locking or real-time editor. Learnability, feedback, and efficiency are crucial principles when designing the user interface. An initial round of testing still shows some areas of possible improvement, but the overall system is far preferred over the old system. Future work involves redesigning a few features to become more intuitive to use and a larger scale test. It may be difficult for those who have never programmed or those who have never coded with a team before to jump into player programming, but with the standardized set of collaborative tools set out in this thesis, developers of all ages and experiences can more easily learn and code together.
References


