Using Tournament-Based Assignments to Motivate Students in Introductory Computer Science Courses

Travis L. Bale

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfilment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Stephen H. Edwards, Chair
Eli Tilevich
Manuel A. Pérez-Quiñonez

April 30, 2013
Blacksburg, Virginia

Keywords: Tournament-Based Assignments, Genava Framework, Education, Programming Games

Using Tournament-Based Assignments to Motivate Students in Introductory Computer Science Courses

Travis L. Bale

(ABSTRACT)

Instructors are hard pressed to create new and exciting projects to use in introductory Computer Science courses. These types of assignments not only have to teach students new concepts but also must cater to the various skill levels that are present in introductory courses. One solution that has been suggested is to used game-based assignment where students develop either a game of a AI strategy capable of playing a game. While these assignments have been shown to motivate students who are learning about programming, there are many drawbacks to using them. Most of these drawbacks come from the time commitment needed to create these assignments.

To counter these drawbacks we have created the Genava Framework, a tool which helps instructors and students to create games and strategies which are playable by the framework. Further, all games and strategies compatible with the framework can be used in tournaments in which multiple strategies can compete to see which one is most effective. This allows a tournament to be added to any game-based assignment with minimal additional effort.

To test the effectiveness of our framework, it was used to develop a tournament-based assignment for an introductory CS course. Students created strategies for the assignment which then competed in a tournament. The framework was able to handle a tournament involving 147 different strategies without error. Students also were surveyed to collect their opinions on tournament-based assignments and a majority indicated they were enjoyable and recommended their continued use.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Instructors in computer science face difficulty when they are designing assignments for introductory CS courses. These assignments serve to not only teach students about basic concepts, but to motivate and excite them about the field. These assignments give them a sense of the time commitment and the self-learning skills needed to succeed in future CS courses. Furthermore, students in introductory CS courses come from many different backgrounds. This makes it quite difficult to design a project that not only teaches a student with no experience, but also presents a challenge to more advanced students. Educators have tried many techniques and strategies to accomplish these objectives with varying degrees of success.

1.1 Game and Tournament-Based Assignments

One type of project that meets the various requirements needed by an introductory CS assignment is one that is game-based. These types of projects usually have students either implement a game or have them create an AI strategy capable of playing a game. Another interesting aspect of these assignments is the potential for incorporating competition between students. For assignments involving strategy creation, students can have their AI players compete against one another in a tournament to see who was able to develop the superior strategy. This subset of game-based assignments known as tournament-based assignments can provide an added level of competition that could be used to further motivate students to go above and beyond the basic requirements of an assignment in an attempt to develop a worthy strategy.

Game-based assignments appear to be an good choice for introductory CS courses, and studies show they are effective at increasing a student's motivation. [9, 12, 13, 14] Also, it has been shown that students prefer this type of assignment and, given the choice, will work on a game-based assignment over a more traditional assignment [10, 11]. Because of these

benefits, many instructors are creating similar types of game-based assignments for their classes.

1.2 Difficulties with Creating Game and Tournament-Based Assignments

Despite the benefits of game-based and tournament-based assignments in introductory CS courses, they do have some drawbacks. For one, these types of assignments usually involve a large time investment and require care and consideration during their development [8]. For a student to be able to develop an AI strategy for a game, the game API needs to be clearly defined. Furthermore, if a tournament is involved, time must be taken to either manually run the tournament or to create a tool capable of running the tournament. Another drawback, is that a game-based assignment may become stale if it is used repeatedly. If the same assignment is given semester after semester, then current students may stumble upon solutions that were created by previous students. Also, over time a game can be *solved*, meaning that there is a strategy that is capable of always producing a win. Connect 4, for instance, has been solved and there is an implementation of a strategy capable of always winning [7]. If students learn of these perfect strategies, they may lose motivation to develop a strategy that goes beyond the basic requirements of the project. To avoid game staleness, an instructor will periodically need to create a new game-based assignments which can be costly. An instructor may not be willing to take the time to do this and nay continue to use the old assignment, which will increase the chance of the game becoming stale.

To avoid these development drawbacks, an instructor may opt to find an existing assignment to use in his or her course. This leads to a new set of problems. An instructor must first take the time to understand and figure out how to run the components of the assignment. Also, the assignment must be analysed to ensure it is not too difficult for students to understand. Also, an existing assignment may not give much room for flexibility as it was most likely not designed with changes in mind by the original creator. With existing assignments, an instructor may be forced to take what is given to them.

These drawbacks to game-based and tournament-based assignments may deter instructors from using them despite their usefulness in teaching CS concepts. Therefore, this thesis focuses on attempting to solve the major development drawbacks of these types of assignments while making them more accessible to instructors. Further, we attempt to gain a better understanding of tournament-based assignments and to analyse the potential increased benefits of using them over a regular game-based assignment.

1.3 Overview of the Genava Framework

To combat the problems an instructor faces when attempting to create a game-based assignment, we have developed a framework that defines a core set of functionality for quickly developing tournament-based assignments. The framework contains features that allow instructors to quickly develop games where students can write AI strategies capable for automated players. Another feature of the framework is that it provides a set of common development tools that can be used when creating game and strategy code. The development tools allow a user to create games and strategies in a common way regardless of the particular details of the game. The framework also contains a tournament package which allows for a tournament between strategies to be held. Any game developed for use in the framework is automatically capable of utilizing the package and allows users to run a tournament with minimal effort. This feature makes it easy to transform a game-based assignment into a tournament-based assignment where strategies can compete against one another.

To evaluate our framework, two assignments that were tournament-based were created for use in a CS1 course. Students were given the task of developing a strategy that would be capable of participating in the tournament. A tournament was run on both assignments and our framework was capable of running a class-sized tournament involving 147 strategies without encountering any problems. Also, we evaluated student's scores and reactions to tournament-based assignments and learned that these types of assignments did not affect student's scores, testing ability, or lateness in a statistically significant way and that students reacted positivity. Students also said that other students should have the opportunity to work on these types of assignments. Analysis also showed that the open-endness of these assignments is well received and that students submit solutions with noticeable variance in lines of code needed to implement their strategies. Finally, we encountered an interesting scenario in which a minority of students abandoned their initial strategy for a game and chose to implement a simpler one that could meet the basic requirements.

1.4 Organization of this Document

In Chapter 2, we discuss the literature that influenced our framework and provided insight about existing game-based and tournament-based assignments. In Chapter 3, we discuss our framework in general, specifically, talking about our goals for the framework and the unique features it provides. In Chapter 4, we discuss some of the design aspects of our framework and how the framework implements the features introduced in Chapter 3. Also, Chapter 4 discusses some of the implementation challenges we encountered during development and our solutions to these challenges. In Chapter 5, we discuss the important components used for game and strategy development and provide a tutorial about how to develop game and strategy code for use in the framework. We also discuss, in detail, the tools within the framework users are able to utilize during game and strategy development. In Chapter 6, we

discuss how our framework and the tournament-based assignments created with it were used in an introductory computer science course and the reaction students had to tournament-based assignments. We also look at the performance of our framework and its ability to run a class-sized tournament. Finally, Chapter 7, summarizes the work performed and the improvements that can be made through future work.

Chapter 2

Literature Review

Because our framework is an attempt to provide a new method for creating game-based assignments, it is important to look over some of work that has been done. Specifically, it was important to analyse the various functionality that can be found within programming games and game-based assignments to determine what capabilities our framework should utilize. Also, it was important to model our development tools after tools known to be effective in the development of code. In this chapter we provide an overview of the relevant prior work.

2.1 Programming Games

A major goal of this project was to create a framework capable of running various types of programming games. As such it was necessary to explore some of the current types of programming games out there. These types of games can be broken into two different categories: Competitive and Educational. Although there is certainly a grey area in which a programming game can be described as both, the division of these games has been made based on their original purpose.

2.1.1 Competitive Programming Games

Competitive programming games are not a recent development. Some programming games have been around since the 1970s. In this section we noted some of the more popular games to emerge and their inspiration on our framework.

Core War

Core War is a programming game, developed in 1984, in which programmers wrote basic assembly programs, called *warriors*, that would fight for control of a virtualized memory space known as the core [3]. Each warrior is loaded into the core at a different memory address. During each turn, a memory pointer representing the warrior executes the next instruction in memory. Each warrior starts at the beginning of the assembly program that it represents. These programs contain instructions that, when executed, can cause new instructions to be placed in the core, or can affect how the warrior will move. For a warrior to win, it must cause the other warrior or warriors in the core to execute a special assembly instruction. If a warrior executes this instruction then it stops executing and is considered dead.

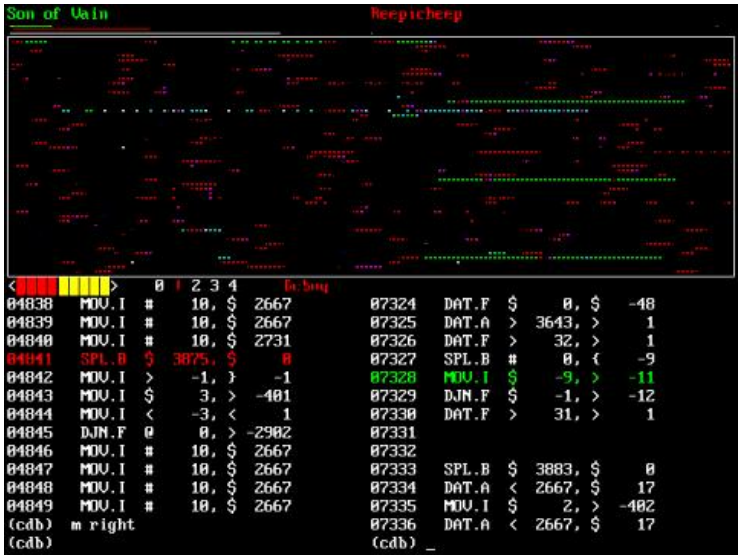

Figure 2.1: A Game of Core War [GNU General Public License]

Core War was one of the first games to use the concept of two AI programs going against each other as a game. The game became very popular amongst programmers and for a time there was an organization know as the International Core Wars Society, or the IWCS, which managed the game and approved new standards of the assemble language used to program warriors. Core War is not as popular as it once was, (The IWCS is now defunct), but nevertheless paved the road for other programming games to evolve. [4]

Robocode

Robocode is a programming game in which programmers write the strategy a robot will use to fight other robots in a battle arena [6]. Robocode provides an API which users can use

to customize their robot as well as test it in a battle. Also, robots can be uploaded to the Robocode site where they will be placed into a competition with existing bots and ranked.

One of the more impressive feats Robocode has accomplished is the ability to take user submitted source code and easily incorporate it into the Robocode framework. All users have to do is submit a zip file of their source code and the Robocode framework takes care of compilation and additional processing needed to incorporate it. This is one of the features that we wished our framework to mimic.

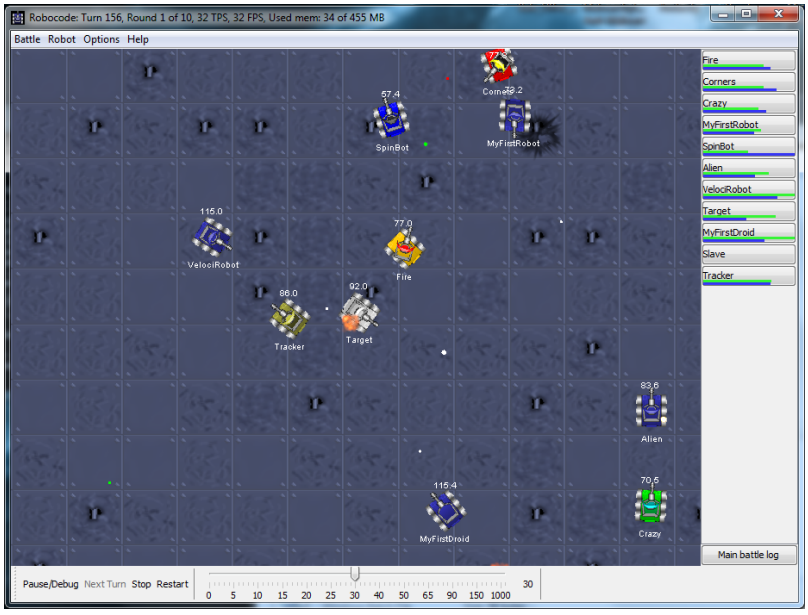

Figure 2.2: Screenshot of Robocode Battle between 11 Robots [Eclipse Public License]

Robocode has also developed into a community of programmers which work consistently on improving their robot's strategies by learning new programming techniques as well as new mathematics skills. This community is dedicated to competition, however, this does lead to problems. A newcomer may become discouraged when attempting to create a bot that will then go against bots that have been fleshed out over years of development. Also, there are many advanced guides and techniques that must be followed for a robot to even be competitive. As time progresses, a strategy could be discovered for Robocode which all strategies would have adopted to stand a chance. Once the game is solved it would be less fun to play. Our framework attempts to avoid this problem by allowing users not only to the ability to create strategies for existing games in our framework, but also create new games capable of being imported into it.

AI Challenge

AI Challenge is a competition, sponsored by Google, in which programmers were asked to use an API to develop a strategy for a game [1]. After a deadline, all submitted compete in a tournament to determine which one is the best. In past years, the games have involved moving game objects around a simplistic board. What makes these challenges unique is that users are not restricted to using one programming language. Each AI challenge was designed so that strategies could be represented easily by multiple languages. This is something that our framework could support in the future.

Figure 2.3: AI Challenge 2011: Ants Game Simulation [Eclipse Public License]

Another feature that is cool about AI challenge is that much of the code designed to run the tournament is separated from the game engine and can be reused each year. This allows for the game developers to focus more on the game and less on the tournament engine. Our framework was designed to ensure tournament code was separate from the other components so it could be used to run a tournament for any game that was created for the framework.

2.1.2 Programming Games in Education

Over time many efforts have been taken to utilize programming games for educational purposes with varying degrees of success. In this section we note some of the more successful and promising attempts at utilizing these types of game-based assignments.

Niffty Assignments: UNO

Niffty Assignments is a recurring special session at SIGSCE that discusses cool and interesting assignments that can be used in computer science courses. The UNO assignment [15], created by Stephen Davies, in particular provided many of the design ideas for our framework. In this project, students were given two weeks to create an AI strategy for the game UNO. Students developed strategies by utilizing an API that was provided to them. When the assignment was completed, the strategies competed in a tournament. Student reaction to this assignment seems to be very positive. Also this project, as Davies notes, is very popular among students and has motivated them into creating a superior strategy to win the tournament.

As mentioned, this project inspired much of the design of our framework. Our framework allows for students to develop AI strategies which can then compete against one another in a tournament. Also, the framework communicates the current state of a game to strategies in a similar manner used in this project. The main difference between the UNO project and our framework is that our framework was designed to support multiple games. In fact, the UNO game used by this assignment was converted into a game that was playable by the Genava framework.

RIT: Boardgames in Introductory Computer Science Courses

The Rochester Institute of Technology has been working with the idea of using boardgames as assignments in introductory Computer Science courses [9]. In these assignments, students create AI strategies capable of playing a board game. These strategies later participate in a *Battle Royale* in which a day is spent allowing the students to see their strategies competing against one another in a vast tournament. Currently, they have three different game engines they use, but have talked about allowing students to create the game engine that next year's students will use.

One thing that RIT is trying to analyse is whether game development or strategy development is more effective in motivating and teaching students about Computer Science. They feel that game development is weaker to strategy development and have focused their efforts on this side. Our framework has chosen to embrace both ideals by allowing easy development of both games and strategies to gain the benefits that both forms of development provide to students.

Another motivation of RIT's project is to see how positively students reacted to strategy development. Their preliminary findings have shown that students react very positively to these types of projects and were interested in continued study of Computer Science after completing them. We wanted to see if our framework would also received positive feedback from students.

One interesting restriction that RIT put on their project was that the games used had to be board games. They used board games because they tend to be discrete. Their claim is that the discrete model of boardgames fits better with basic CS concepts, such as sorting and data structure management. In our framework, game development is currently restricted to turn-based games as focus was made to support boardgames first due to the findings of RIT. However, it should be noted that an eventual goal of the framework would be to support many types of games.

2.2 Android Development Influences

Not all design influence for the Genava framework came from programming games. Some of the tools created to aid in strategy and game creation were influenced by Android Application Development using Eclipse. [2] A plugin for Eclipse, ADT, provides a user with many features that can be used to easy the development of an Android application.

When a developer runs an Android application using Eclipse, a device emulator pops up and allows the developer to see how their application runs on a device. Our framework contains development tools that are tied to strategy and game creation. When work on a Genava framework project these development tools allow a user to run their strategy or game and perform debugging operations upon it.

Android development also makes use of a logging feature called logcat. As a Android application is running on the emulator, the log of the device is displayed in Eclipse to the user. Our framework also provides a basic logging utility that users can use to debug their strategies.

ADT also provides a feature where a user can download and mange new versions of the Android API. This allows a user to keep up-to-date on all the latest features of Android. We also envisioned a system like this that could be used to manage games for which the user can develop strategies. Currently, this feature is not in the framework, but there are some basic forms of game and strategy management.

Chapter 3

The Genava Framework

The Genava Framework is a tool that can be used to create programming assignments in the form of games and tournaments. Users can program games for which other users can write strategies for. The framework was designed with the idea that it should support multiple games and eventually should be able to support a wide variety of games. Strategies developed for use in the framework can compete in a tournament-style simulation to see which strategy is able to perform the best. Also, the framework contains a tool to help convert game and strategy code into usable components of the framework. Other tools exist to simulate individual games as well as full tournaments. These tools can help a user to debug game and strategy code and can also be used to assess the performance of a strategy. The driving force behind these features is to use the framework as an educational tool capable of helping instructors to create interesting game-based programming projects.

Figure 3.1: Overview of Using the Framework for Creating a Tournament-Based Assignment

3.1 Multiple Games

The Genava Framework differs from other strategy programming frameworks, such as Robocode [6], in that it is designed to run more than just one type of game. Robocode only allows users to create strategies for robots to battle in an arena. The Genava Framework, on the other hand, is capable of running several types of turn-based games. Another difference is that users are able to create the games that other users can make strategies for. Users can easily add games created by other users into their local copy of the framework and begin developing strategies for playing them. Not only does this provide a variety in the types of games users can create strategies for, but it also gives them a common set of tools for creating, running, and executing strategy code. Users experienced with creating strategies for one game should have no problem with creating a strategy for another game. This gives users the flexibility to develop strategies for any game in the framework and at the same time allows them to create a game if the existing pool of games is not to their liking.

3.2 Tournaments

One of the major features of the Genava Framework is the ability to run tournaments with a given set of strategies. These tournaments are used to provide competition between users who are programming game strategies. All tournaments in the framework are played round-robin style. In this style of tournament, a selected number of strategies go into a round. The strategies compete and accumulate points during the round. After a round has completed, each strategy holds on to the points it has gained during that round. Each strategy continues being entered into rounds until it has competed against every other strategy. The winner of the tournament is the strategy that has gained the most points over each round. One thing that differs in framework tournaments from a traditional round-robin tournament is that multiple games are played during a round. This achieves the goal of enabling the user to assess the true skill of a strategy. Since many of the games that can be played in the framework are based on chance, multiple games are necessary to ensure an inferior strategy does not win simply due to a lucky game.

This tournament style of play was chosen to eliminate several problems that occur with other traditional match-ups such as bracketed tournaments. For example, bracketed tournaments can knockout good strategies early on and allow bad ones through to later rounds. If the two best strategies played against each other in the first round, then one would be knocked out early. This could allow a lesser developed strategy to do better in the tournament which defeats the purpose of having students developing the best strategies possible in order to win.

Another reason that this tournament style of play was chosen is that it gives more consistent results. If the same tournament using the same strategies is run multiple times, the results should be fairly close to previous tournament results. It should be noted that this would be consistently true if enough games are played to account for a strategy having a lucky game. Therefore, it is important for the person measuring the results to be aware of this factor as well as ensure that enough games are run to mitigate this possibility. It is anticipated that there would be enough students competing in the tournament that the chances of this would be low; however, it is a factor to be aware of.

Another reason tournament style of play was chosen is that all rounds of a round-robin tournament can be played at the same time. This is because each round does not depend on the results of a previous round as is the case with bracketed tournaments. This allows multiple users to play the games at the same time which meets a goal of developing a system for a learning environment where multiple students can be engaged in the process simultaneously. It should be noted that the Genava Framework currently does not take advantage of this, but could eventually be changed in order to run a tournament in a parallel or a distributed manner. This should be an additional goal for a future project from a continual process improvement standpoint.

A final necessary feature incorporated into the tournament style of play in order to eliminate

potential problems is the ability to disqualify a strategy. This occurs whenever strategy code makes an illegal move, throws an exception, or goes into an infinite loop. When the framework disqualifies a strategy from a tournament it receives a score of zero and it is not scheduled to participate in future rounds of the tournament. Also, the scores of each successful strategy are modified to remove any points they received in rounds where they were competing against a disqualified strategy.

3.3 Development Tools

The Genava framework not only provides the means to run games and strategies, it also has tools to aid in their development. Any game or strategy project that imports the framework gains access to these tools automatically. When the project a developer is working on is ran, the framework is automatically booted up and provides access to these development tools. Most of the tools provided are used to help test game and strategy code developed by the user. Also, there are tools to help integrate the Java source code written by game and strategy developers into executable components of the framework.

In total there are four different tools to help aid users in development.

- The **Strategy Export Tool** is used to convert strategy code into a special file that can be used by the framework to execute the strategy.
- The **Game Export Tool** is used to convert game code into a special file that can be used by the framework to execute the game.
- The **Game Simulation Tool** is used to run a game. The user also specifies which strategies will participate in the game. This tool also provides game visualization so a user can see how the game and strategies are performing.
- The **Tournament Simulation Tool** is used to run an entire tournament in the same round-robin style as described in the previous section. This shows a user how well a strategy competes over a long set of games.

These tools are described in greater detail in Chapter 5.

3.4 Educational Tool

Most of the features of the Genava Framework were designed with the goal of making it an educational tool which could be used to teach programming concepts. The idea is that instructors can create assignments similar to Davies's Uno project [15], but can change the

game being used for the assignment. Because the framework allows for game development as well as strategy development, instructors can make games for the framework which could be used for an assignment. As long as the game can be modeled to work within the framework, instructors are free to create different games to emphasize different programming concepts. Many possibilities open up with this model. Instructors could trade games they have used in different classes, or even have advanced students create games as an assignment for a class.

Another feature found in the framework that enables it to be used as an educational tool is the previously discussed tournaments. The idea is that students can compete in a tournament with their fellow students in the class. By having a competition, students can become motivated and excited about working on a project. Also, students today come from an era where they grew up with computer games. They tend to take gaming competition seriously and through a tournament style of play, would put more effort into creating strategies as opposed to possibly doing the minimal amount of work needed to complete a typical project or assignment. This is evidenced by the success Professor Davies has had with his Uno project.

The testing and debugging features of the framework provide students with a visual way of seeing their code execute. By seeing what moves their strategies make in a game, students may be able to tell if their strategy performs incorrectly. Also, seeing results of their code execution in this manner may lead to easier debugging than simply looking at the text output of a log. Another good thing about the debugging tools is that they are embedded into the Genava Framework. Because of this, the tools will stay consistent regardless of what game a student is developing a strategy for. Students only have to learn one set of debugging tools that can be used when writing a strategy for any game compatible with the framework.

Chapter 4

Framework Design and Implementation

To provide all of the functionality described in Chapter 3, the Genava Framework was designed to support all features appropriately. To separate large portions of the functionality, the framework was divided into five packages. The framework also makes extensive use of dynamic loading to mange and run game and strategy code. Also, careful consideration was made to ensure that each feature of the framework performs in a correct and timely manner. The overall design of the framework was also shaped by challenges that arose during development of the project, such as retrieving moves from a strategy and disqualifying strategies that make illegal moves.

4.1 Framework Packages

The `jarfile` package works with external game and strategy JAR files and dynamically loads them into framework. It also manages all files loaded into the framework and provides classes which allow other packages in the framework to interact with them. Only games and strategies loaded by the `jarfile` package are recognized by the framework.

The `game` package handles the simulation of individual games. The game package contains a class called `GameSimulator` which takes in games and strategies loaded by the `jarfile` package. The simulator serves as the mechanism games use to communicate state and retrieve moves from strategies. The game package also includes several interfaces and abstract classes which are used by developers to create games and strategies capable of being used in the framework. Games, for example, must extend the `GenavaGame` class and provide implementations of `GenavaMove`, `GenavaGameState`, and `GenavaPlayerState` to be recognized by the framework.

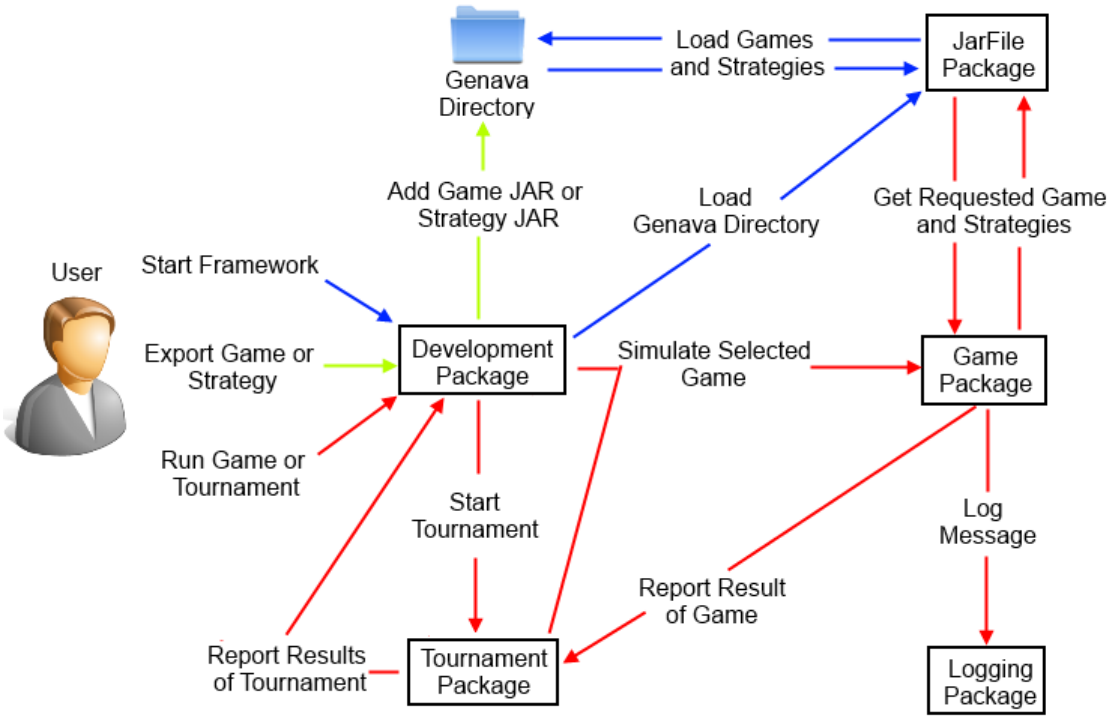

Figure 4.1: Genava Framework Packages and Interactions

The `tournament` package handles the simulation of a round-robin tournament between strategies. The `Tournament` class manages the strategies currently competing in a tournament and the scheduling of rounds. The `Round` class handles an individual round of the tournament and uses a `GameSimulator` to simulate every game in the round. After a round has completed, the scores of each player are reported to the `ScoreTracker` class. After a tournament is over, the results are retrieved from the `ScoreTracker` class.

The `logging` package handles all logging within the framework. Both games and strategies have these logging capabilities built into them. Also any component of the framework can use the log to display information about how it is currently functioning. This package contains a simple interface which can be implemented to provide logging in different formats. Currently, there are three different implementations of this interface which all print to different components. Logging was placed into a separate package because each package may need to use it.

The `development` package contains all the GUI code used by the testing and debugging tools.

This package currently serves as the frontend of the framework and provides a visual interface to all other packages of the framework. Through the tools provided by the development package, users are able to create and simulate game and strategy code as well as create tournaments.

4.2 Dynamic Loading

To allow both games and strategies to be recognized by the framework, Java reflection is used to dynamically load game and strategy files onto the Java classpath. Both game and strategy files are created using the export tool. This tool takes Java source code and transforms it into a specialized JAR file. These JAR files have additional fields located within their manifest files that allow them to operate with the framework.

Two classes within the framework are used to dynamically load each type of file. The `GameJarFile` class loads games, while a similar `StrategyJarFile` class loads strategies. Both of these classes attempt to load files into the framework based on the file path they are given. If any manifest entries are missing or if the file is not the correct type, a flag is set to indicate that the file is not the correct type and that no operations can be performed on it.

Once loaded, instances of the game or strategy can be created from the file. The framework contains interfaces that both games and strategies must adhere to. These interfaces are used by the framework to control execution of game and strategy code.

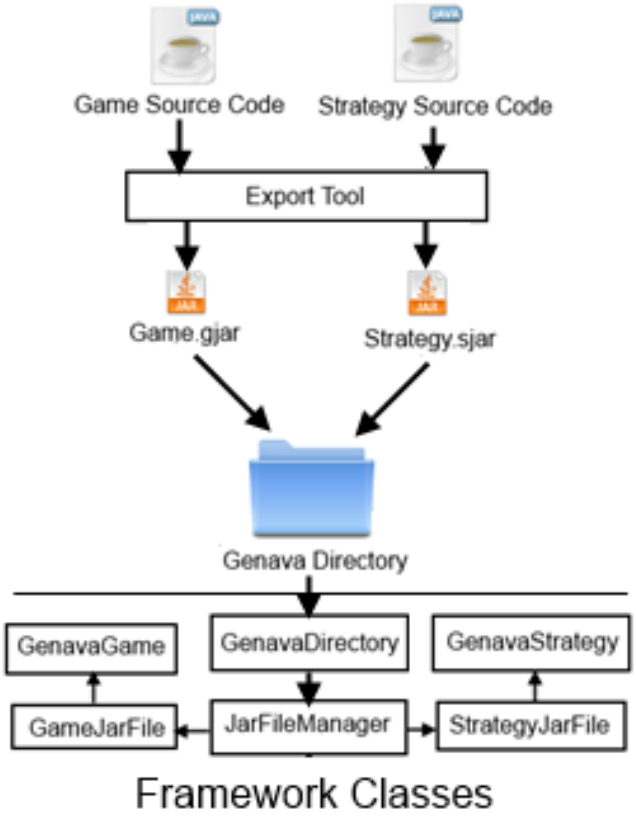

Figure 4.2: Dynamic Loading Process of the Genava Framework

Due to the large amount of game and strategy files that can be loaded into the framework, some file management was implemented within the framework. The framework is capable of opening a special directory known as a Genava directory. Genava directories store game and strategy files. When the framework opens one of these directories, every file within them is loaded into the framework. Two subdirectories within a Genava directory are used to split up game and strategy files. The export tool places strategy and game files within the current Genava directory that is being used by the framework.

A benefit of allowing the user to specify a directory containing games and strategies is load time. If this directory were centralized, then all games and strategies would end up being loaded into the framework. If too many files where loaded by the system, it could slow down the frameworks responsiveness. Also, the loading of files could cause the JVM to run out of memory. By allowing users to specify the Genava directory, they can control which games and strategies are loaded.

Also, within the framework, loaded game and strategy files are managed by a class known as the `JarFileManager`. This class pairs game files with the strategies capable of playing them and provides a mechanism for retrieving the game and strategy files it manages. This class

also handles the mass loading of files into the framework. When the framework is started, the manager loads all game and strategy files found within the current Genava directory into the framework.

4.3 Implementation Challenges

During development of the framework, there were several implementation challenges. Highlighted below are some of the more interesting challenges and their solutions.

4.3.1 Retrieving Moves

One of the most difficult problems to solve was how to actually retrieve moves from a strategy. Each strategy contains a method used to retrieve the strategy's next move. Simplistically, this method could have been called directly by the framework to retrieve the move. This approach would cause several problems however. First, this method may not actually return when called. If a strategy goes into an infinite loop, then the framework would be left waiting for a move to complete. This strategy could also raise an exception during its move which would in turn crash the framework. The occurrence of both of these scenarios are extremely likely given that the Genava framework's main purpose is to be used as an educational tool for beginner students. Also, if an error like this were to occur, the tournament would have to be run again which is time consuming. Therefore, it was important to develop a better system for retrieving moves in order to eliminate the potential for these problems.

The first method considered to correct this problem was a process based solution. The idea was that each strategy would run in its own process. An interface was created to talk with the process to retrieve moves. If a move was not received within a certain amount of time the process would be destroyed and catastrophic errors would not harm the process running the framework. This solution, while good for individual games, did not scale well in tournaments. Process creation took too much time to run a tournament quickly enough. This was mainly due to the fact that the strategy processes had to be recreated every time a tournament round changed because it would not have been feasible to have a running process for every strategy in the tournament.

The second method considered uses a thread and latch based approach to safely retrieve moves. This method, which was ultimately the selected solution, is the one currently used in the framework. A class called `MoveGetter` serves as a buffer between the framework and a strategy. `MoveGetter` uses two different methods to achieve safe move retrieval: `getNextMove()` and `run()`. When a `MoveGetter` is created, a thread is started that executes the `run()` method. Immediately, this method begins to wait for the current player to change. This is done by using a countdown latch. Once the current player changes the `run()` method continues and attempts to get a move from the player. If the player throws an exception it

is caught and the move is considered a null value; otherwise, the move is set to the value returned by the player. Once a move has been retrieved, another latch is triggered to notify that a move has been made. At this point the `run()` method, again, begins to wait for the player to change.

The `getNextMove()` method is called by the framework to get a move from the player passed into it. The player is set as the current player and a countdown latch called `playerChanged` is triggered. At this point the method waits for a move to be set by `run()`. This wait has a five second timeout. If a strategy takes longer then this timeout to make a move, then the framework assumes the strategy has stopped working in some way and returns null as the next move for that player.

This method of move getting has many advantages over the process method. Only one thread needs to be used and never needs to be recreated. The `MoveGetter` was designed to be created once and used continuously by every strategy. If a strategy does not supply a move within the timeout the `MoveGetter` interrupts the thread and is recreated by the framework. This allows threads to be cleaned up by garbage collection over time. Also this method runs significantly faster than the process method.

4.3.2 Handling Illegal Moves

Another major problem that could occur within the framework is a strategy that makes illegal moves. If not handled properly, this could cause the framework to crash and require a tournament to be run again. It is obvious that illegal moves need to be prevented. This lead to two major questions: The first being, what should happen to strategies that make illegal moves? And secondly, who is responsible for handling illegal moves?

As mentioned earlier in Chapter 3, a strategy that makes an illegal move or fails to make a move is disqualified immediately from a tournament. This is done for many reasons. For one, a strategy that has already made an illegal move is bound to make another simply because the implementation has already proven itself to be error prone. Getting rid of the strategy after one illegal move prevents the possibility of more occurring during the tournament. Another reason is that a serious implementation error in a strategy could slow down an entire tournament. If a strategy goes into an infinite loop, many things have to be done to clean it up. First, the `MoveGetter` must be reset which would cause a new thread to be created. Secondly, a new instance of the strategy must be created, because the old instance is most likely in an inconsistent state. Finally, the new instance must then be placed back into the tournament. This process takes time and if a strategy caused this to happen in every game, it would take a very long time to complete. Therefore, disqualification of error prone strategies caused by illegal moves is the best method to ensuring tournaments complete in a reasonable amount of time.

Once the methodology for handling illegal moves was resolved, the second question of re-

sponsibility for handling illegal moves had to be answered. The framework isn't capable of analysing moves based on the rules of a game; and the game is unable to disqualify strategies. Therefore, a hybrid solution was developed to resolve the responsibility issue. When a move is retrieved from a strategy, the framework sends the move to a method in the game that checks the validity of the move given the current state of the game. This method is implemented by the developers of the game and the framework trusts that it is exhaustive in its testing of a move. If the method determines the move to be legal, the framework returns the move to the game. If the move is illegal, the framework disqualifies the strategy which made the move. Additionally, a flag in the move object is set to indicate that the move is invalid. Once a game receives a move it must check to see if this flag has been set. If it has, the game should not process the move and declare the game to be over by exiting. The hybrid solution requires the human element of the developers of the game while also depending on the framework to disqualify players.

4.3.3 Adjusting Scores

As mentioned earlier, strategies that make an illegal move or crash are disqualified and removed from the tournament. However, this creates a problem in scoring. Because a tournament in the framework is played round-robin style, every strategy competes against every other strategy. If a strategy is disqualified in the middle of a tournament, then there are some strategies that have played against the disqualified strategy and some that have not. This creates a problem when computing score. Some strategies were able to play more games than others, because they were able to play against the disqualified strategy before it was removed from the tournament. By playing more games, these strategies have gained an unfair advantage in scoring.

To fix this problem, scores are adjusted at the end of a tournament to remove any points a strategy gained when facing a later disqualified strategy. A temporary file is used to hold the scores each strategy received during a round in the tournament. After the tournament, a list of all disqualified strategies is created and the round data of the temporary file is analysed. If a round is found to contain a disqualified strategy, all scores in that round are deducted from the total scores of each strategy that participated. This will leave all disqualified strategies with a final score of zero. Also, this will make it appear as though no strategy ever played a disqualified strategy in the tournament.

Chapter 5

Game and Strategy Creation

The entire purpose of the framework is to allow instructors to create games and allos students to create corresponding strategies. This chapter documents the process of creating game and strategies that are capable of interfacing with the framework. It describes in detail every step needed to make a working game and strategy and the tools required to do so. Also, tips for game development are given as well as requirements necessary to ensure a game or strategy can run successfully within the framework. Finally, a more detailed explanation of the development tools introduced in Chapter 3 is given.

To illustrate the development process for games and strategies, an example game (Battleship!) is used. In Battleship!, players take turns firing at each other's boards in an attempt to sink their opponent's ships. The first player to sink all of the opponent's ships wins. This game shows off many of the features found within the framework.

5.1 Setting Up the Development Environment

Before games and strategies can be created, the environment in which they are developed needs to be setup. All games and strategies are coded in the Java language and as such a Java IDE is needed. Although any IDE could be chosen, the Genava framework has been most tested with Eclipse. Also, some of the tools in the framework rely on how Eclipse organizes Java projects. To begin either game or strategy development, a new Java project needs to be made within an Eclipse workspace. Next, the framework needs to be added to the project. This is so games and strategies can use the testing and debugging tools from within Eclipse. This is done by adding a JAR file containing the framework onto the build path of the project. At this point game development can begin, however strategy development cannot.

To complete the setup for strategy development, a game needs to be selected that the strategy

will be able to play. After the game is selected, a JAR file containing the game code needs to be added to the strategy project's build path. For example, if a users wishes to make a strategy for Battleship!, they must add a JAR file containing the Battleship! game code to the build path. If this JAR file was not included, Eclipse would see any game specific classes used by the strategy as being undefined. Eclipse would mark these as syntax errors and prevent the strategy from being built. By adding the game code to the build path, Eclipse will be able to resolve these class names. After adding the game code to the build path, the environment is ready for strategy development.

One thing that should be noted is that the development environment is only setup for the Java project and not for Eclipse. To create more games and strategies, new Java projects need to be created for each game and strategy. Also, each new project must go through the setup described above.

5.2 Game and Strategy Implementation

After setting up the development environment, game or strategy development can begin. From here it is up to developer to determine what kind of game or strategy they wish to develop. Within their Java project, a developer can create as many different classes as they need to create their game or strategy. However, there are a few things required by both games and strategies. When developing a game for the framework it is important to note that only games that are turn-based, only one player is moving at a time, are supported. Therefore, any game can be developed for the framework as long as it can be expressed in this manner. Most of the other requirements require the developer to extend the functionality of abstract classes and interfaces found within the framework. This will allow their code to be executed, once it has been loaded into the framework. Within this section descriptions of all notable classes and interfaces are provided as well as usage techniques.

5.2.1 `GenavaGame` Class

Any game project needs to have one class that extends the `GenavaGame` class. The Battleship! game, for instance, has a class called `BattleshipGame` which extends this class. The `GenavaGame` class contains six abstract methods that are called by the framework at various points during game simulation. These methods define different behaviors such as setting up new games and checking the validity of a strategy supplied move. It is up to the game developer to determine the implementation of these methods.

The six abstract methods are `initialize()`, `newGame()`, `simulateTurn()`, `simulateEntireGame()`, `isMoveValid()`, and `setupGamePanel()`.

- The `initialize()` method is called when an instance of the game is created by the

framework. Here a game developer should create all persistent variables the game relies on. The Battleship! game uses this method to set initial values for its fields. The `newGame()` method is called by the framework when a new game should be started. At this time, the game should forget all state of the previous game and reset itself. The Battleship! game uses this method to reset its fields and creates new `PlayerState` objects.

- The `simulateTurn()` method simulates a single turn of game play. Here a game developer should define the actions a player is allowed to take in one turn. Also, this is where a game should attempt to get a move from the current player's strategy. In this method, Battleship defines a single turn as both player's making a move. There are two types of turn defined in the Battleship game. First, players must place their ships on the board. After this is done, players then take turns shooting at each other's board. The `simulateTurn()` method defines both of these turn types.

- The `simulateGame()` method simulates an entire game. The developer should create a loop here to call `simulateTurn()` until a strategy has won or been disqualified from the game. Battleship! repeatedly calls `simulateTurn()` until a strategy is able to sink all of their opponent's ships.

- The `isMoveValid()` method is called by the framework whenever it receives a move from a strategy. The move passed into this method needs to be analyzed thoroughly to determine if the strategy is able to make the move it provided. If so this method should return true, else it should return false. Battleship! uses this to check if a player's move is a possible ship placement, where all ships must be on the board and not intersecting. Also, it checks to see if a player's shot will land on the opponent's board and hits a space that was not previously attacked.

- The `setupGamePanel()` method is called by the framework if a visual representation of the game is needed. This is mainly used by the game simulation tool to visual show a user the game the simulation is playing. This method should return a class which extends the `GenavaPanel` class. The `BattleshipPanel`, of the Battleship! game, extends `GenavaPanel` to provide a visualization of both player's boards during game simulation.

5.2.2 `GenavaStrategy` Class

Similar to game projects, strategy projects must also have a class which extends an abstract class. However, in this case, it is the `GenavaStrategy` class. `GenavaStrategy` contains three methods which are called during various points of game simulation. These methods are `initialize()`, `newGame()`, and `nextMove()`. To help better understand the usage of these methods, a simple strategy capable of playing Battleship! is described. This strategy, called

`BruteforceStrategy`, will fire at every position on the board in order until all ships are sunk.

- The `initialize()` method operates in the same way as a game. It is called by the framework when an instance of the strategy is created and should setup persistent variables the strategy relies on. The `BruteforceStrategy` uses this method to set an `x` and `y` value which describes the position the strategy will fire on next.
- The `newGame()` method is called by the framework whenever the strategy is about to participate in a new game. This method exists as a way for a strategy to reset itself between games. Strategies need to make sure that they use this method to reset any values they are holding that might affect move decision in the new game. If not done correctly, a strategy may use previous game state during move decision which in turn could cause a player to make bad or illegal moves. The `BruteforceStrategy` uses this method to reset the `x` and `y` values back to their initial values. If this was not done, the strategy would fire at the wrong position when the new game started.
- The `nextMove()` method is called by the framework whenever the game requests a move from this strategy. Here is where a strategy developer should return a move to the game and in effect determine how their strategy will play the game. For Battleship! there are two types of moves that can be returned. The first is a ship placement move which describes the positions of every ship on the strategy's board. The `BruteforceStrategy` uses a fixed placement for each game where the ships occupy the first five rows and are all oriented horizontally. The second is a call shot move which specifies a position to attack on the opponent's board. The `BruteforceStrategy` uses its current `x` and `y` value as the shot. It then increments `y` by one. This is to setup the next shot for the next turn to be one space to the right. If `y` equals ten, then `x` is incremented by one and `y` is set to zero. This is so that the next position fired will be at the start of the next row on the board.

5.2.3 `GenavaGameState` Class

For a strategy to be able to determine how it should move, it needs to be able to get information about the current state of the game. This is accomplished with a class know as `GenavaGameState`. An instance of this class is passed from a game to a strategy when a game requests a move from a strategy. The game developer is responsible for defining what state will be delivered by this class. To do that, a game developer extends the `GenavaGameState` class and adds methods and variables that can be used by a strategy to determine its next move. Battleship! uses a class called `GameState`, which extends `GenavaGameState`, to send a strategy both its and its opponent's board. Only code found in the extended `GenavaGameState` will be visible to the strategy as it is the only parameter passed into the `nextMove()` method. All strategy decisions are made by analyzing the information found in

the game state. A Battleship! strategy can use the opponent's board to determine if its last shot was a hit or a miss.

A game developer must be careful as to what objects are made available to the strategy through a game state object. Any methods that are able to affect the game should not be accessible to a strategy. For example, if the `GameState` object of Battleship! gave the player access to the Board classes `applyMove()` method. A strategy could keep using this method to make additional moves on its turn.

To prevent strategies from gaining functionality that should only be available to a game, a game developer should analyze the classes made available by the `GameState` to see if these problems occur. Any classes that should not be accessed by a strategy should be package protected to prevent strategies from creating them. Also, any classes given to the strategy by `GenavaGameState` should ensure that all methods that change the game in some way are either private or package protected. For example, Battleship! gives a strategy its opponent's board. The Board class contains a method to apply moves to it. This method is declared as package protected so that other classes within the Battleship! game can apply moves, but strategies will not have access to them.

5.2.4 `GenavaMove` Class

For a strategy to be able to communicate its next move back to a game it must create a move object that is a subclass of `GenavaMove`. It is up to the game to define what types of move objects it will accept. A game can define as many subclasses of `GenavaMove` that it needs to describe every type of move within the game. It should be noted that the return type of the `nextMove()` method within a strategy is a generic type that is specified by the strategy class. Because a strategy is only able to set this to one type of move to return, a game developer should make a class which can then be extended to accommodate additional types of moves. As mentioned, the Battleship! game has two different types moves. One is used to place ships and the other to fire at an opponent's board. Both of these move types are described as subclasses of Move which is a subclass of `GenavaMove`. A strategy playing Battleship can set its `nextMove()` method to return types of Move. This gets over the limitation of `nextMove()` being able to return only one type of move.

5.2.5 `GenavaPlayerState` Class

To allow games to keep track of player specific state, a game developer needs to create a subclass of the `GenavaPlayerState` class. This class is responsible for holding game state that is only relevant to one of players in the game. The Battleship! game uses this class to hold the board that is associated with a given player. The `GenavaPlayerState` class also allows a game to award points to a strategy during the game. A game simply calls

the `addPoints()` method of the `GenavaPlayerState` class to update the score of the given player. Battleship! gives a player 1 point for winning a game.

5.2.6 `GenavaPanel` Class

The `GenavaPanel` class is a special class that is only used in a game project. A game project should extend this class if it wishes to provide a visual representation of the game to the user via the game simulation tool. A `GenavaPanel` is simply an extension of the `JPanel` class. A game developer can override the `paintComponenent()` method to draw a graphical representation of the game. An instance of the class that extends `GenavaPanel` should be returned by the `setupGamePanel()` method that is provided by the `GenavaGame` class. Adding a visual representation of the game is completely optional as a tournament does not display a `GenavaPanel`, however, it is a good idea to as it allows a strategy developer to visually debug their strategy using the game simulation tool. As mentioned, Battleship! displays both strategy's boards and ship placements as well as the shot locations.

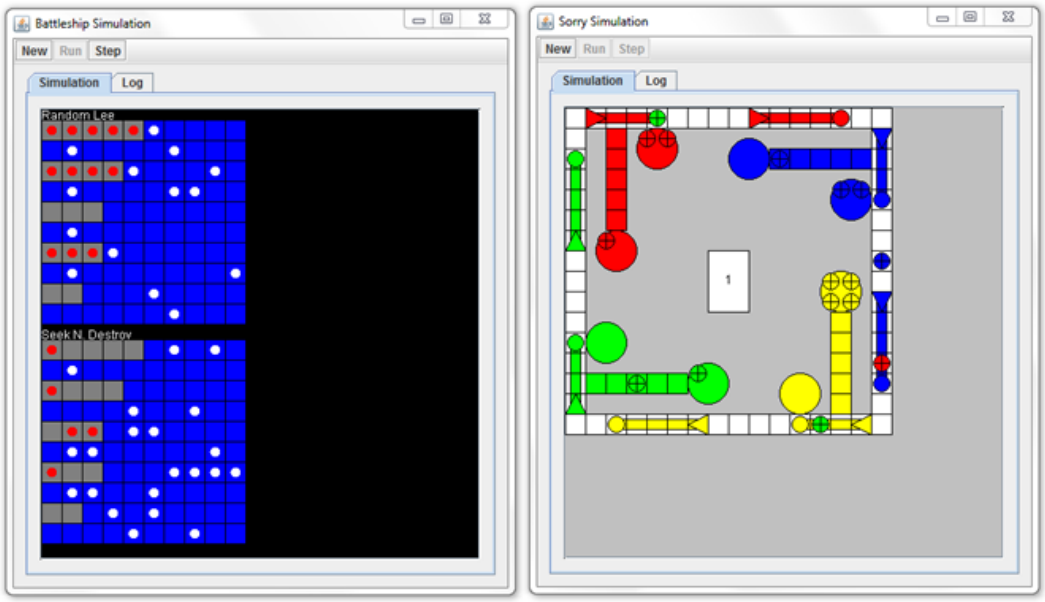

Figure 5.1: Game Simulation for Battleship! and Sorry!

5.2.7 Logger

To create debug messages that are visible within the debugging tools, developers can use the logger. Both the `GenavaGame` and `GenavaStrategy` classes contain a reference to the current logging mechanism being used by the framework. A developer can call different

logging methods using this reference. Currently, there are only two methods within a logger object which both print messages. The only difference is that one can print a message with an id while the other cannot.

One of the main places the log is used is in the game simulation tool. The tool contains a section where users can see any messages that are being logged using the logger. The framework, the games, and the strategies can all make use of logging functions to let the user know about different events currently happening within game simulation.

5.3 Exporting, Testing and Debugging

Once a game or strategy has been implemented it needs to be tested for correctness. As mentioned earlier, the framework provides many tools used to accomplish this task. When a developer runs their project in Eclipse they will a dialog which asks them to specify the Genava directory they wish to work with. Once selected, the directory will be loaded into the framework for use and the user will be shown the Development Form which provides access to all of the development tools provided by the framework.

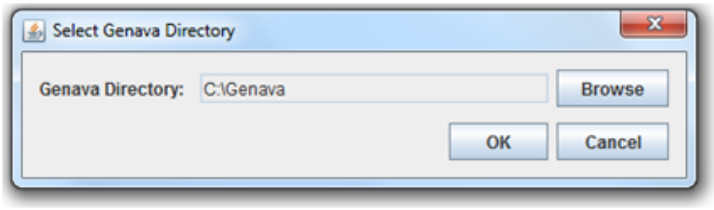

Figure 5.2: Genava Directory Selection

Figure 5.3: Development Form

5.3.1 Export Strategy and Game Tools

Before a developer can begin testing their strategy or game, it must be converted into a form that the framework can recognize. The export tool is responsible for preparing either a game or strategy for use in the framework. This tool takes the compiled Java source files created by Eclipse and creates a either a GJAR file for games or an SJAR file for strategies. Both GJAR and SJAR files use a modified manifest file containing additional fields used by the framework. The export tool is used to specify these additional fields. Games and strategies have different values needed for their manifest and as such have slightly different displays.

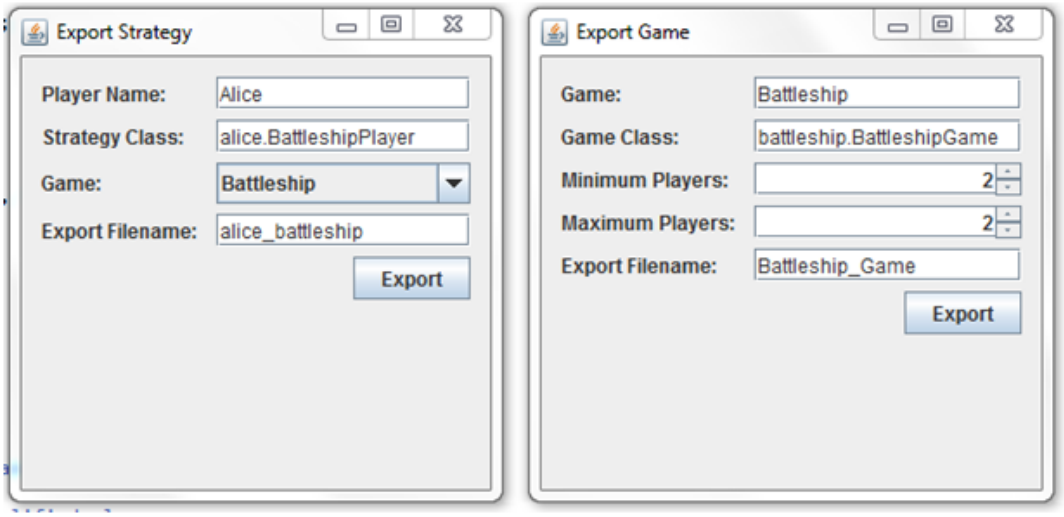

Figure 5.4: Export Strategy and Game Tools

To export a game or strategy, all fields are required. Below describes each field and what their valid values are.

Game Export Tool Fields

- **Game** is used by the framework to identify the game. No two games should have the same value. If they do, then only the first one that is loaded will be recognized by the framework. Battleship! uses its name as its game name.
- **Game Class** is used to specify which class within the project extends the `GenavaGame` class. The framework uses this field to create an instance of the game. The `battleship.BattleshipGa` class extends the `GenavaGame` class.
- **Minimum Players** is used to specify the minimum number of players need to play the game. Battleship! is a two player game so this value is set to 2.
- **Maximum Players** is used to specify the maximum number of players that can play the game. As mentioned, Battleship is two players so this value is also 2.

- **Export Filename** is used to specify the name the GJAR will have.

Strategy Export Tool Fields

- **Player Name** is used to specify the name that the framework will use to identify the strategy. The testing and debugging tools use the player name as the identifier for the underlying strategy.
- **Strategy Class** is used to specify which class within the project extends the `GenavaStrategy` class. The framework uses this field to create an instance of the strategy.
- **Game** is used to specify the game this strategy is meant to play. This field contains a list of games currently loaded into the framework.
- **Export Filename** is used to specify the name the SJAR will have.

After all fields have been entered, a developer can hit export to create the GJAR or SJAR file for their project. This file is stored within the current Genava directory that was specified. Also, after the project is exported it is immediately loaded into the framework and can begin being used by the other tools. If at any time a developer changes their strategy or game, then they must remember to export their project again so that the changes will be reflected. Failure to do so will result in the framework using older versions of the strategy or game.

5.3.2 Game Simulation Tool

The Game Simulation Tool allows developers to simulate a game that is currently loaded in the framework. When the tool is launched, users are shown a screen which allows them to select which game and strategies will be part of the simulation.

After specify the settings, users are shown a new window containing the simulation. Users can use this form to start new games, run an entire game, or step through a game one move at a time. If a game extends the `GenavaPanel` class, then it is displayed as part of the simulation. Also, developers can see the messages their games or strategies are creating.

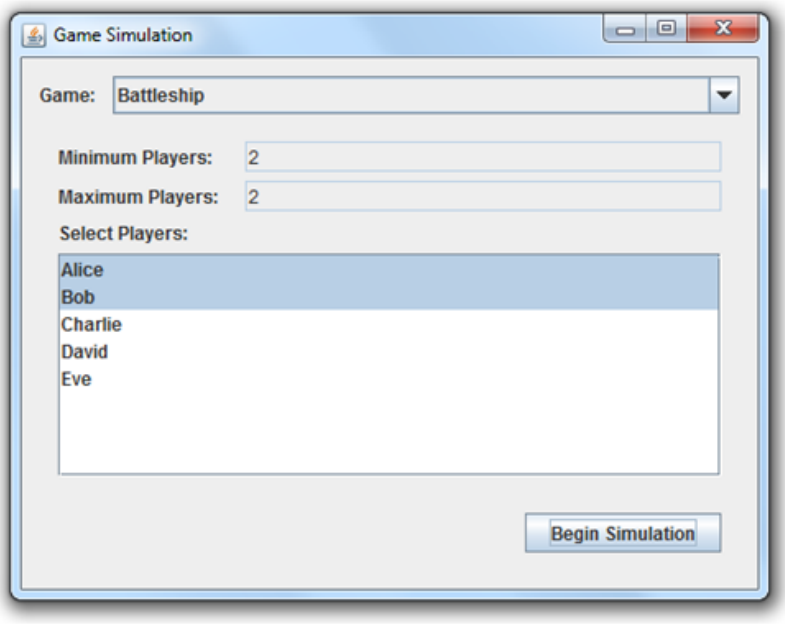

Figure 5.5: Game Simulation Setup

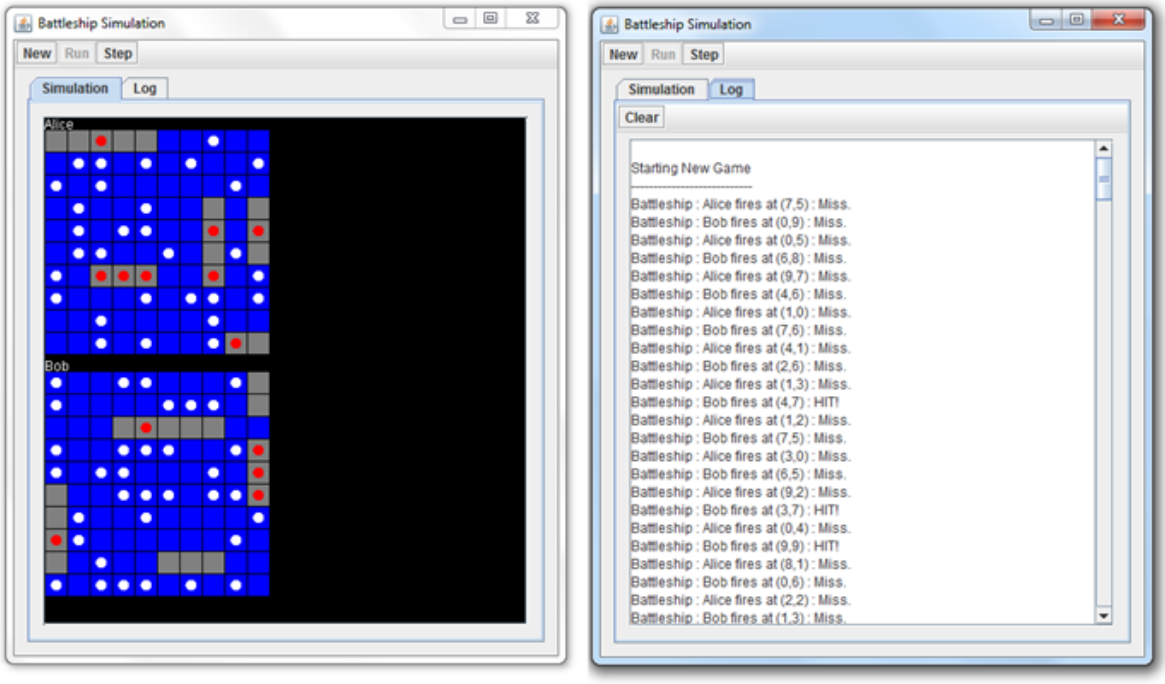

Figure 5.6: Game Simulation and Log

5.3.3 Tournament Simulation Tool

The Tournament Simulation tool allows strategy developers to simulate a tournament between multiple strategies. Users can specify the game, the number of players per game, the number of games that will be played per round, and the strategies participating in the tournament.

Once values for all the fields have been specified, the user is able to run the tournament. The results of the tournament are displayed once the tournament has completed. Using these results, developers can see how well their strategy performed. This knowledge can help developers in determining whether their strategy is good or bad.

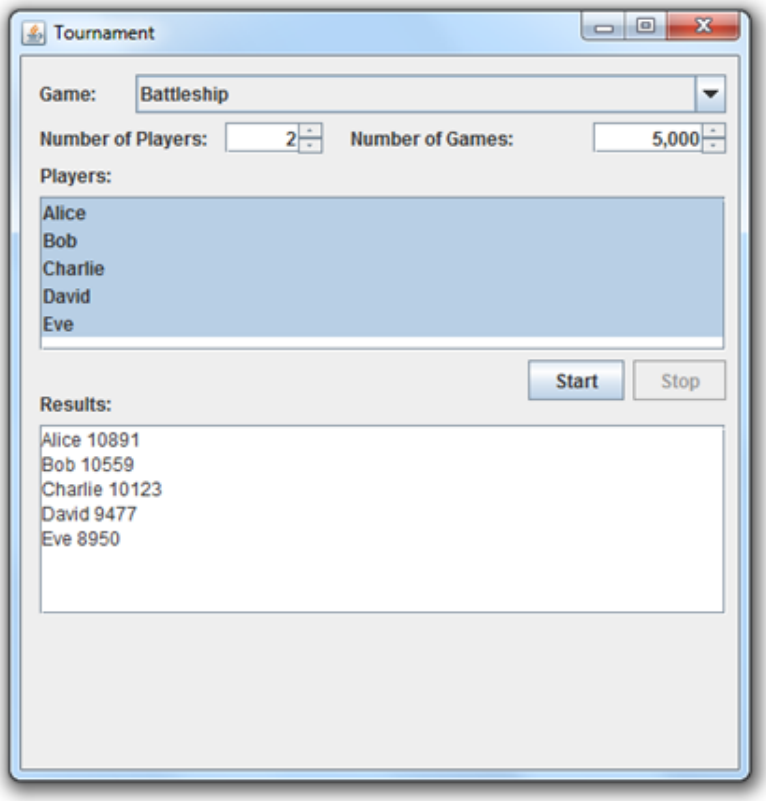

Figure 5.7: Tournament Simulation

Chapter 6

Evaluation

6.1 Invasion of the Greeps Assignment

The first tournament-based assignment given to the students was Invasion of the Greeps. For this assignment, students had to implement a class and provide the functionality to move aliens, know as Greeps, around a grid-based world. The goal was to have these Greeps collect as many tomatoes as they could from the world and bring them back to their ship. After the assignment had ended, a tournament was held to see which implementation of Greep could collect the most tomatoes from 10 different worlds. For full assignment description see Appendix A.

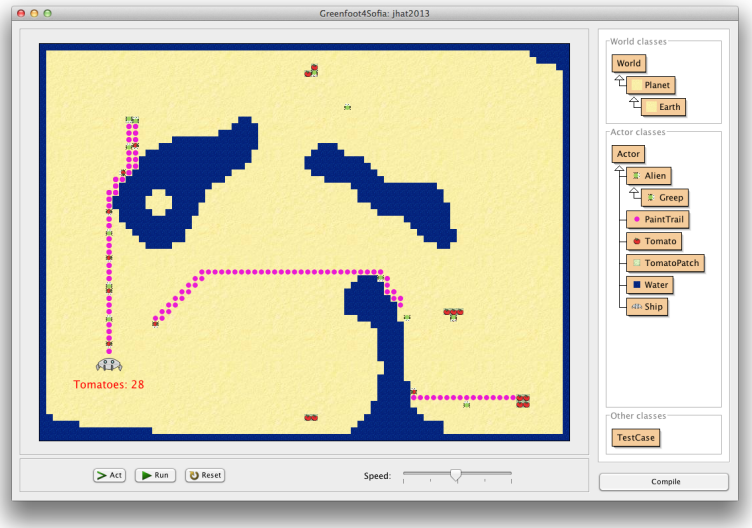

Figure 6.1: Greep Strategy being Simulated in Greenfoot

Despite the fact that this assignment was tournament-based it did not use the Genava framework due to time constraints (although it could have). Instead the Greenfoot IDE along with the Sofia API was used. These tools provide a similar experience too the Genava framework does for strategy development, but it does not provide the built-in capability of running tournaments. These tools were used instead because the project that would use the framework was still being finalized at the time.

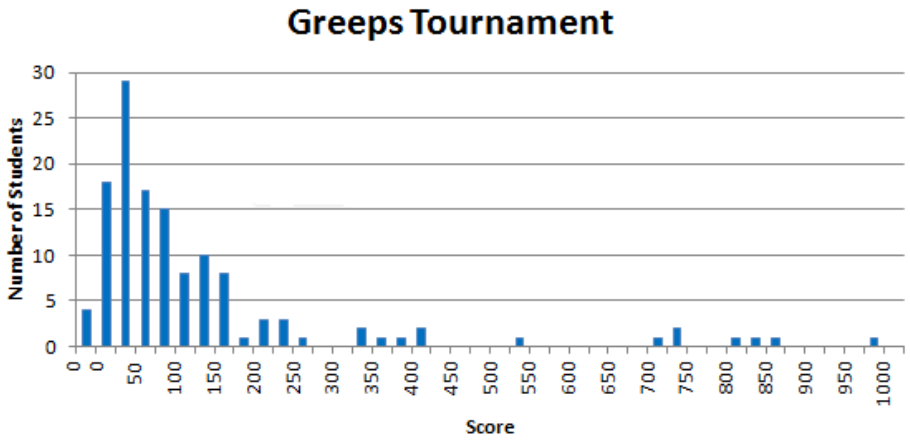

Figure 6.2: Greeps Tournament Score Distribution

The tournament was held after all submissions had be received. This tournament consisted of running each strategy manually on 10 different worlds and a strategy's score was comprised of the number of tomatoes they were able to collected from each world. If a strategy caused an error, it was disqualified. Out of the 160 students in the class, 153 students completed the assignment. Of these 153, 24 submissions were late and 22 submissions contained bugs. The results of the tournament revealed that the 25th percentile was 38 tomatoes, the 50th percentile was 74 tomatoes, the 75th percentile was 144.5 tomatoes, and the highest number of tomatoes collected was 992.

6.2 Battleship! Assignment

The other tournament-based assignment given to the students was Battleship. For this assignment, students had a week to implement an AI strategy capable of playing the board game Battleship. Their strategies were responsible for performing three basic behaviours at a minimum. First, when their strategy was asked for a ship placement, it had to return a valid move which placed all five ships on the board legally. Second, their strategy could not use the same ship placement for each game. Finally, their strategy had to generate legal firing moves each turn that eventually hit all 100 positions of their opponent's board. After the assignment had ended, a tournament was run between all students that had submitted a

solution to see whose strategy performed the best. The full assignment description appears in Appendix B.

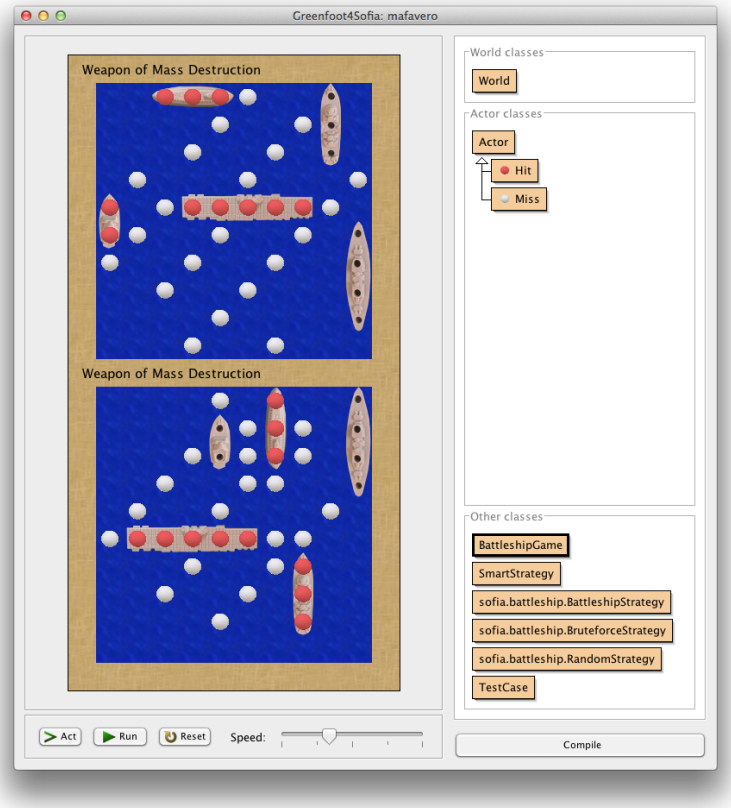

Figure 6.3: Two Battleship Strategies Competing Against One Another in Greenfoot

Unlike the Greep assignment, this assignment made use of the Genava Framework. The students used the same environment they used in the Greep assignment to develop their Battleship strategies. We felt that changing the tools students were using would have caused confusion that would have detracted from their performance in the assignment. To keep it simple, we created a version of Battleship for both Sofia and Genava and had students develop against the Sofia version of the game. After they submitted their source it was compiled and converted into a form capable of running within the Genava framework and could play against the framework version of Battleship.

As mentioned, a tournament was held, using the tournament simulation tool of the framework, between all strategies that were submitted. Out of the 160 students in the class, 144 completed the assignment, with 48 being late, and competed in the tournament. Each strategy in the tournament played 50 games against every other strategy and each strategy went first as often as they went second.

Along with the student's strategies, four additional strategies were also placed into the tournament. This brought the total to 147 competing strategies. These strategies were included as baselines to see where the student's strategies fell. These strategies were based on an analysis of the game done by the Data Genetics blog [5] and are as follows:

- **Random** simply fires randomly at every square on the opponent's board.
- **Bruteforce** starts by firing at the top-left square of the opponent's board and continues firing at squares in order until it reaches the bottom-right square.
- **Seek and Destroy** fires randomly at the opponent's board in a checkerboard pattern until it hits a ship. It then attacks every square around the initial hit, collecting a list of additional hits. This list is then processed until there are no more hits within it. At this point the ship will have been sunk and it resumes searching for new ships.
- **Probability Grid** computes the probability that a ship is at a position on the board by making use of previous hits, misses and sunken ships. After the probability has been computed for the entire board, the most probable position is chosen to be fired at.

While running the tournament, 33 strategies were disqualified by the framework for throwing exceptions, performing invalid moves, and causing infinite loops. The exceptions that occurred were caused mainly from strategies attempting to use the number 10 in the various methods accessible in Battleship. A Battleship board is a 10x10 grid, however, to access it a value between 0 and 9 is used. Students may have been confused by this and still used 10. Also, accesses were made using -1 which indicates that some strategies did not adequately perform checking of edge conditions.As mentioned, invalid moves also caused strategies to be disqualified. The invalid moves that were made by student strategies came in three different types. In some cases, students created a ship placement where ships would intersect each other. Another invalid move that students made, occurred when a strategy attempted to fire on a position that had previously fired at. The final type of illegal move occurred when strategies simply returned `null` and did not supply a move. Some strategies were also disqualified when their code went into an infinite loop while determining their move. As mentioned, the move retrieval mechanism of the framework will report a strategy's move as `null` after a 5 second timeout and disqualify them.

Error	Number of Occurrences
Exception	**14**
Illegal Argument	8
Array Out of Bounds	6
Invalid Move	**11**
Ships Intersecting	2
Position Already Fired On	4
Returned `null`	5
Infinite Loop	**8**

Table 6.1: Breakdown of Coding Errors found in Disqualified Strategies

After accounting for disqualified strategies, the total number of valid strategies came down to 114. Each strategy received 1 point for winning a game against another. Given 50 games with 114 strategies, each strategy participated in a total of 5,650 games and had the chance to get 5,650 points by winning each game.

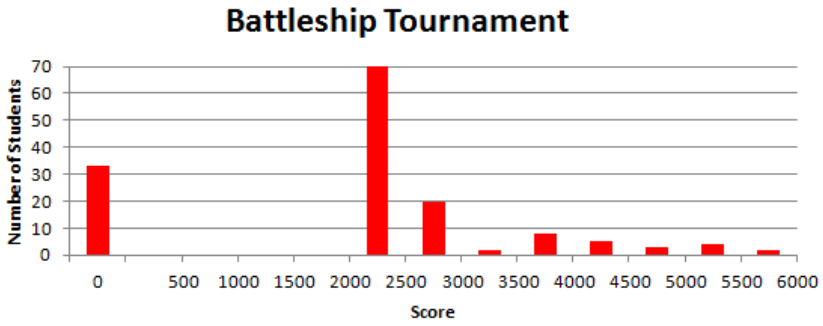

Figure 6.4: Battleship Tournament Score Distribution

After running the tournament, we were able to see the results of the students strategies compared to the baseline strategies. Random placed last in the tournament (not including disqualified strategies) and won 30.3% of games. Bruteforce placed 21st and won 63.2% of its games. Bruteforce's decent performance in the tournament is probably attributed to how strategies placed ships. Many strategies placed ship at random. This could cause ships to be clustered near the top of the board and would cause Bruteforce to win very quickly. Seek and Destroy placed 3rd in the tournament and won 96.1% of its games. This was surprising as originally this strategy was designed to represent a very basic strategy with room for improvement. However, there is a simple explanation for its performance. If many student's were placing ships randomly, then there was a high chance that two or more ships would be adjacent to one another. Seek and Destroy methodically checks the area around a hit which means that if two ships are touching it will hit the other ship. This means Seek and

Destroy could quickly find ships on boards where ships were placed randomly. Probability Grid placed 1st in the tournament winning 99.4% of its games. It should be noted that the student that placed 2nd had a strategy that performed at 98.8% and was most likely based on the same strategy that Probability Grid used.

6.3 Genava Framework Performance

Running the tournament for the Battleship assignment gave the perfect opportunity to test the framework's ability to handle a class-sized tournament. Also it provided a chance to see if the move retrieval and score correction mechanisms of the framework could keep up and perform correctly given the number of strategies within the tournament.

As mentioned, the Battleship tournament had 147 strategies competing. Given that a Battleship game is played by two players and that a framework tournament ensures each combination of players competes against each other, then there are 10,731 different player combinations. Also because each combination played 50 games this brings the total number of games to played to 536,550. However, during the tournament, 33 strategies were disqualified. After removing all trace of disqualified players, this brought the number of games down to 322,050 with each strategy participating in 5,650 games. The framework was able to simulate this tournament in 34.25 minutes on an machine with an Intel i7 processor and 8 GB of ram.

One major concern while developing the framework was that a strategy could crash the entire tournament if there was a bug in its implementation. As mentioned, a move retrieval mechanism was created to safely call methods from a strategy. When the tournament was run this mechanism held and the tournament was able to complete successfully even with a significant chunk of competing strategies causing errors. Also, it held up against every type of anticipated error that a strategy could cause.

The other major concern during tournament running was score correction. We needed to make sure that a strategy's score was only comprised of games against non-disqualified strategies. As mentioned, each strategy scored 1 point for every game it won. Also mentioned, was the fact that 322,050 valid games were played in the tournament after considering disqualified strategies. Therefore, if score correction was done correctly then the total score of each strategy when summed should equal 322,050. This proved to be the case and showed that score correction was indeed working successfully.

Overall, the framework was able to run the tournament smoothly. It also was able to handle loading various different strategies and was capable of running them independently. The framework also safely handled move retrieval and performed score modification correctly.

6.4 Student Data Analysis

To examine potential differences in student performance on tournament-based assignments compared to more traditional assignments, we asked students for permission to use data collected about each of their project submissions. Out of the students in the class, 88 allowed us to use their submission data.

One thing we were interested in investigating was if there was a distinct differences between the scores students received on the tournament-based assignments compared to more traditional assignments. Mainly, we wanted to see if tournament-based assignments would motivate students to perform better as opposed to the traditional assignments. It turns out that students scored on average a 34.13 out of 40 on the traditional assignments and on average scored a 34.83 out of 40 on the tournament-based assignments. The difference between the two is very minimal and shows that there was almost no difference between the scores on the tournament-based and traditional assignments. Analysis of the student's scores also showed that there was no statistically significant difference ($t = -0.45$, $p = 0.7242$).

Due to the open-ended nature of the tournament-based assignment, we wanted to know if students had difficulty in testing their strategies compared to testing a more traditional assignment. We tested this by computing the percentage of code coverage students achieved for each assignment. For the traditional assignments, students were on average able to test 88.12% of their code. For tournament-based assignments, students were on average able to test 89.76%. Analysis of the coverage showed that a comparison between the two does not show a statistically significant difference ($t = -o.71$, $p = 0.4759$). This helps to show that students had about the same difficulty debugging tournament-based assignments as they did the traditional assignments.

We also wanted to investigate whether tournament-based assignments had any affect on submission time. Mainly, we wanted to make sure that students did not struggle with tournament-based assignments causing them to turn them in late. After analysing student lateness we determined that there was no statistically significant difference (chisquare = 0.18, $p = 0.6746$) between the likelihood of an assignment being turned in late regardless of whether it was traditional or tournament-based.

6.5 Survey

To gain a better understanding of tournament-based assignments, a survey was given to the students after they had completed both the Invasion of the Greeps and Battleship assignments. The survey focused on how engaging and frustrating these assignments were as well as how students felt about the tournaments present in both assignments. The survey contained 32 questions split into two portions.

One thing to note is that the survey was optional and many students chose not to participate in the survey. In fact, out of the entire class of 160 students only 26 students participated in the survey. Although this is a small portion of the class, the answers were varied enough to indicate that a wide range of students with differing opinions on the assignments and not just those that enjoyed them answered the survey. Also, from these few people their was some noticeable trends in their answers. Because of this, an analysis of the answers was done with the hope that they represent the class as a whole.

6.5.1 Invasion of the Greeps and Battleship Questions

The first portion of the survey consisted of the same set of 11 questions. These questions were asked about both the Greeps and Battleship Assignments. All results pertaining to the Greeps assignment will be shown in red, while all results associated with the Battleship Assignment will be shown in blue.

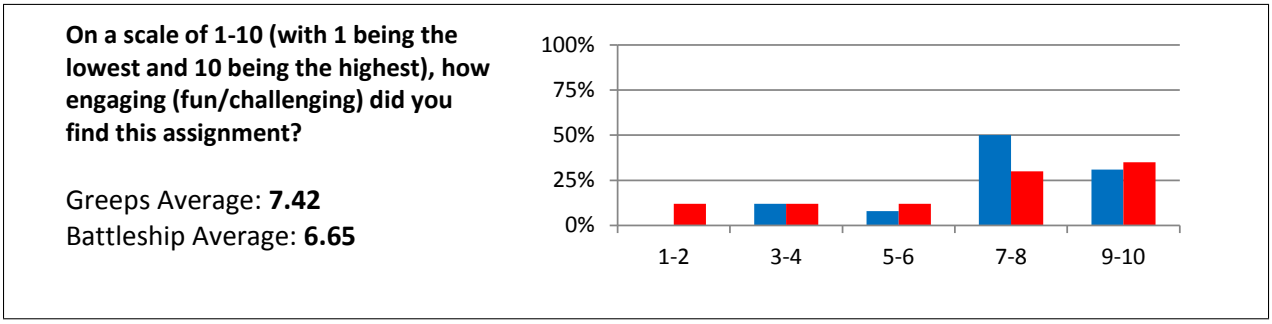

Figure 6.5: Survey Question on Engagement of Tournament-Based Assignments

This question was asked to see if students enjoyed the Greeps and Battleship assignments. We wanted to see if students found tournament-based assignments to be fun and engaging. On average, students seemed to find both assignments to be engaging with many students given values between 7 and 10. It was encouraging to see that these assignments were capable of providing a fun an interesting challenge to students.

Figure 6.6: Survey Question on the Engaging Aspects of Tournament-Based Assignments

This question was asked to determine better what made the assignment engaging. Specifically, we wanted to see what components of these tournament-based assignments made them enjoyable to students. Also, we wanted to see how the tournament factored into student's enjoyment.

For the Greeps assignment, students said that they enjoyed the competition created by the tournament. Many students commented about how they kept trying to change their strategy so it would be able to collect more tomatoes. Also, many students enjoyed the challenge that the problem posed and enjoyed trying to figure out a solution. Some also said that the constraints put into the assignment helped to level the playing field between the novice programmers and the ones with more experience. Overall, the competition and challenge of the assignment helped to make it engaging.

For the Battleship assignment, students also seemed to the enjoy the competition produced by the tournament with some students mentioning that they liked how their strategy would be going directly against other strategies and not against a scenario like with the Greeps assignment. Students also commented that they really enjoyed the open-endness of the problem and that they enjoyed the freedom they had in coming up with their strategy. Finally, students said they enjoyed analysing the problem an attempting to figure out an optimal solution for the game of Battleship.

Figure 6.7: Survey Question on Increasing Engagement of Tournament-Based Assignments

This question was asked to determine what aspects hindered these assignments from being as engaging as they could be. We wanted to see what could be done to improve on these assignments.

For the Greeps assignment, many students did not know how it could be made more engaging, which shows that this assignment is at a very good place. The major thing mentioned by students was that the worlds that they had to test on ended up shaping their strategy to the point where they created specific solutions that worked well for these maps, but ended up not working very well on the unknown maps. Students wished that they had a way to create random maps to test on.

For the Battleship assignment, students indicated that wanted to have more example code to help them get started in making their strategy. Many students felt some of the concepts introduced in Battleship were not fully explained during class time and that the game functioned in a different way then they expected. Students also wished that they had some way of comparing their strategies with others. In the Greep assignment they could post the scores their strategies were getting and could look up scores others had posted, but with Battleship students did not know how to compare their strategies. This problem could be solved by having a few mock tournaments for submitted strategies before the final tournament takes place. Finally, some students complained that Battleship had some optimal solutions that could be looked up. Although they were difficult to program some students felt this detracted from strategy experimentation.

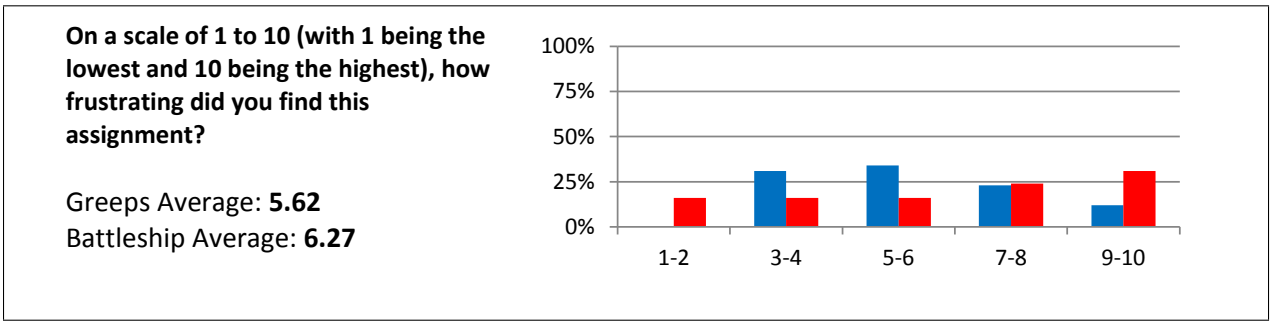

Figure 6.8: Survey Question on Frustration of Tournament-Based Assignments

This question was asked to see how frustrating students found the tournament-based assignments. It was good to see that students were not overly frustrated, on average, with the assignments. The results indicate that students found Battleship to be more frustrating than Greeps. It should also be noted that despite students finding these assignments engaging they also found them to be frustrating. This could indicate that an assignment that is highly engaging may also cause a moderate amount of frustration to occur within students.

What did you find most frustrating about the assignment?

Figure 6.9: Survey Question on the Frustrating Aspects of Tournament-Based Assignments

This question was asked to see what students found the most frustrating about these assignments. We wanted to see if their frustrations came more from failing to adequately explain aspects of the project and less from the tournament aspects of the assignment.

For the Greeps assignment, many students found it frustrating to figure out how to move their Greep around the water that was found on the worlds. Many students found this to be a very difficult and frustrating problem. However, this was part of the assignment and was a problem students were expected to solve. The second most frustrating thing students discussed in their answers was about testing. Many students found it hard to test their solutions until they were given examples on how to do so.

For the Battleship assignment, students said they had a tough time understanding how to begin creating their strategy. To create their strategy students had to implement a Java interface, which they had no experience with prior to this assignment. Also, the assignment required students to read through the Battleship API to understand how the different components worked. This caused further difficulty with students. Students also wished to be given a sample strategy so they could see how they could create their own. One concern that comes from this is that the Genava framework relies heavily on interfaces and the introductory students had some difficulty in understanding how they worked. Before doing an assignment using the framework, students will need to be given a lesson in interfaces

as opposed to the simple crash course like the one given in the assignment write up for Battleship.

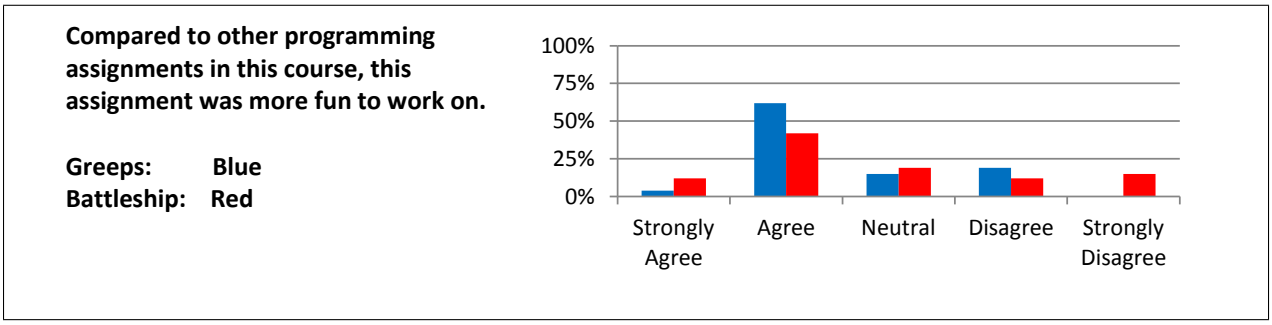

Figure 6.10: Survey Question on Enjoyment Levels of Tournament-Based Assignments

This question was asked to see if students found the tournament-based assignments fun. The idea behind these assignments was for students to have fun while working on them so they would be more interesting then traditional projects. Looking at the results, it is clear that a majority of students thought the assignments were fun. Also, it should be noted that the phrasing of this question would put the two tournament-based assignments against one another which would affected how students answered depending on if they liked Greeps or Battleship more.

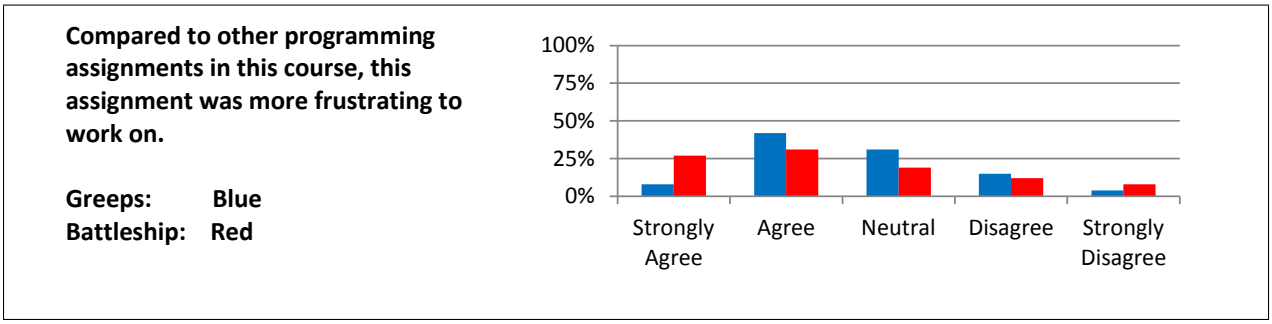

Figure 6.11: Survey Question on Frustration Levels of Tournament-Based Assignments

This question was asked to see how frustrating students would find the tournament-based assignments compared to other assignments they had completed. We had hoped that by creating assignments that were more fun it would cut down on students frustration. Interestingly, a majority of students found the assignments to be frustrating. In fact when comparing the fun and frustrating graphs, they are very similar to one another. As mentioned earlier, this may be evidence that a fun project also might be frustrating. Also, the tournament-based projects were some students first efforts in working with a large code base. This could have added to their frustration as they worked on the project.

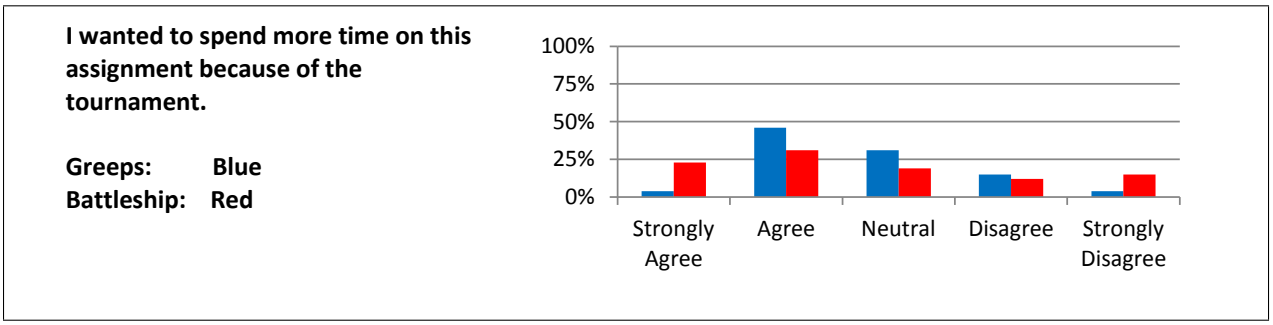

Figure 6.12: Survey Question on Increased Time Commitment of Tournament-Based Assignments

This question was asked to see if students would embrace the open-endness of creating a game strategy capable of being competitive in the tournament by going above the basic requirements of the assignment. The results confirmed that some students did want to spend more time working on their strategies so that they would have a better chance in the tournament. Also an interesting effect noticed is that more students strongly agreed to this question in regards to the Battleship assignment then to the Greeps assignment. The only difference between the two tournaments was that in the Battleship tournament strategies directly competed against one another, as opposed to the indirect method used by the Greeps assignment. Students may have been motivated to spend more time on the assignment knowing that their strategy would be directly competing against other strategies and that their strategy could influence another strategy's placement in the tournament.

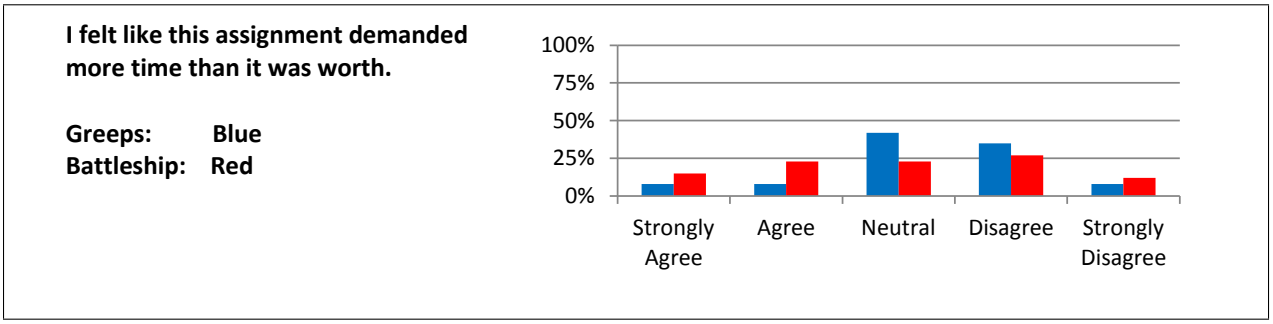

Figure 6.13: Survey Question on Gains versus Time Commitment of Tournament-Based Assignments

This question was asked to see if students felt that the gains of knowledge given by these types of assignments were worth the time investment. The results were mixed in not only response but for the two assignments as well. For the Greeps assignment, a majority of students seemed to be indecisive about this question or disagreed. For Battleship, student responses were very mixed with students feeling both ways about this question. This very diverse result may indicate the difference in skill level between students. Students that had

a better grasp of programming concepts may have been able to complete the assignments in a timely manner, whereas struggling students may have spent more time than they felt necessary.

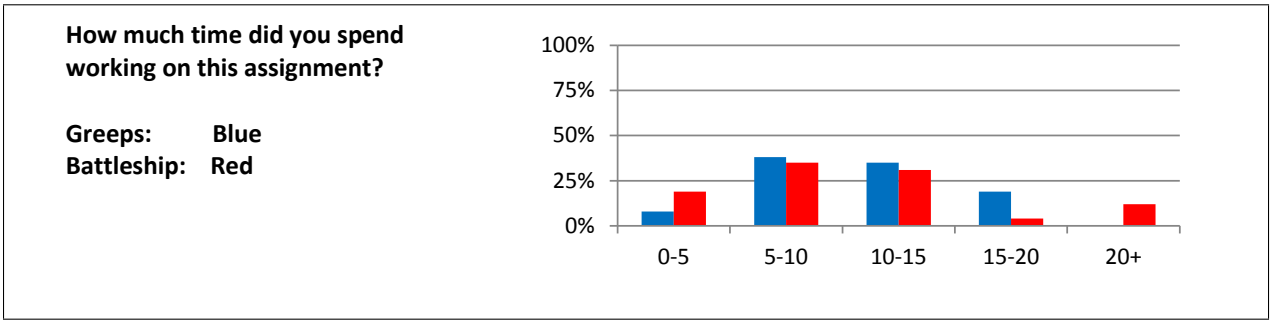

Figure 6.14: Survey Question on Development Time Spent on Tournament-Based Assignments

This question was asked to see the amount of time students were spending on the tournament-based assignments. We first wanted to see if students were investing an appropriate amount of time into completing their strategies and we also wanted to make sure that the open-endness of strategy creation did not cause students to go completely overboard on the assignment. The results show that most students spent between 5-10 hours or 10-15 hours. This range of time spent does not seem too excessive for these types of projects and shows that students were willing to put time into working on these types of assignments.

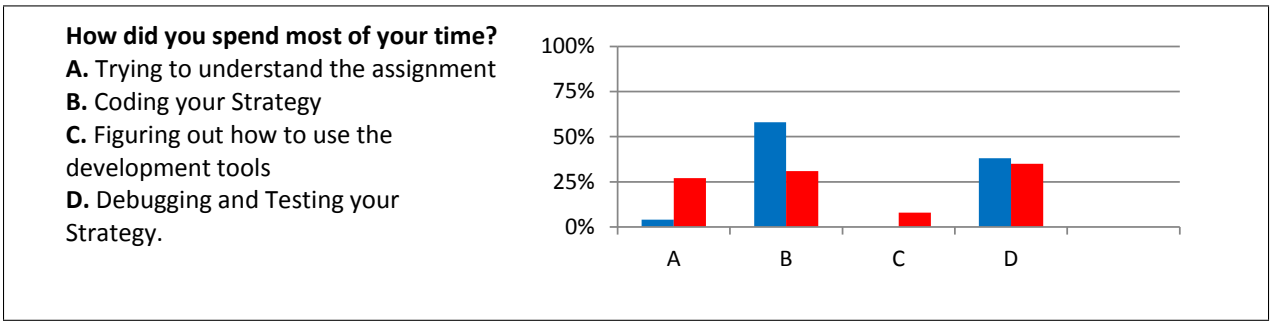

Figure 6.15: Survey Question on How Time was Spent on Tournament-Based Assignments

This question was asked to see how students were spending their time. We wanted to make sure that students spent the most of their time developing their strategies and less time trying to understand the assignment and learning how to use the development tools for an assignment. For the Greeps assignment this ended up being the case as most students spent their time creating and testing their strategy. However, students reported having difficulty understanding the Battleship assignment which took time away from strategy development.

This indicates that the Battleship assignment still needs some refinements to ensure a student has complete understanding of it before they begin to develop their strategy.

On the whole students have indicated that they enjoyed these assignments because of how engaging and fun they were. Also, students noted that these assignments were also frustrating though not enough to impact the enjoyment the students got from working on the assignments. Students found the competitive nature of the tournament to be enjoyable and mentioned that it was the tournament aspect that made the assignments fun and engaging. Students also indicated that they enjoyed the open-endess of these assignments. Overall, it seems the Greeps assignment is very good and can be made better with some minor tweaks, while the Battleship assignment could use some improvement to make it not as difficult to understand. Based on the student response it is clear that these tournament-based assignment, specifically, were well received by students.

6.5.2 Tournament-Based Assignments Questions

The second portion of the survey was dedicated to asking students about tournament-based assignments in general and about how they felt about them. This portion consisted of 10 questions.

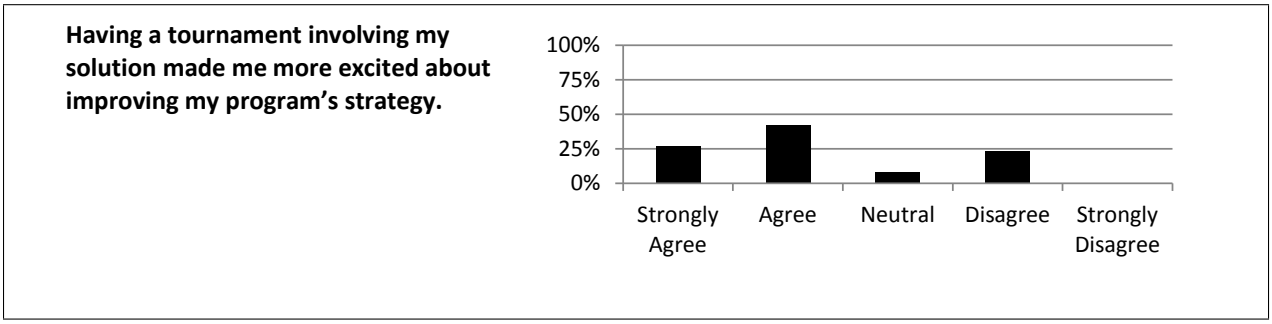

Figure 6.16: Survey Question on Improving a Strategy

This question was asked to see if having a tournament involved with the assignment would cause students to become excited about creating their strategy and wanting to improve it until they felt it would be competitive in the tournament. The results showed that some students generally were more excited while some were not. It was also good to see that students that disagreed with this question did not do so strongly whereas some students strongly agreed that they became more excited.

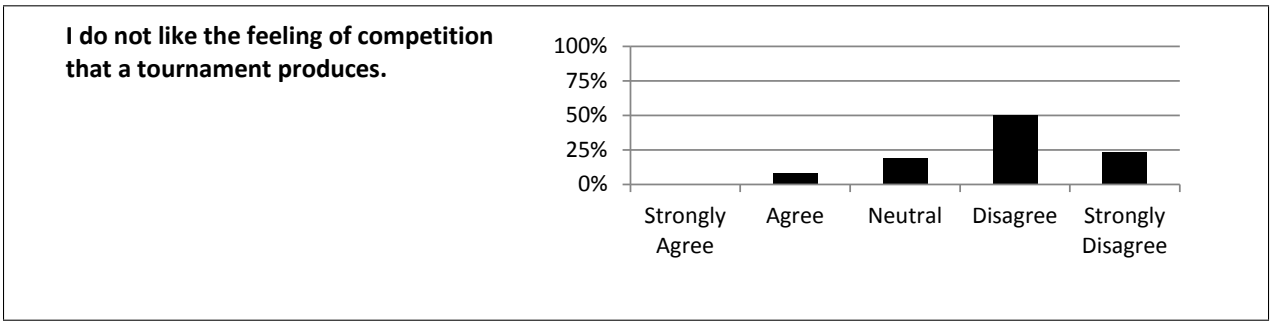

Figure 6.17: Survey Question on Tournament Competition

This question was asked to see if students were put off by the idea of the competition a tournament creates. We wanted to make sure that the tournament did not intimidate students. It was good to see that many students did not mind the competition with very few students agreeing that they did not like the competition the tournament produced. Another positive is that no students strongly agreed to not liking the competition.

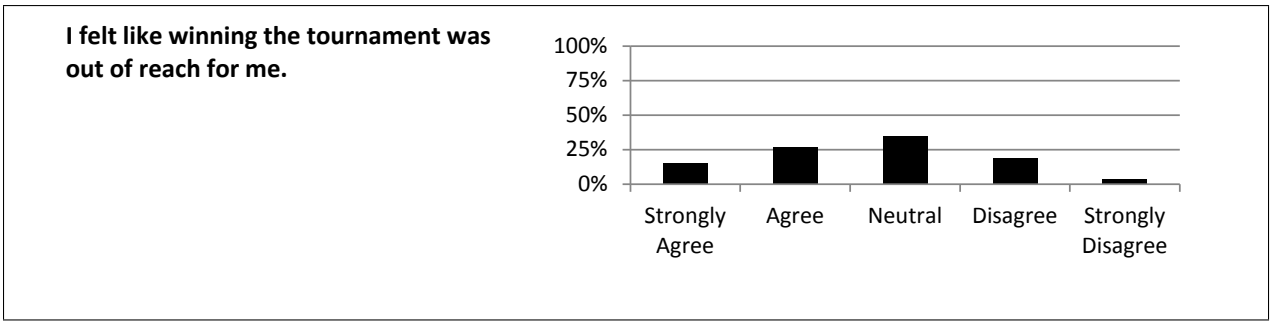

Figure 6.18: Survey Question on Winning the Tournament

This question was asked to see if students felt confident enough in their programming ability to be able to produce a strategy capable of winning the tournament. Unfortunately, it seems as if more students agreed that they did not feel capable of winning the tournament. However, given the large size of the class it is understandable that students would feel this way. It was also reassuring to see that some students felt they could win.

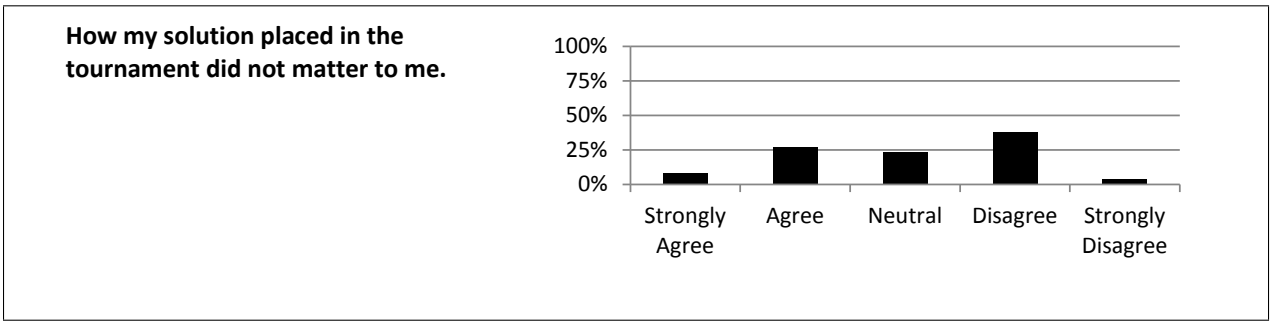

Figure 6.19: Survey Question on Tournament Placement

This question was asked to see how students felt about creating a competitive strategy capable of doing well in the tournament as opposed to a simple strategy capable of meeting the basic requirements. The results were fairly even with a few more students caring about how they placed in the tournament. This split helps to show that tournaments may motivate some students to creating better code for their strategies, but there are still students who may only attempt to do the minimum requirements needed to complete the assignment.

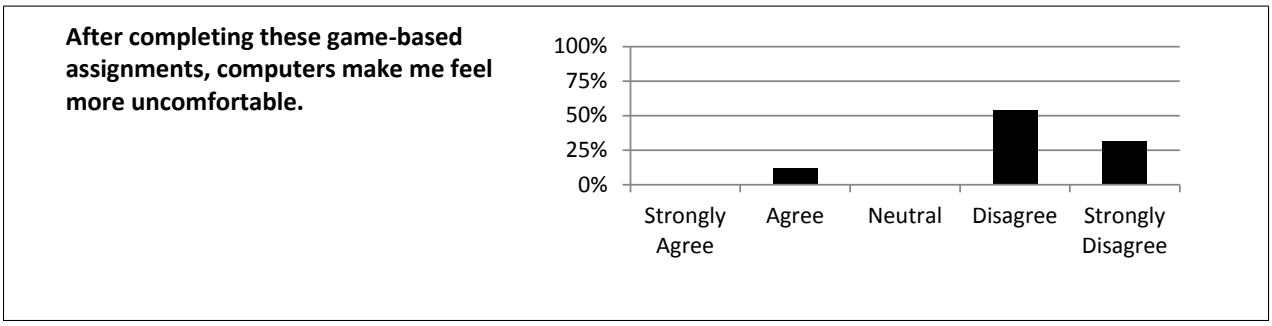

Figure 6.20: Survey Question on Student's View of Computers

This question was asked to see if tournament-based assignments had a negative impact on students and made them more uncomfortable of computer programming. While many students did disagree, the phrasing of this question might have been interpreted by some students as general computer use which would affect how they answered.

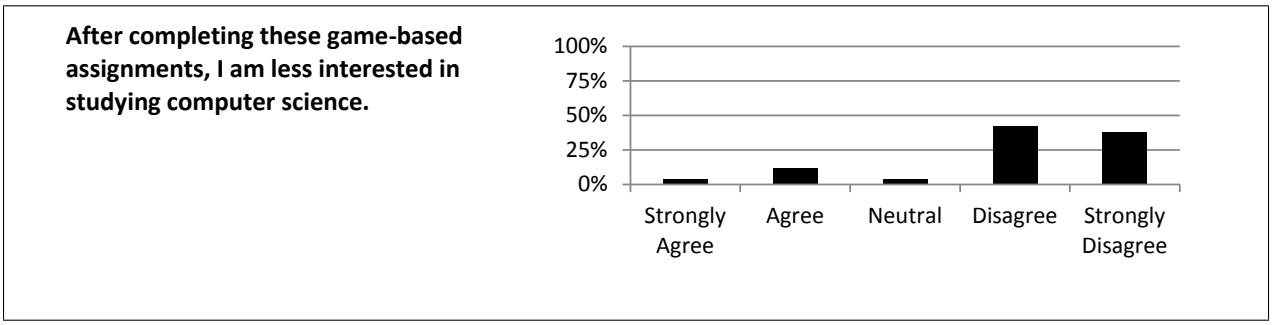

Figure 6.21: Survey Question on Student's Interest in computer Science

This question was asked to see if tournament-based assignments had a negative impact on a students interest in Computer Science. It was encouraging to see that many of the students surveyed were not adversely affected by tournament-based assignments and that only a minority had lost interest.

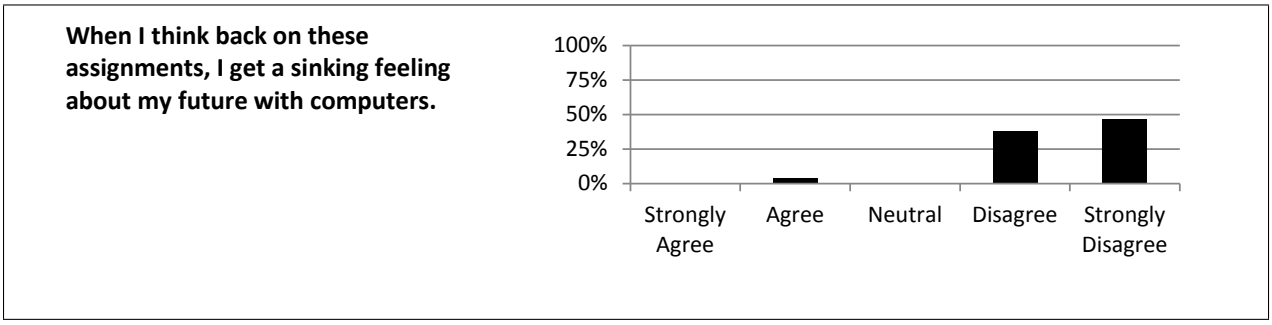

Figure 6.22: Survey Question on Student's Future with Computers

Similarly to Question 5, this question was asked to see if students, after participating in these assignments, felt that they had a future that involved computers and programming in some form. While most felt they did, this question has the same problem as Question 5 and students may have interpreted it as generally using computers in the future.

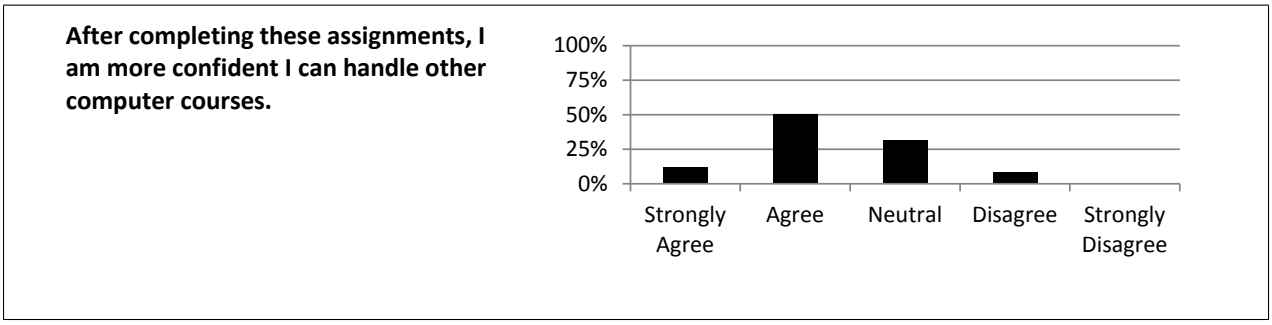

Figure 6.23: Survey Question on Student's Ability to Handle Future Computer Courses

This question was asked to see if students felt that they would be able to handle other computer courses. It was extremely encouraging to see that most students felt that these assignments helped to improve their confidence of their ability to d well in future computer courses. It is good to know that these types of assignments can be used to instil confidence into students.

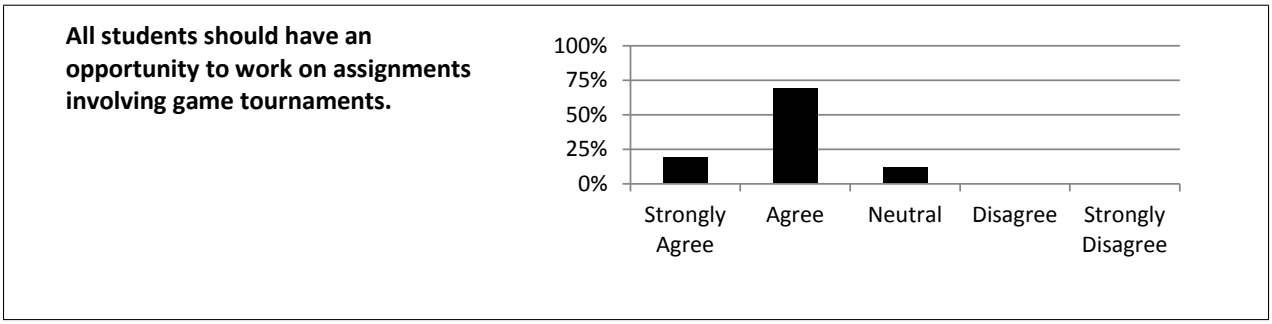

Figure 6.24: Survey Question on Opportunities to Work on Tournament-Based Assignments

This question was asked to see if students felt that other students should be given the opportunity to participate. Mainly, we wanted to see if students enjoyed and learned from the assignments enough to recommend that their use should be continued in introductory Computer Science courses. Of all results this is probably the most encouraging as a vast majority of the students agreed that students should have the opportunity to work on a tournament-based assignment. This was further encouraging by the fact that even students who did not care as much about the tournament or their placement in it still said that tournament-based assignments should be given to students.

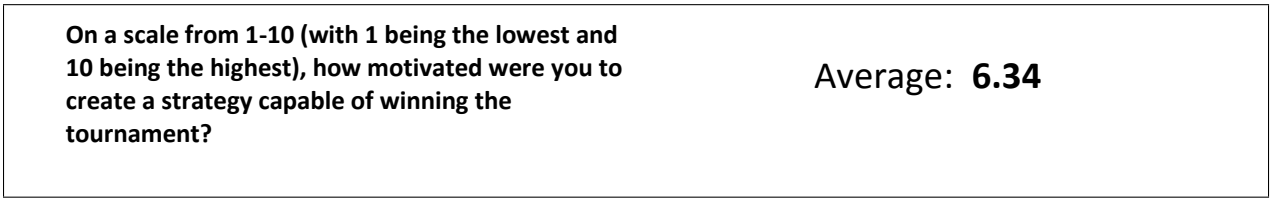

Figure 6.25: Survey Question on Student's Motivation to Create a Winning Strategy

This question was asked to gain an understanding about the motivation levels students had to create a strategy that could win the tournament. It was good to see a value of 6.34 as a more even distribution of unmotivated and highly motivated students would have been closer to 5. A value of 6.34 helps to show that students tended to show higher levels of motivation towards creating a winning strategy.

Overall, these questions helped gain some insight about tournament-based assignments. For the most part it seems that students agree that tournament-based assignments are an interesting idea and that other students should have the chance to work on them. Also, these

types of assignments are capable of motivating students to go above the basic requirements of an assignment when creating their solutions and the competition inherit in a tournament does not seem to affect this motivation. Also, students felt prepared for future computer courses after completing these types of assignments and indicated that they feel that they have a future involving computers. All these results indicate that tournament-based assignments can have a positive effect on students.

6.6 Solution Size Distribution

One of the claims that has been repeated throughout is that tournament-based assignments are more open-ended than traditional projects and as such allow students to create a wider range of solutions compared to more traditional assignments. To see if this was true we looked at the number of non-commented lines of code that were contained within each students final submission to each project. In addition to Greeps and Battleship, there were three non-tournament-based assignments that were given to the students. The first two were very small projects designed to teach the students basic programming concepts. The last project was for students to implement a working version of the game Asteroids given certain restrictions.

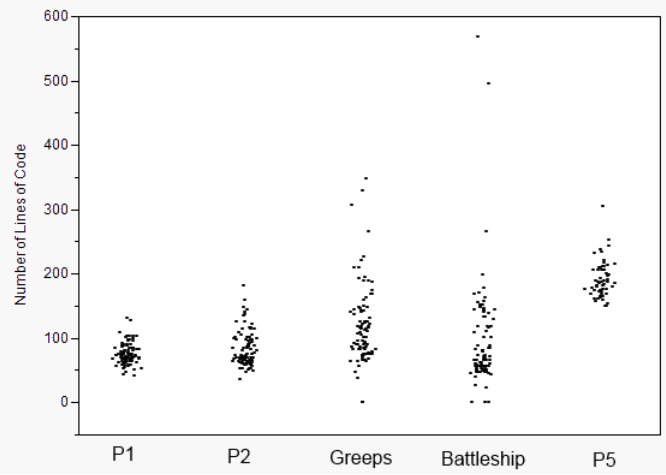

Figure 6.26: Number of Lines of Code per Student for each Project

The distribution showed that the tournament-based assignments had a much wider range in the number of lines of code used in a students solution. What is even more surprising is that the distribution of program 2 and program 5 is very similar. This is interesting because students were told they could make any modifications they wanted to their Asteroids game as long as it met basic requirements of the project. It seems that without the motivation of the tournament, students focused on meeting the basic requirements of the project. Overall,

this figure shows that tournament-based assignments can result in a much larger distribution of solution sizes.

6.7 Strategy Abandonment

During the duration of the Battleship assignment, some students mentioned that they abandoned a more complex strategy in favor of one that was simpler. Students mentioned that they did this due to difficulty in testing strategies and due to the time it would have taken to complete their complex strategies. We wanted to make sure that this was the case for a minority of students. Of the 144 students that completed the Battleship assignment, we were allowed to analyse 88 of the student's submissions data. In attempt to determine if strategy abandonment occurred we compared the average number of non-commented lines in a student's solution compared to the number of non-commented lines of a student's final submission. We felt that a dramatic change in the values could indicate a strategy was abandoned and substituted with a new strategy.

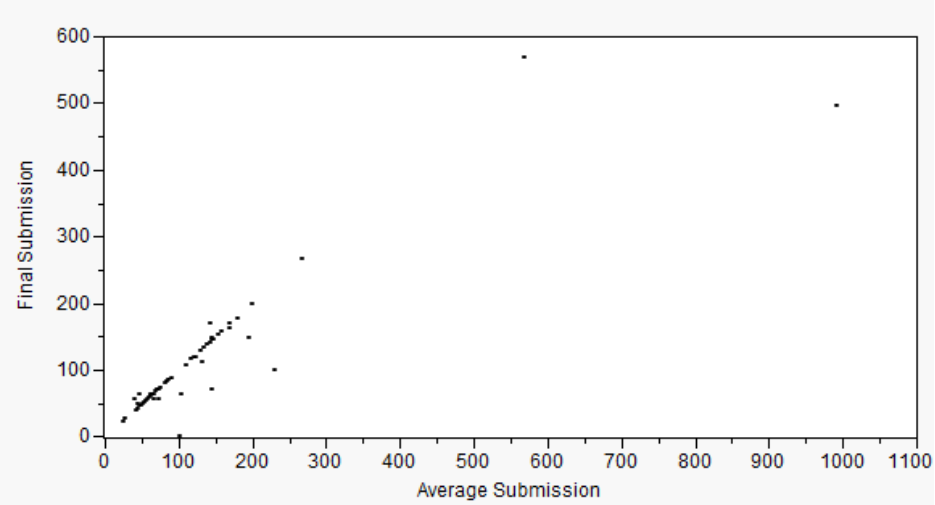

Figure 6.27: Final Submission NCLOC vs Average Submission NCLOC

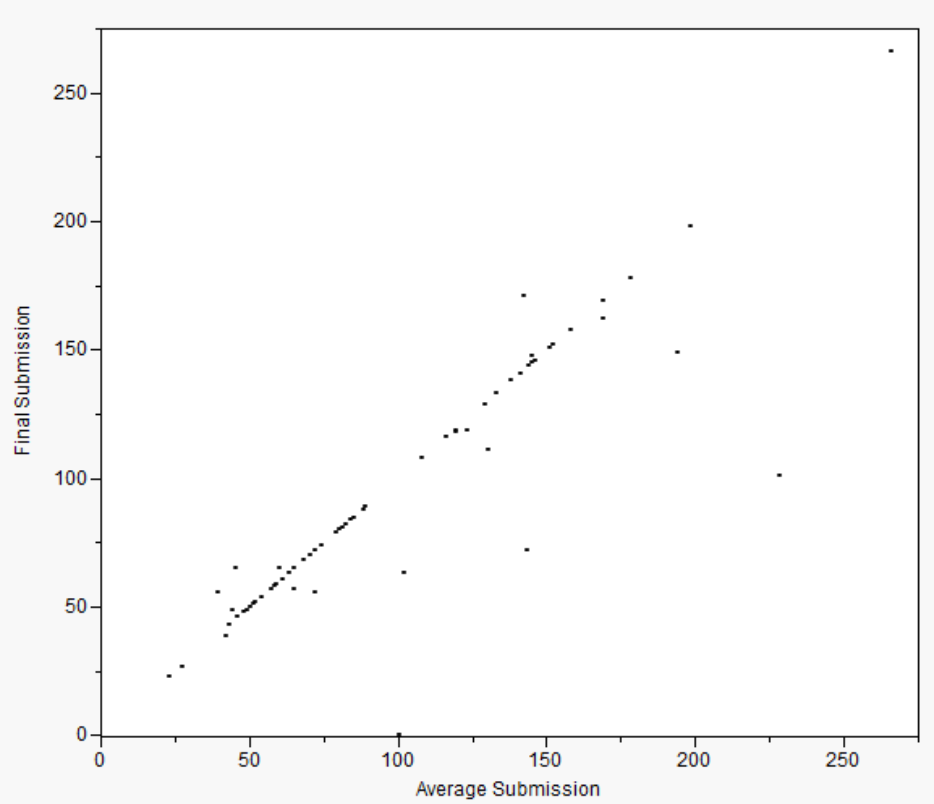

Figure 6.28: Final Submission NCLOC vs Average Submission NCLOC (excluding outliers)

Comparison of the submissions showed that the average and final sizes for many students did not differ by much. This indicates that a student's strategy was fairly consistent during each submission. There are also a few students who have notable drops in their line count (In one case a student dropped more than 400 lines of code) which could indicate that a strategy was abandoned. However, it is also possible that students could have been removing redundant code and compressing their solution. The important thing to take away is that only a few students deviate vastly from their average and final submissions. Therefore,the students who reported strategy abandonment seemed to be isolated incidents and a vast amount of the students were not affected.

Chapter 7

Summary

The goal of our work was to create a framework capable of reducing the development burden of tournament-based assignments while also providing the capability of easily running a tournament with any games and strategies that are created for it. We have taken steps towards this goal. Our framework is capable of running a variety of turn-based games and of running class-sized tournaments easily. Overall, the framework is heading in the right direction and with a few more modifications it can be improved further.

One thing that could be done is to expand the types of games that can be played by the framework. Currently, the framework only supports games that can be represented as turn-based games. One obvious expansion would be to support a more real-time game in which players must constantly make moves. Also, the library of games playable by the framework could be expanded. Currently, there are only a few games that have been implemented. By increasing this library, a wider range of programming topics can be covered by the framework. Also, as more games are made they can be ranked by difficulty.

One small flaw in the framework design needs to be fixed. While the move retrieval methods within a strategy are called safely, no other strategy methods are protected in this way. Both the `initialize()` and `newGame()` methods could also contain bugs capable of crashing the tournament. An error in these methods did not occur in the Battleship tournament simply because most students did not do much with these methods. However, a student could have easily caused an infinite loop or exception and this would have crashed the program. A similar mechanism, like the one used for move retrieval, could correct this problem. One way this could be done is to create a class that all calls to a strategy go through. This class would ensure that any errors that occur within the strategy would not affect the tournament.

Another interesting thing to explore would be trying to distribute the execution of a tournament. Currently, all rounds of a tournament are run back to back. However, because of how the tournament is structured it is possible to run multiple rounds of the tournament at the same time. It could decrease tournament runtime if multiple rounds could be played at

once. To make this work, a small game simulation program would need to be developed that could be replicated across nodes. This simulation program would need to be able to accept parameters that would define which game to create and run and what strategies would be involved. After running, the results would need to be sent back to the node running the tournament. It would be interesting to see if a decrease in runtime could be gained by such a system. Also, tournament rounds could be run using multiple threads.

The development process of games and strategies could be better integrated within an IDE. Right now, the development tools are located within the JAR which contains the framework. It would be better if these tools could be better hooked into the IDE itself. For Eclipse, this could be done by creating a plugin which would facilitate the creation, development, and debugging of games and strategies for the framework. The idea would be that developers could quickly choose a game they wish to create a strategy for and have a project created which provides the necessary classes needed to create the strategy. Also, the development tools could be builtin to the GUI of Eclipse which would allow for easier access.

Another way to expand the framework would be to expose the framework's functionality as a web-based application. A design like this was considered when originally creating the framework, but it was scrapped due to time constraints. The idea is that users would be able look up a game they wish to create a strategy for. They could download all the tools needed to create the strategy and develop it. After developing they could use a submission tool to submit their strategy where it would begin participating against other strategies and receive a ranking within the system. On the other side users could create games that other users could create strategies for. These users could also create tournaments that others could participate in. There is almost no limit to what like this could accomplish and serves as an end goal for this system.

The other goal of our work was to analyse tournament-based assignments to see the potential benefits of using them over a more traditional assignment. From the reactions of students and our experiments it appears that tournament-based assignments are indeed an excellent choice when creating assignments for introductory CS courses. These types of assignments are popular amongst students and they provide an open-ended experience to students who will go beyond the basic requirements while working on them. These types of assignments help motivate students to learn CS concepts and also provide students with a fun and engaging experience. These assignments also show no signs of negatively impacting student achievements, with student's scores not differing by much compared to traditional assignments.

Students also reacted slightly differently to both tournament-based assignments that were examined. The differences in the assignments led to different levels of enjoyment. Future work could be done to analyse specifically these differences to better create tournament-based assignments students enjoy. This can be done using our framework as it provides the ability to switch out games easily. Although, the individual assignments themselves may need tweaking to be more engaging, tournament-based assignments in general seem to be a

suitable choice over a more traditional assignment.

Overall, we are happy with the results we have obtained about tournament-based assignments and feel that the Genava framework can eventually provide instructors the ability to create new and exciting assignments for their students. Hopefully, more work will be done with our framework in the future.

Bibliography

[1] Ai challenge. http://aichallenge.org/.

[2] Android developers. http://developer.android.com/index.html.

[3] Core wars. http://www.corewars.org/.

[4] Core wars history. http://corewar.co.uk/history.htm.

[5] Data genetics blog. http://www.datagenetics.com/blog/december32011/index.html.

[6] Robocode. http://robocode.sourceforge.net/.

[7] V. Allis. A knowledge-based approach of connect-four–the game is solved: White wins. Master's thesis, Vrije Universiteit Amsterdam, 1988.

[8] Jessica D. Bayliss. Using games in introductory courses: tips from the trenches. In *Proceedings of the 40th ACM technical symposium on Computer science education*, SIGCSE '09, pages 337–341, New York, NY, USA, 2009. ACM.

[9] Ivona Bezakova, James E. Heliotis, and Sean P. Strout. Board game strategies in introductory computer science. In *Proceeding of the 44th ACM technical symposium on Computer science education*, SIGCSE '13, pages 17–22, New York, NY, USA, 2013. ACM.

[10] Daniel C. Cliburn and Susan Miller. Games, stories, or something more traditional: the types of assignments college students prefer. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, SIGCSE '08, pages 138–142, New York, NY, USA, 2008. ACM.

[11] D.C. Cliburn. The effectiveness of games as assignments in an introductory programming course. In *Frontiers in Education Conference, 36th Annual*, pages 6–10, 2006.

[12] Peter Drake and Kelvin Sung. Teaching introductory programming with popular board games. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 619–624, New York, NY, USA, 2011. ACM.

[13] Eman El-Sheikh and Lakshmi Prayaga. Development and use of ai and game applications in undergraduate computer science courses. *J. Comput. Sci. Coll.*, 27(2):114–122, December 2011.

[14] Scott Leutenegger and Jeffrey Edgington. A games first approach to teaching introductory programming. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, SIGCSE '07, pages 115–118, New York, NY, USA, 2007. ACM.

[15] N. Parlante, J. Zelenski, D. Zingaro, K. Wayne, D. O'Hallaron, J. T. Guerin, S. Davies, Z. Kurmas, and K. Debby. Nifty assignments. *In Proceedings of the ACM SIGSCE '12*, pages 475–476, 2012.

Appendix A

Invasion of the Greeps Assignment

This appendix contains the full assignment write up for the Invasion of the Greeps assignment.

A.1 Goal

Alien creatures have landed on Earth–they are the Greeps. Greeps like tomatoes. By some incredible stroke of luck, they have landed in an area where tomato patches are found randomly spread out over the otherwise barren land. Help your Greeps collect as many tomatoes as possible and bring them back to their ship.

You will accomplish this task by implementing your own `Greep` class (the other classes are already provided as part of a library). At a minimum, your solution must be able to bring at least one tomato back to its ship.

A.2 Starting Materials

Download the scenario for this assignment, which contains all the classes you need to start.: program3.zip (available Monday night).

The starting scenario has the following classes already defined (none have methods you can use in your solution):

`Earth`

The Earth represents the map (or world) where Greeps operate. The Earth class provides two constructors for your convenience as you work on this project:

The `Earth()` constructor takes no parameters and uses three separate maps (with randomly placed tomato patches) to show a simulated ship landing on each map. An entire ship's crew of Greeps will be dispatched and allowed to harvest for a limited time. After all three maps have been used, your "score" (the number of tomatoes retrieved) will be shown. You can use this for running an interactive mission test of your `Greep` class.

The `Earth(int)` constructor takes a number from 1-3 as parameter, and shows only the specified map, with tomato patches at fixed locations. You can use this for testing by dropping a greep (and a ship) anywhere you choose to test out individual methods.

`Water`

As with jeroo scenarios, water represents an obstacle that cannot be crossed. If a Greep enters the water, it drowns and is removed from the simulation.

`Tomato`

Tomato objects grow on tomato patches are represent the food that Greeps love. However, Greeps are pretty small, and cannot simply pick up a tomato by themselves. In order for a Greep to carry a tomato, two greeps must be together–then one of them can load the tomato onto the other.

`TomatoPatch`

Represents a fertile area of the ground where tomatoes grow. If a tomato is loaded by one Greep onto another, a new tomato will replace it after 10 turns.

`PaintTrail`

Greeps cannot communicate directly with each other, but they can leave a trail of paint on the ground so that other Greeps can find it. This paint disappears over time.

`Ship`

The space ship the Greeps landed in serves as their home base. Greeps can always sense roughly in which direction their ship lies. Tomatoes must be returned to the ship in order to be counted as "harvested".

`Alien`

The base class for `Greep`. This class defines the methods that all Greeps share, and which you can use to construct your solution. This class is documented in the next section.

A.3 Structuring Your Solution

Your task is to write the `Greep` class, which must be a subclass of Alien. However, unlike jeroos, the `Alien` class is a bit more general and does not use a `myProgram()` method.

Working with the `Alien` class in this assignment requires us to understand a more general model of how micro-worlds operate. Micro-worlds are a kind of discrete event simulation–think of a "clock" that ticks, and for each clock tick, all of the actors in the world get a chance to do something. At every clock tick, every actor gets a "turn" to act. When we were working with Jeroos, we provided a fully scripted sequence of actions in a `myProgram()` method, and on each "turn" the Jeroo took "the next step" in that plan, methodically plodding along. In this simulation, however, there will be a large number of Greeps all working at the same time. For each turn, every one of them will have a chance to act. Their actions will change the state of the world, so they may act differently on the next turn–they will react to their surroundings instead of carrying out a predetermined sequence of actions.

To make the simulation work, your `Greep` class will provide an `act()` method that determines what that object will do for its current turn. Running the simulation a single step by pressing the "Act" button amounts to calling `act()` on every single actor–giving each one its single turn. Running the simulation by pressing the "Run" button amounts to running turn after turn after turn, where each turn consists of calling `act()` on every single actor. The machinery for calling `act()` and managing all the actors is all built into the micro-world, so all you have to do is say what a given actor does on its turn–you have to write the `act()` method for the `Greep` class.

Because `Greep` is a subclass of `Alien`, let's see what that class provides.

A.3.1 The `Alien` Class

The `Alien` class, by a stroke of luck, includes a number of methods that are similar to `Jeroo` methods, plus a few more. You can use any of these inside your `Greep` class. Also, Greeps understand the same directions that jeroos do: `NORTH`, `SOUTH`, `EAST`, and `WEST` for compass directions, and `LEFT`, `RIGHT`, `AHEAD`, and `HERE` for relative directions.

Method	Purpose
`hop()`	Hop one space ahead. If the Greep jumps into the water, it drowns. Note that Greeps *can* occupy the same location (be next to each other), so water is the only hazard.
`turn(relativeDirection)`	Turn in the indicated direction [`LEFT` or `RIGHT`, just like jeroos; `turn(AHEAD)` and `turn(HERE)` are meaningless]
`hasTomato()`	Is this Greep carrying a tomato?
`isClear(relativeDirection)`	Is there a clear space in the indicated direction? A clear space contains no other objects. [`isClear(HERE)` is meaningless]
`isFacing(compassDirection)`	Is this Greep facing in the indicated direction?
`sees(objectType, relativeDirection)`	Is there an object of the specified typein the indicated direction? For example, `this.sees(Greep.class, LEFT)` or `this.sees(Tomato.class, AHEAD)`.
`loadTomato(relativeDirection)`	Load a tomato onto a neighboring Greep in the indicated direction. Nothing happens if there is no tomato here, or if there is no Greep in the indicated direction, or if the Greep is already holding a tomato of its own. Note that, since Greeps can be in the same cell at the same time, it is possible to `loadTomato(HERE)` if there is another Greep in the same spot--but you cannot load a tomato on yourself.
`unloadTomato()`	If the Greep is currently carrying a tomato and is at the ship, the Greep can unload the cargo onto the ship, which counts as a "score". If the Greep is not at the ship, it will just drop the tomato and, splat, it's gone. Thankfully, Greeps don't need any help dropping off a tomato and can do this by themselves.
`paint()`	Drops a blob of paint at the Greep's current location, so the Greep can leave a trail. Paint gradually disappears over 50 turns.
`getDirection()`	Returns the current `CompassDirection` in which the Greep is facing.
`getShipDirection()`	Returns the `CompassDirection` in which the Greep's ship lies. Note that the ship is not necessarily in a straight line this way--it may be off in a diagonal direction instead, but the closest cardinal compass direction is returned.
`turnTowardShip()`	Turns the Greep so that it is facing in the direction (NORTH, SOUTH, EAST, or WEST) closest to where the ship lies. Note that the ship is not necessarily in a straight line this way--it may be off in a diagonal direction instead, but the closest cardinal compass direction is where the Greep will turn.

Figure A.1: `Alien` Methods

You can use any of these methods in constructing your solution. At the same time, however, there are several restrictions you must follow:

- **Only one move per turn.** Your Greep can only `hop()` once per turn (which is why there is no `hop(n)` method). "Once per turn" means only once per call to act().
- **Greeps cannot communicate directly to each other.** That is: no field accesses or method calls to other `Greep` objects are allowed. (Greeps can communicate indirectly via the paint spots on the ground.)
- **No long vision.** You are allowed to look at the world only using the Greep's `sees()` method, and may not look farther. That is: no use of methods similar to getObjects...() or getOneObject...() are permitted.
- **No creation of objects.** You are not allowed to create any scenario objects (instances of any `Actor` classes, such as `Greep` or `Tomato`). Greeps have no magic powers–they cannot create things out of nothing.
- **Only boolean fields.** You can add boolean fields to your `Greep` class, but cannot add other kinds of fields. In particular, you cannot "remember" coordinate positions

for key features, like the ship or tomato patches. Only yes-or-no values can be kept in fields (other than test classes, of course).

- **No teleporting.** Methods from Actor that cheat normal movement (such as `setGridLocation()`) may not be used.

A.4 Generating Random Numbers

Although it is not necessary in this assignment, sometimes you might find it helps to make random choices so your Greeps are a little more independent and varied. For example, sometimes you might want a Greep to wander around randomly looking for tomatoes. Or maybe when it finds water, sometimes you want it to turn right and other times turn left. Java provides a built-in class called Random for generating random numbers, and the Sofia library includes a special version of this class that is helpful for beginners. To use this class, add the following import statement at the top of your file:

```
import sofia.util.Random;
```

The `Random` class provides a method called `generator()` to get an object that represents a random number generator. Here, we only need to deal with generating random integers, and the generator provides a method that is very useful for this purpose. You can use it like this:

```
int value = Random.generator().nextInt(4); //generate a random number from 0-3
```

The generator provides a method called `nextInt()` that generates a random integer. It takes a single parameter, which is an upper limit. When you provide this upper limit, the `nextInt()` method will generate a number from 0 (inclusive) up to (but not including) the upper limit.

So, for example, if you want to generate a number from 0-99, you would call nextInt(100). Suppose that you would like to perform some action 15% of the time. You could do this:

```
if (Random.generator().nextInt(100) < 15)
{
...
}
```

Here, the call to `nextInt()` will produce a number from 0-99 (that is 100 possible values), and the if statement will execute its true branch if the generated number is in the range 0-14 (which is 15 possible values, or 15% of the time).

A.5 Testing Random Behaviors

Random behaviors are great for chance-based events. But random behaviors also make software testing hard. When you add random behavior to your Greep and then want to test it, what will your test case do? Suppose you want your Greep to turn left at the water's edge half the time, and right the other half. If you write a test case where the Greep sees the water ahead, it might turn left ... or it might not. How can you write tests for that???

The answer is simple: the `Random` class helps you. Consider the following code sequence, which generates three random numbers less than 100:

```
int x = Random.generator().nextInt(100);
int y = Random.generator().nextInt(100);
int z = Random.generator().nextInt(100);
```

It would be difficult to write a test case that used this code, since you have no way of controlling what values end up in x, y, and z. For test cases, however, the `Random` class provides a special method called **setNextInts()** that lets you control what numbers are generated for testing purposes. You use it like this:

```
// In your test case, do this:
Random.setNextInts(40, 50, 60);


// In the code you are testing, this happens:
int x = Random.generator().nextInt(100);
int y = Random.generator().nextInt(100);
int z = Random.generator().nextInt(100);


// You know x will get the value 40, while y is 50, and z is 60
```

So, when you are testing behaviors that are random, you can force the actions to be predictable just by saying in your test cases what sequence of values you want the random number generator to produce. Outside of test cases, the generator will produce a truly (pseudo-)random sequence, but inside your test cases, the numbers will be completely determined by you.

A.6 Comments on Design

As in other assignments, you will be graded in part on the design and readability of your solution using the posted grading criteria, so consider these factors when devising and naming your methods. The Program Grading Rubric (same as on Programs 1 and 2) describes the

grading criteria. Note that a portion of your grade will be based on your approach to testing your solution.

A.7 A Contest

At a minimum, your solution must be able to retrieve one tomato within the turn limit provided by the Earth class. However, your goal is to retrieve as many tomatoes as possible before time runs out.

So, for for this assignment, all submissions that are completed on time will be entered in a tournament. Each submission will be played off on a set of new maps that you haven't seen, and scored based on the total number of tomatoes harvested across all of these maps. Awards for the tournament winners will be given in class. Good luck!

A.8 Submitting Your Solution

All program assignments are submitted to Web-CAT. Use the Controls-¿Submit... menu command to submit your work.

Appendix B

Battleship! Assignment

This appendix contains the full assignment write up for Battleship! assignment.

B.1 Goal

In this program assignment, you will implement a computer "A.I." player for the classic board game, Battleship!. Hopefully, you're already familiar with the rules for this game. As with the last assignment, the minimum requirements for your game strategy are very basic, but you also have the freedom to adapt or expand on the basic ideas presented here to build the most effective strategy you desire.

B.2 Quick Game Summary

Battleship! is a two-player game where each player starts out with a fleet of 5 ships of different sizes. Before beginning play, each player positions his or her 5 ships on a 10x10 map. During play, players take turns firing at cells on their opponent's map, scoring either a "hit" (if they strike one of the ships), or a "miss". Play continues until one of the players has "sunk" all 5 ships in their opponent's fleet.

There is lots of information on Battleship! available on the web. You can read the authentic Hasbro instructions for the game, and Wikipedia has an article with some of the history and other details. There's plenty more to find for those who wish to explore.

B.3 Learning the Classes

In this assignment, you will be working with a brand new package of classes tailored just for this assignment. Part of this project is *learning how to learn about other classes from their documentation*, so that you can become familiar with looking up reference material on library classes you wish to use. You can find all the reference material here:

`sofia.battleship` Javadoc

Among the classes, five to pay particular attention to are:

BattleshipStrategy

This isn't a class, it is an interface. An interface contains only method declarations (and possibly some constant declarations). When a class implements an interface, it is required to provide definitions for all of the methods contained in that interface. Your solution must implement this interface, so it defines the methods that you must write at a minimum (you can add more of your own, of course).

Board

This class represents one player's 10x10 grid containing that player's fleet, together with which cells have been fired on and whether each attacked cell was a hit or a miss.

GameState

This class represents the entire game, including both players' boards. This is the kind of object your strategy receives on each turn, so that it can plan its next move. Your strategy can use `getMyBoard()` to get at your own fleet's grid, where you can see all your own ships. Your strategy can also use `getOpponentsBoard()` to get at your opponent's board, although you cannot see where the ships are on your opponent's board–only what cells you have hit or missed, which ships you have sunk so far, and whether your last turn successfully sunk a ship.

ShipPlacementMove

This class represents the initial game action of placing your fleet on the board. Your strategy has to determine its own ship placement, and your strategy cannot use the same placement in every game it plays.

CallShotMove

This class represents one "firing" move by your strategy, so your strategy will be asked for this in every round of the game. Your strategy will create and return such an object to indicate its choice of what location it wants to fire at next.

B.4 Structuring Your Solution

Download the starting scenario for this assignment: program4.zip.

Your task is to write one main class: your player strategy. Actually, you can write multiple strategies if you want. You might want to start by writing a simpler strategy just to get it to work, and later create a new class to embellish or refine your ideas–then you can play games where these strategies play against each other, to see if your own ideas work out the way you expect.

Your strategy class can have any name you like, but it must **implement the BattleshipStrategy interface**. This means it should include "`implements BattleshipStrategy`" after the class name (instead of "`extends`"), and that it must provide all methods listed in the documentation for `BattleshipStrategy`.

Behaviorally, there are only three specific requirements for your strategy:

- When asked for a ship placement, your strategy must be able to return a ship placement move that positions all 5 ships of your fleet in legal positions on the board.
- Your strategy cannot use the same ship placement in every game. In other words, you cannot simply "hard-code" in one ship placement you devised yourself, and have your strategy use that same placement every time.
- Your strategy must be able to generate legal firing moves on every turn, eventually firing on all 100 locations on the opponent's board.

Beyond that, you are free to use any strategy you like to play the Battleship! game.

Note: the `BattleshipStrategy` interface requires your strategy to provide a method called `getName()`. This is the name you'll see displayed on-screen when your strategy is playing. Please include your PID in the string returned by this method so that your strategy can be easily identified in tournaments. Otherwise, you can include anything else (except for inappropriate content) in the result of `getName()`, and if you write more than one strategy, you can give them different names.

B.4.1 The `BattleshipGame` Class

The starting scenario for this project includes a world class called `BattleshipGame`. This class represents a playable version of the game that includes two automated computer players.

Before you turn in your solution, You must edit this class so that its `createMyStrategy()` method creates and returns a new instance of your strategy.

As you develop your solution, you can edit this class how you wish. You can change both `createMyStrategy()` and `createOpponentsStrategy()` to control which strategies play against each other when the game runs. Just press the "Run" button to play a game between these two strategies–"your" strategy will be shown on the bottom, firing at the board on top, while the opponent's strategy will be shown on top, firing against your board at the bottom.

To help you get started, the scenario already includes two very simple game strategies that you can play against:

BruteforceStrategy

This strategy uses a fixed placement of its ships, horizontally across the first five rows of the board, flush against the lefthand side. It fires on its opponent's (0, 0) cell first, and works horizontally across the first row, and then left-to-right across the second row, and so on, methodically firing on every cell in its opponent's board.

RandomStrategy

This strategy uses a fixed placement of its ships, vertically in alternating rows. It fires on randomly selected positions on its opponent's board that it has not tried before, and continues selecting new random firing locations on every turn until it has fired on every location.

Admittedly, these two strategies aren't the best opponents–in fact, they don't even meet all of the requirements for this assignment. But they are decent "practice dummies" for you to try your strategy out against. The `BattleshipGame` provided in the starting scenario initially plays these two strategies against each other, but you'll need to replace (at least) one of them to see you own strategy in action.

Note that if you right-click on the game board background on the screen to run methods from the `BattleshipGame` strategy, it provides several methods for playing multiple games that are inherited from the `sofia.battleship.Game` class. See the documentation for that class for more details if you wish to run multiple games for a "playoff" of your own.

B.5 For Fun

While this assignment only requires you to implement a "regular" strategy for Battleship! that follows all the rules, the original game is played in such a way that each opponent cannot see the other's board. This opens up the possibility of certain kinds of devious "cheating" at the game.

If you wish, you can also enhance your strategy so that it can take advantage of this cheating. The `BattleshipStrategy` interface requires your strategy to define a method called `canPlayDeviously()`. You can define this method to simply return false if you always want

to play by the rules. If you define this method to return true, however, in games where devious cheating is allowed, your strategy will have a chance to surreptitiously move its ships between turns during the game. Of course, when ships are moved, all "hits" previously scored by the opponent must remain occupied by some ship, and all "misses" previously registered by the opponent must remain vacant of ships. Further, if a ship has been hit a certain number of times before, after moving, it must still occupy the same number of hit locations (i.e., if the carrier has been hit 3 times so far, after it moves, it must still occupy exactly 3 cells where the opponent has made hits).

By moving ships around (but in a way that is consistent with all the previous hits/misses scored by the opponent), your strategy may be able to gain an advantage when cheating is allowed.

B.6 Submitting Your Solution

All program assignments are submitted to Web-CAT. Use the Controls-¿Submit... menu command to submit your work.

B.7 For Even More Fun

As in the last assignment, we will have a class-wide tournament using solutions from this assignment. The winners will be allowed to drop an additional lab score or homework score from their grade computation.

All submissions that are completed on time will be entered in a "no cheating" (non-devious) tournament and played off head-to-head against all other valid submissions, where no ship repositioning is used.

In addition, all player strategies that can play deviously will also be entered into a second "devious" tournament, and played off against all other devious opponent strategies.

First, second, and third place winners in both tournaments will be awarded prizes (although each student is only eligible for one prize for this assignment).

Good luck!