Tournament Wizard:

Simple and Lightweight Software for Running Fencing Tournaments

A Senior Project

Presented to

The Faculty of the Computer Science Department

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

For the Requirements of the Degree

Bachelor of Science in Computer Science

By

Ryan Daley

June 2016

Table of Contents

Introduction

Many sports use technology to assist in officiating and administration. Fencing tournament organizers use specialized software to help determine how their matches occur, and to help them administer a tournament. However, the software most commonly used to run these tournaments has some significant limitations, and is not well-suited for use with small events. As a former President of Cal Poly's Fencing Club, I have had a chance to interact with one popular application for running fencing tournaments a number of times, and made note of its shortcomings. These included issues with operating system compatibility, license transference, and amount of space needed to maintain the database of fencers.

After struggling to cope with the limitations of the existing software, I began to wonder how this software would be used if it was scaled down to fit the level of a local club team or a youth league. The purpose of this project is to create a light-weight alternative to these large scale systems. Designing a system like this would result in an application customized to the needs of smaller events, and could also serve as a basic training tool, familiarizing users to the format of larger fencing tournaments. I also wanted to use this project as an opportunity to explore software design using the Model View Controller paradigm, creating separation between the Graphic User Interface and the underlying logic that is used to maintain the data [5].

Existing Tournament Software

The most commonly used software to run fencing tournaments in the United States is a program called Fencing Time. The software has become very robust over the years, and admittedly does an excellent job at the large scale tournaments that need very precise organization. It has

become the official software the runs tournaments at a national level for the United States Fencing Association (USFA), which is the official body for the sport of fencing here in the United States. Their services have expanded to have premium versions that can maintain tournaments hosted from a server, or over the cloud for more accessibility in the case of tournaments spanning one hundred or more fencers in several events [4].

However, this feature bloat can also be a hindrance for those users who don't need its advanced features. As more features have been added, a licensing fee became necessary to use the standard version of the software. Though a case could be made that such a fee allows for continued expansion, a $99 starting fee, with $49 renewal fees every year can seem a bit steep if you don't plan on running many tournaments in a season. Transferring the license can also be an issue. One of the duties I had as a fencing club President was providing all the equipment needed to run a tournament, including the computer and software. While we had a tournament organizer with access to Fencing Time, getting his license to work on my computer took several bouts of troubleshooting, and involved sticking with an older version of the software instead of the most up to date version. It is exasperating to repeat the process every year when a new President is elected for the club.

Fencing Time also has the ability to maintain a database of registered fencers, which can be initially provided and updated from a CSV file available for download from the USFA website [3]. This provides a way for the software to identify an individual registering for a tournament, which speeds the process when hundreds of athletes are waiting to register. However, it is that large size that makes it impractical, and a smaller tournament has little need for a database

housing hundreds of thousands of possible fencers when the organization might only need to worry about one or two dozen. That database also needs to be updated regularly, as memberships expire or ratings change. In the instance of smaller tournaments, omitting the database and starting with an empty roster would be a more practical approach.

Another important limitation of Fencing Time is simply its operating system compatibility. Fencing Time is only created to work on Windows machines or servers, and while this does account for a large section of computers or laptops in use, other operating systems have been growing in popularity in recent years. The Fencing Time website claims that a Mac port is unlikely to ever happen due to the amount of workload it would involve, and gives a list of suggestions for running the software from a PC virtualization system [4]. This won't work for all users, and even when it does, there will still be the possibility of significant impact on the performance of the product. There is also no indication of how the software will run on a Linux based system, leaving Linux users to do extra research or contact support if they wish to know how to get started.

While not meant to scale up to the national level that Fencing Time has grown to, Tournament Wizard is designed in a way to circumvent these specific problems, while focusing on a smaller tournament size within the environment of local clubs.

Requirements

In the process of designing the requirements for Tournament Wizard, one of the initial tasks was to research the official structure of the standard tournament. As a fencer, I had an understanding of the general flow of a tournament, but never really needed to know the details. For example,

when sorting the fencers into their initial seeding, I was unsure of what exactly the official method of determining where fencers were placed. Research revealed to me that this could be somewhat subjective, based on anything from past performances, points scored in a league thus far, or even the discretion of tournament organizers [2]. Therefore, I felt free to design this initial seeding process in a way that seeded competitors based on their listed USFA rating, while having fencers with equal ratings to be in an arbitrary order.

A process I used to gather requirements that was fairly new to me was writing a User Manual before doing any development work, in order to place myself in the mindset of a user and what they would desire in the software. This was an interesting experience, as I always considered documentation like that, which is made more for the benefit of the user, to be one of the final tasks of development, once most of the software was complete. Having the User Manual written as part of the design process allowed me to incorporate elements of the GUI from the start, such as buttons for clearing screens or the ability to save results as a file. I also had hoped to use the Manual as documentation for myself as well, updating it as more functionality was added to the software as development continued. However, this began to fall to the wayside as I got more heavily into the development, especially when coding became a major task. The latest version of the User Manual is included in the appendix of this paper.

Sketching the GUI was another way that I was able to give myself a concrete idea to strive for during development. Even though I already knew the basic functionality needed to maintain a fencing tournament, having a layout of where each of the basic tasks to be carried out helped immensely with the design. In particular, one of the worries I had was how I was going to

display the bracket structure used for the direct elimination rounds of the tournament. Some initial research online showed me a couple layouts for similar issues, and after incorporating some of those ideas into my own design and putting them on paper, the outcome was satisfying to look at, and I believe it is one that allows a user to easily proceed through that phase of the software without much explanation. It was interesting going through development with so much planning and design completed ahead of time, as it gave me a true appreciation for the process. So much of the curriculum at Cal Poly revolves around the actual development of code, that it can be hard for students to really gain an appreciation for actual Software Engineering techniques once they are introduced to them later on in their college careers. While I don't know if I would continue to design software in the exact same way I did with Tournament Wizard, I feel that the experience of sketching out the basic GUI is a useful tool for requirements gathering that I will continue to use in the future.

One thing I will note about gathering the requirements for Tournament Wizard is that the process never really seemed to end. It would happen fairly often that I would think of a useful new method or view screen while I was beyond the design phase of development. I especially became more lax in this as development went further, as any redesign work would have been exponentially harder to complete within the time constraint the farther into development I was. While I tried not to let it happen too often, I found it could be an interesting teaching experience for how compromise would work in larger scale software systems, as development teams iron out requirement issues with the clients they are developing for.

Design

When designing the structure of Tournament Wizard, the conceptual software was initially divided into two components; the tournament logic classes and the GUI. While the classes responsible for the tournament logic in the background were based on personal design preferences, I decided to base some of the GUI design decisions off of similar functionality from the existing Fencing Time software. I also had concrete design processes for both the GUI aspects and the underlying logic aspects of the software, and that served as a solid basis for the continuing into the main development phase.

Class Design Process

The first step taken in designing the logic for the software was listing out functionality that would be needed, and using that list to create a basic class diagram structure to illustrate what dependencies would be present. This helped iron out where each piece of functionality would be included, as well as what each class would be responsible for. The class diagram document is included in Appendix 1 of this paper.

After the class diagram document was complete, a list of methods was created and assigned to classes within the structure. With this list, Javadocs were written for each method to outline what each of the parameters passed in to the method would be, as well as what the return value of the method would be. This created a solid outline of how the functionality would be called from within the logic structure. With the Javadocs done, the final step in designing the basic tournament logic was to write pseudocode for each method, outlining about how long each of the classes would be and giving them a basic Java structure. This was especially helpful in the development phase, as it allowed for a fairly simple translation into actual Java code, instead of relying on design structures maintained from short term memory.

GUI Design Process

While a major part of the GUI design was completed by sketching out what each screen would look like, refinements could still be made once the tournament logic was farther along in development. Final sketches of the GUI were produced after the initial development of the tournament logic, and made it easier to plan out how interactions would occur between the GUI classes and the logic classes. Furthermore, the GUI was sketched with some of the aspects of Fencing Time in mind. In particular, the method of using grids to record the scores of pools largely took its inspiration from the Fencing Time software. The simple and straightforward use of a grid helped make the overall system more concrete during development, and was helpful for mentally mapping out how the classes responsible for tournament logic would interact with the views that the user would interface with. The design also allows for easier translation of scores from a physical score sheet, as they are typically recorded on paper using a grid format as well, meaning the user just needs to copy the numbers in the order they appear on the physical score sheet into the grid in the GUI.

Tournament Wizard was designed with the Model View Controller scheme in mind, and that is reflected in the way the GUI classes are laid out. Each screen the user encounters is meant to encompass one phase of a fencing tournament, with the view held in each screen being easily updatable based on calls from a Controller class. The GUI classes had very little functionality to them, as they were designed to send their action commands directly to the Controller class, which would react and alter the data accordingly before notifying the correct GUI element of the actions taken [5].

Design High Level Explanation

The design for the tournament logic is split into several classes, including phases of the tournament, individual fencers, and one class responsible for calling on the functionality of the individual classes. The Fencer class represents individual competitors, and most of the other classes contain lists of fencers. The Pool class contains a list of fencers assigned to that Pool, and maintains an NxN grid of integers where N is the number of fencers in the Pool. The EliminationBout class is used to maintain a single matchup within the bracket structure of a tournament, and has two Fencers assigned to it, and also keeps track of another EliminationBout instance that the winning Fencer advances to. Finally, the Event class maintains the entire list of fencers registered for the tournament, creates a list of pools and assigns the fencers to that list when prompted, and creates the bracket structure, which is represented as a list, where each individual item is its own list of EliminationBouts. The Controller class is then able to call the functionality from this Event class based on events performed.

For the GUI, each phase of the tournament has a frame to represent it, and the software is designed to flow from one frame to the next to avoid having too many windows open. The initial registration screen (Fig. 1 in the User Manual) has a box to display all competitors registered so far, and buttons for adding fencers, either in bulk through a csv file selected through the standard Java file chooser, or individually through a pop up dialog box with fields to fill in (Fig. 3 in the User Manual). Proceeding brings up another dialog which prompts the user for the number of pools to create (Fig. 4 and 5 in User Manual). The pool screen presents a tab layout, each tab containing a grid to represent a pool. Each competitor in the pool is represented by one row and one column in the grid, a single cell represents how many points the Fencer in that row scored against the fencer in that column (Fig. 7 in the User Manual). The final main screen

displayed to the user is the bracket screen, where each matchup is represented by the Fencers involved, and a dialog box to choose which one wins and advances to the next round (Fig. 10 in the User Manual). Once the final bout is decided, the user is able to view a results screen, and has the option to save the results of the tournament to a text file (Fig. 11 in the User Manual).

Design Tradeoffs

While designing Tournament Wizard, several aspects of the software could have been implemented in several ways. A prominent example was how the software would record the results of the pool, and use it to sort fencers into their seedings once pools were completed. In this instance, I decided to include two fields in the Fencer class to represent pool win percentage and indicator, allowing for easier sorting. The indicator value is actually a fencing term, and refers to the number of touches a fencer scored in pools minus the number of touches received by a fencer in pools. By overriding the compareTo method of the Fencer class, and having it compare the stats of two Fencers, the seedings could be generated by simply calling the sort method of the Collections class on a list of competitors that is generated once pools are complete. In my experience, this was worth the extra design and code time it took to add those fields and methods to the Fencer class, as well as ways to alter them from the Pool class while results are being generated.

While designing the Fencer class, one of the main points of contention was what to pass into the constructor to create the fencer. I had initially designed it to just take in a string for each field it needed to fill, but in the end, I made the decision to make the constructor take in one string of comma separated values. Even though Java allows for an Object to have multiple different constructors, this decision was made thinking that the resulting code would be cleaner and simpler to understand. Ultimately, this decision turned out to be somewhat regrettable, as the

extra manipulation of data took up more time while coding. However, designing the software to be easier to maintain in the event of future development is a good practice that I hope to maintain in the future in more efficient ways.

While designing the EliminationBout class, I had initially planned to just store the competitors in a list and assume that no more than two competitors would be added to the bout. However, I viewed this as poor practice, and having to constantly reference the get method of the lists to access the Fencers would be unintuitive and repetitive. Instead, I chose to use hardcoded fields in the EliminationBout class to hold two Fencers (Fencer1 and Fencer2). This prevents the bug I was initially worried about, as the class will never have more than two Fencers assigned to it. The easier access to the Fencer fields, as well as the lower possibility of a bug being introduced this way is what led me to deciding on this design aspect.

One of the largest design decisions I ended up encountering was how to make the rows and columns of each grid shown in the pool phase unique. I had initially thought of putting basic labels above and to the side of the grid, but inserting each label through code and setting up the spacing would be a laborious task, and the names above the grid would likely run into issues with spacing unless they were rotated so the words ran vertically. I also toyed with the idea of listing the Fencers in the pool in order of what row or column they were off to the side, but I still felt like that would be too easy for a user to misread. I eventually settled on adding an extra row and column to the GUI view only, that would contain the name of the Fencer located on that line. This did mean that there was a slight difference in how the grid in the view was indexed compared to how the grid in the logic was indexed. However, this still seemed to be the ideal

choice, as it was relatively simple to map the grid index to the logic index, as subtracting one from both the row index and the column index of a cell in the view would map to the equivalent cell in the logic. This was a small but understandable price to pay for a feature that would greatly assist the usability of the software for the user.

Implementation Process

The implementation process for Tournament Wizard was fairly simple. Looking at the design documents, implementation started with the simplest classes with the fewest dependencies, and worked up from there. This meant that the Fencer class was the first to be implemented, as it had no reliance on other classes in the design, and only used simple Java structures. Once that class was finished, the Pool and EliminationBout classes could be implemented, as they only relied on the Fencer class, but were a bit more complicated in the functionality they provided. The final logic class to be implemented was the Event class, as it mainly served as the glue to hold all the others together, and was responsible for the major actions involved in using the software to run a tournament. The Event class actually began its development while the Pool and EliminationBout classes were still being developed as well, but only so that that the Event class development could progress to reflect when development of certain functionality was finished in one of the other classes. Even once those classes were done being implemented, the Event class still had a bit more progress to be made on it, as it needed to support the ability to have all the modules work together.

Implementing the GUI classes had a bit of a different start. Using the User Interface Editor in the NetBeans IDE, I translated the GUI sketches into Swing JFrames and Dialogs, and made sure they were laid out in an acceptable fashion. I then created a Controller class that would be the

main driving class of the entire program, not unlike the purpose that the Event class serves for the internal logic. This step was where I had some deviation from the standard MVC paradigm, as a Controller ideally has no interaction with the GUI, apart from blindly listening to events from it [5]. However, my Controller class needed to have access to an instance of each of the created views, so that it would be able to start and stop them based on commands it received. From there, the development progressed through the views in order of the progression of the actual tournament as follows: registering Fencers for a tournament, selecting the number of pools and running those pools, proceeding through the direct elimination phase by choosing the winner of each bout, and viewing or saving the results. In each of those phases, the necessary views had methods added to them that allowed them to update the screen as they were notified, and also were set to send any commands they might receive back to the Controller class. In turn, the Controller class would then have functionality added to its Event Listener class that would allow it to interpret the commands sent to it through the views from the user, and implement the functionality needed to process those commands.

Testing

One of the earliest aspects of testing that was used in the development of Tournament Wizard was the creation of a simple test file that would contain specific data and expect a certain outcome. This file was actually created early on in the design phase, and was used to ensure that the design of the system matched the requirements that had been listed. As the design process progressed further, the test data was used along with a sort of paper prototype, and was run through the written pseudocode by hand, calculating all the values created by the design and listing them to ensure correctness. After initial development was completed on a module, the

data from that test file could then be input into the system, and the result could be compared against what the expected state would be based on the paper prototype run.

In order to automate the testing process, JUnit tests were created for the logic classes to verify that their methods were working as expected. These were implemented in the same order that the logic classes were implemented, to fit in with the dependencies necessary, as it would have been difficult to test functionality that depended on other classes, like the Pool class's functionality to store the results of the pool bouts in the competing Fencers, without having already verified the correctness of the dependent class.

A test driver was also created once all of the logic classes had been implemented. This driver used the earlier mentioned sample input and ran it through the phases of a tournament from beginning to end, outputting the results from each phase for the user to read. While this served to verify functionality while there was no GUI to work with, it became obsolete once the GUI was in a working condition. Therefore, the driver was altered to become another JUnit test, eliminating the need for the user to read the output and verify that it was correct, making it easier to test if changes to one part of a class affected the system as a whole.

A final JUnit test was created to test functionality of the GUI classes. Since the GUI classes had no actual output, and it would be difficult to test them one at a time due to the serial nature they were designed in, this test was somewhat similar to the previous test driver in that it just ran through a sample run of data. Instead of going through the tedious job of checking every aspect

of the view, it would instead only check which screen of the view was visible, to ensure that the Controller class was able to transition between phases of a tournament in the correct order.

User Feedback

Late in the testing cycle, I decided to get some user feedback on the current version of Tournament Wizard. I brought a working demo to a practice with the Cal Poly Fencing Club to show to some of my club mates. Ideally, I wanted to get a broad spectrum of users, ranging from those who had experience with tournament software like Fencing Time, or those whose tournament knowledge limited to participation and not actual organization. Doing this provided invaluable feedback, which served for both the improvement of Tournament Wizard, and confirmation of its viability. In particular, feedback about the Elimination Bracket phase of the software, where the user is meant to choose which competitor advances by selecting from a drop down box for each bout, tested rather well, and was observed to be easy to understand for both experienced tournament organizers and new users.

Some new features of Tournament Wizard were added as a result of this user feedback, as some users expressed desire for things to be done in a particular way, or for the software to have extra functionality. The result screen that appears in between the Pool phase and the Elimination Bracket is one example. Several of the users queried that having a screen to show how each competitor fared in the Pool round not only served to help validate the software's results, it also served as a useful function for the tournament organizers to show the results to the competing fencers. The users giving feedback expressed that due to their own experience competing in tournaments, they knew the competing fencers would like to see how they fared compared to others, so they could have some level of preparation for the next phase. They also expressed a

desire for a minor change to the way each EliminationBout is portrayed on screen, by requesting that each Fencer have their seeding displayed alongside their name. This small change allowed viewers to make predictions of who would be proceeding, and better portray any upsets where a lower seeded Fencer defeated a higher seeded competitor.

While it was not intended to fill the role of bug fixing, the user testing also helped reveal some major flaws in the initial builds of Tournament Wizard. The biggest one was that scores from the Pool round were not being correctly transferred into the software, so the competing Fencers were not being seeded correctly. This bug was actually found while implementing the previously mentioned Pool results screen, as it was responsible for showing the stats that were not being recorded correctly. The other flaw, while not as significant, came from the way a Fencer object was designed when created as a bye to fill in open slots in the first round of the Elimination Bracket phase. When proceeding to the final results screen, all the byes in the first round would be included in the list of results, and there was even a possibility of a fencer placing lower than the byes if they lost within the first bout and performed poorly in the Pool round. While this was not causing incorrect results, it was better to have the results ignore any byes while reporting the results anyway.

Problems Encountered

Several problems were encountered in the development of Tournament Wizard. One problem encountered that showed deviation from the initial design documents was how the bracket structure appeared to the users. Traditionally when a tournament bracket is displayed, each round collapses toward the center. Within the short scope of the initial development of the software, getting the functionality of running a tournament was more important than having the

bracket displayed in the same way as other brackets. Having to draw all the lines to connect the bouts would take a lot of data manipulation, and keeping track of where each view is. It would also cause problems whenever the view was updated, because a Fencer with a longer name advancing to the next round might shift some of the later rounds, since each round is contained in a column. There also needed to be a simple way to choose who won between the two Fencers. Fencing Time does this by listing all the bouts that are readily waiting for results in a sidebar, and the winner is decided when the user enters the scores. I specifically wanted to avoid a solution similar to that, as deciding the winner from a different point of the screen seemed likely to cause confusion to the user. As mentioned in the Design section, the decision that was arrived at was to add a drop down dialog element to each individual bout view within the bracket screen. This dialog would populate with the competing fencers as they are decided, and this design allowed the user to choose the winner directly from the position in the bracket for more clarity.

Pool Screen Issues

Some of the most significant errors discovered were in the way the Pool screens report the results of the pools back to the Event structures. During development, initial testing of the grids in the Pool screen resulted in exceptions being thrown whenever Validate Scores was clicked while the current cell in focus had input entered by the user. At first, it appeared this was caused by the software recognizing empty cells as null values instead of empty Strings. The first fix for this was to make the software recognize empty cells to contain a score value of zero. While this strategy fixed errors caused from exceptions when any cell was empty, later testing found that the error would still occur when the cell in focus was the last score entered by the user. In order to rectify the issue, a few lines of code were added that shifted the focus of the grid to be on some arbitrary cell that wasn't meant to hold any values.

Later on in testing, another issue found on the Pool view screen was discovered once an extra screen was added so the user could view the order the fencers had been seeded, as well as the indicators of each of the fencers. Upon completion of the pool, the result screen would show that every competitor won none of their pool bouts, and had no value entered for their indicator. The software would then proceed, and the Fencers would just be seeded in the same order as their initial seeding based on their rating. After inspecting the source code, it was discovered that this was caused by a simple omission of a method call that is responsible for compiling the results of a single pool, and inserting them into each individual Fencer. Once that method call was inserted into the correct location, the results after the Pool phase reflected the correct values for the data that was input.

Subversion Issues

One of the more unique issues that came up in the project happened right before development began as I was attempting to set up the repository that would eventually hold all of the source code and the NetBeans project. While registering for the service and creating the repository were relatively simple tasks, the act of organizing the file structure of the repository to contain the directories that are standard of Subversion repositories turned out to be much more difficult. This was a result of difficulties using a client to remotely log into the repository from the command line. The Secure Shell Client installed on my computer did not seem to be compatible with calls to a Subversion repository, so it was rejecting any commits made though the command line, although pull requests were still allowed, as that could be done without an actual remote login. This issue had to be resolved with assistance from Dr. Dalbey, who was able to make the changes and add the file directories to the repository from his own machine.

There was a second issue stemming from problems with the Subversion repository. While remotely logging in through the command line presented a problem, doing checkouts and commits through the Subversion client of the NetBeans IDE proved to work well on my machine. However, this wasn't a universal solution, as Dr. Dalbey was unable to make checkouts or commits from NetBeans using the same settings. This appeared to be caused by version difference on the side of the NetBeans environment. My machine was running version 8 of NetBeans, while Dr. Dalbey was using a build from version 7. While he was able to find settings that would allow him to make checkouts and commits as well, it was interesting to note how critical of an error could result from using different versions a development environment, and worth noting in case of any future development.

Miscellaneous Issues

One aspect of the development process that caused several problems was whenever it was discovered that functionality was missing, or could be implemented in a better way. It would commonly happen that development on a certain module would reveal that adding some extra functionality to an already completed class would help make the process of writing the current class more efficient. This generally happened in fairly simple ways, with a leading example being the creation of several different methods in the Fencer class to turn an instance of a Fencer into some kind of String representation. The first one was created because the only String representation of a Fencer needed in the initial design was just printing the names of the fencers to put in the main registration screen and the Pool grid screen. However, as development continued, and feedback indicated users would like to see more of the results of different phases, extra information from each Fencer needed to be displayed. Thus, for each of those scenarios, instead of creating code to constantly call for the needed information from each Fencer object within the calling class, a separate String creating method was created within the Fencer class

that would just be called once for each Fencer to create the String. The issue of discovering more functionality than was currently designed was not resolved in any specific way, due to the varying nature of the new features being added, but it did help emphasize how difficult it can be do create a complete design that work perfectly before the development starts, and be able to stick to it throughout the entire development cycle.

Results

Currently Tournament Wizard is able to successfully handle the test data created, and is thus suitable for running the small scale or local tournaments it was designed for. I believe that the software currently fits the problem statement expressed earlier, and is a lightweight program that has no dependency on any kind of massive database to keep track of Fencers. Being designed as a Java application, Tournament Wizard will also work regardless of the operating system being used, and has been tested on Windows, Linux, and Mac environments. In order to get a quantitative measure of the success of this project, it's necessary to compare the project state to the goals set out in the beginning of the paper. Tournament Wizard is a lightweight file in comparison to Fencing Time, has no reliance on an underlying database, and is compatible with the three major operating systems. So Tournament Wizard is currently accomplishing all that it set out to do at the beginning of the project.

Future Work

As with any software system, there is more that can be done with Tournament Wizard given another development cycle or more time in development. Aspects of the GUI could look a little bit cleaner or have a little bit more functionality to them. In particular, the Elimination Bracket screen could have the lines drawn between bouts that depict a visual representation of where

bouts lead into the next round. While the system is clear enough for the small scale it is currently intended for, such a feature would allow for Tournament Wizard to be more easily scaled up if that became a goal for future development.

Beyond more development time, subsequent development cycles could allow Tournament Wizard to begin to interact with some services beyond its own functionality. One feature that could be implemented would be a more descriptive results file that contains information about the entire tournament. This would contain the result of each Pool, the seedings that were created, and the results of each bout in the bracket (See Appendix 2). In a version with this functionality, it would be useful to have the bouts of the Elimination Bracket decided by entering the scores of each fencer, so that this could be saved and listed in the results output file. As development of this version continued, with some extra work, the software could even optionally create an output file compatible with the web service commonly used for hosting the registration of fencing tournaments, called AskFRED [1]. The site allows for posting the date of a tournament, allows a competitor to pre-register for attendance, check which other athletes are also pre-registered, and will even allow the tournament organizer to upload a file that has the results, and AskFred will display the results of each round for users to view.

Lessons Learned

One particular lesson of this project that I believe will stick with me the most is the overall value of the Model View Controller design scheme. Developing Tournament Wizard allowed me to observe firsthand the utility of having one class responsible for the overall flow of the entire program, while having core logic separated into its own distinct package. While developing a Java application certainly makes it easy and possible to just create GUI screens with code in each

screen implementing all the functionality, having the Controller class control the flow from one screen to the next meant that data had to all come through this one class, and made a good stopping point while debugging to catch potential errors. Additionally, the design of having the logic be a separate package from the GUI also assisted in debugging, as any errors found to be a result of a logic mistake could be fixed without making many alterations to the GUI classes, if any at all. That separation between the different aspects of the software made debugging and understanding where issues were coming from, and proved to be invaluable to the development process.

Looking back through my experience and development notes, I would say the only thing that really should have been done differently was the allotment of time for working on the project. Earlier in the project, when most of the work was involving design and planning, the time commitment needed was relatively sparse, and I began to let that set the tone for later work. While a schedule had been created to outline when specific aspects of the project needed to be completed, some of the earlier portions could have been completed ahead of schedule with a bit more time commitment. I would stick to the schedule, instead of looking ahead to get more work done early, which really would have benefitted the project once it reached the later, more time intensive phases. It showed me to not get too complacent in what has been accomplished so far, and that continued development should always be beneficial to the overall project.

Acknowledgements

At this point, I would like to give acknowledgement to those who served to help with the development of Tournament Wizard. To John Dalbey, who served as the advisor to this project, and was a major help in multiple aspects of development, from overseeing the design process to

assisting with learning Swing for creating the GUI. To the Cal Poly Fencing Club, for getting me started with the sport all the way back since freshman year, and providing me with the experiences needed for coming up with this project. Within the club, I also wish to acknowledge my fellow sabre fencers, who were willing to be the guinea pigs and provide the user feedback utilized to improve Tournament Wizard. Finally, to the Computer Science faculty, and to the faculty of Cal Poly as a whole, for providing me with the knowledge set for this project, and for preparing me with the skills needed for transitioning beyond the university.

References

1. About FRED, 2001. Retrieved June 6, 2016 from Askfred.net: https://askfred.net/Info/about.php

2. Bhutta, O., ed. *Fencing Rules for Competition*. Vol. 9/1/15. Hanover: Fencing Officials Commission, 2015. Print. PDF form of rulebook available at http://www.usfencing.org/athletes-usa-fencing-rule-book

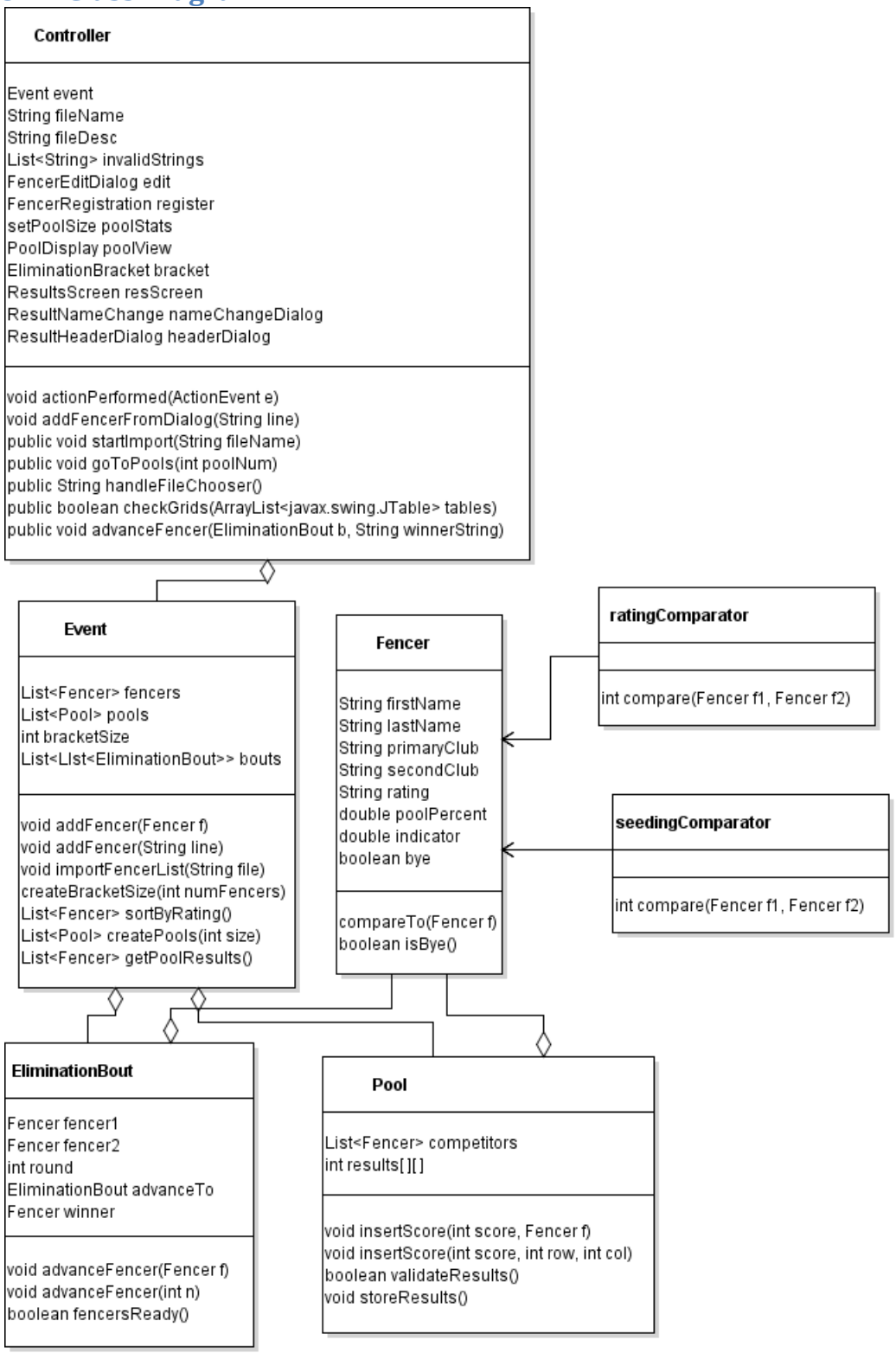
3. Current Member List, 2016. Retrieved June 2, 2016 from USAFencing: http://www.usfencing.org/current-member-list

4. Fencing Time 4.2, 2016. Retrieved June 2, 2016 from Fencing Time: https://www.fencingtime.com/Download

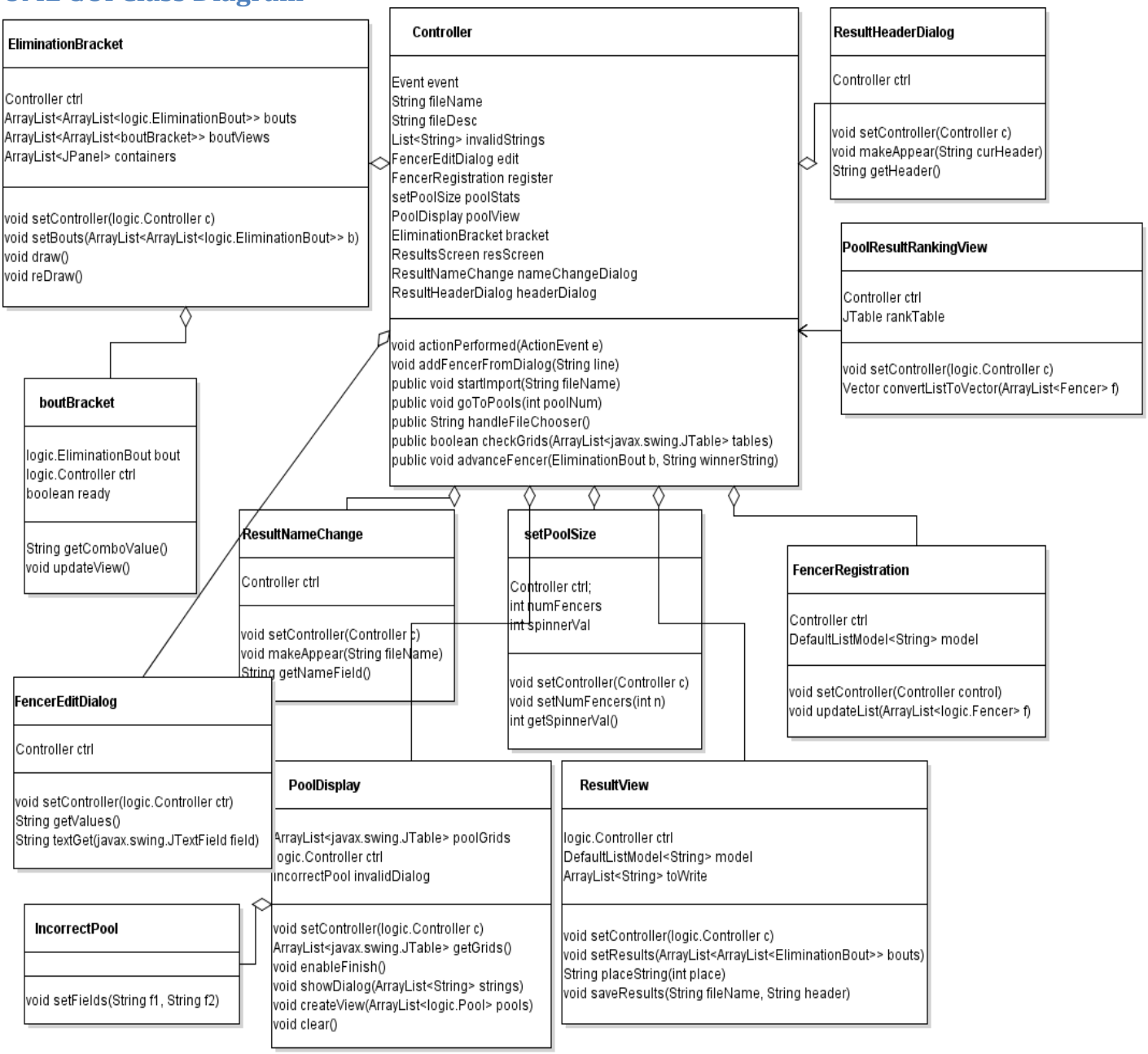
5. Lethbridge, T. C., and Laganière, R. *Object-oriented Software Engineering: Practical Software Development Using UML and Java*. London: McGraw-Hill, 2001. Print.

Appendix 1

UML Class Diagram



UML GUI Class Diagram



Appendix 2

Mockup of Future Result Output File

```
This is a simple mockup of what a more detailed result file
could look like, in addition to the final results being stored.

Pool 1:
            Kent,Clark Todd,Mike  Smith,Bob  Wayne, Bruce
Kent, Clark            X          5          5          5
Todd, Mike             3          X          5          1
Smith, Bob             1          3          X          1
Wayne, Bruce           4          5          5          X

Pool 2:
            Zar, Jack  Holly, Molly    Doe, Jane  Michael, John
Zar, Jack              X          4          5          5
Holly, Molly           5          X          5          5
Doe, Jane              3          3          X          4
Michael, John          4          2          5          X

Seedings
Kent, Clark...(100%)...+7
Holly, Molly...(100%)...+6
Wayne, Bruce...(66%)...+7
Zar, Jack...(66%)...+2
Michael, John...(33%)...-3
Todd, Mike...(33%)...-4
Doe, Jane...(0%)...-5
Smith, Bob...(0%)...-10
```

Tournament Wizard

User Guide

Version 1.0

Ryan Daley

A California Polytechnic State University Senior Project
June 2016

Disclaimer

At the time of this writing, the software detailed in this manual is not meant for tournaments regulated by the United States Fencing Association (USFA), and makes no guarantee results generated through the software will be accepted by officials from the USFA.

The author has made every effort to ensure that the information in this manual is correct as it applies to the software, as it should seeing as the author wrote the software as well. The author disclaims any inaccuracies or omissions that should occur, unless the information left out is something that is promised at a later point in the manual, then it is the author's fault.

Preface

This user manual exists to help guide you through organizing and managing the events that take place in a fencing tournament, using the Tournament Wizard software. However, fencing experience is of course not required, and the steps in this manual can be used to run a fencing tournament. If you feel the need to update yourself on the rules regarding the fencing itself, you can find them at the USFA official website:
http://www.usfencing.org/page/show/695208-rulebook

This guide will go through all of the basic actions that go into using it to run a tournament.

The main features are:

- Starting and naming the tournament
- Adding fencers to the tournament roster
- Establish pools, and track their progress
- Create and maintain elimination brackets
- Display the results for competitors to view

Chapter 1: Starting the Tournament

This section will go over the aspects of setting up the tournament, including:

- Starting the software
- Setting up the event, weapon, and name
- Registering participating fencers
- Choosing the size of pools

Starting the Software

Since the product is written in Java, the only file needed to run it will be the executable jar file, "tourneyWizard.jar". You can run it by either navigating to the file location through a file explorer and double clicking the file, or from the command line by navigating to the directory containing the file and using the following command:

```
java -jar tourneyWizard.jar
```

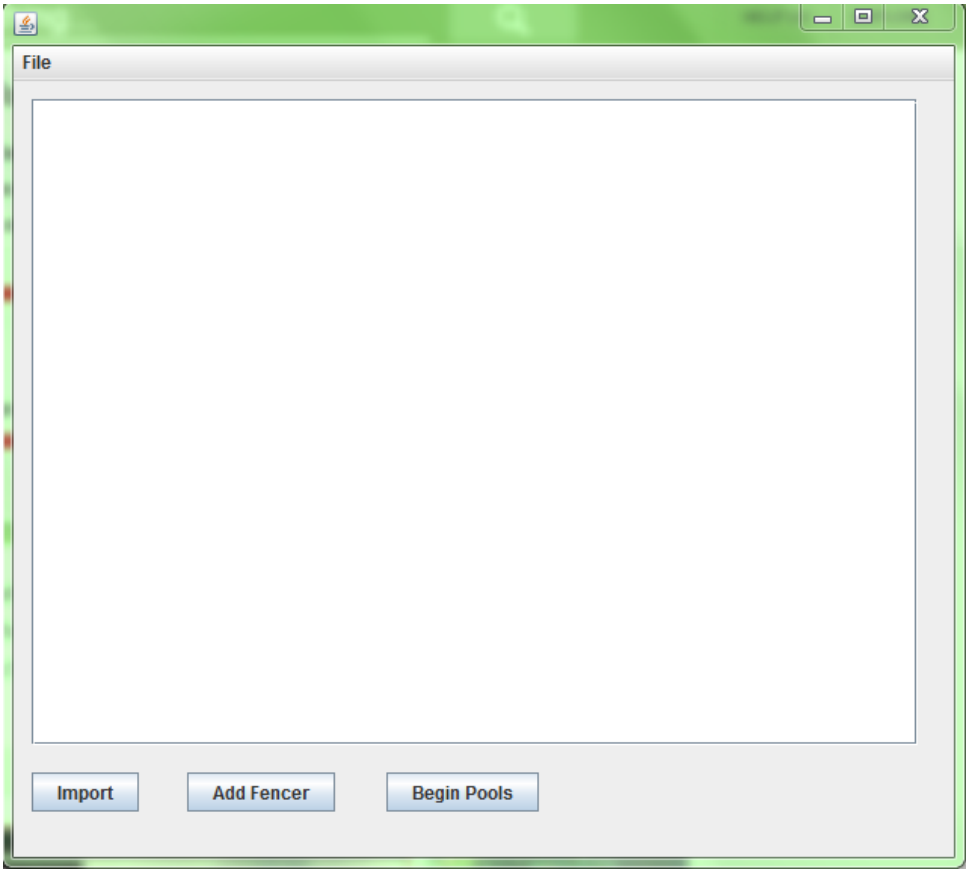


Fig. 1: The starting screen

Entering Fencers to Compete

Fencers registered in an event are the ones who will be competing to see who is best at whacking each other with swords. By default, an event will have no fencers, but there will be two ways to add fencers, either manually, or in bulk from a csv file.

Input File Format

To add fencers from a csv file, you will need to click the "Import" button from the event window. This will open up a file navigation window, and you will need to select the desired file. The format the file should follow is:

```
LastName, FirstName, Club, Rating
```

Where

- LastName and FirstName are the respective names of the fencers
- Club is the club the fencer represents
- Rating is the rating assigned to the fencer through USFA
    - Ratings range from A to E, with A being the highest
    - Fencers without a rating are "Unrated", or U. This is the default

The import option is what you will want to use to quickly enter any fencers who registered for the event prior to the tournament date. This is usually done through a third party web site, and allows fencers to get an idea of what the competition will be like before committing to attending. In many cases, a CSV list of the participating fencers can be obtained from the site used by the tournament organizer, so asking them is the best way to obtain the file.

```
Smith,Bob,Cal Poly Fencing,U
Doe,Jane,UC Davis Fencing,U
Zar,Jack,UC Davis Fencing,D
Todd,Mike,UC Santa Cruz Fencing,E
Holly,Molly,Cal Poly Fencing,D
Michael,John,Cal Poly Fencing,U
Kent,Clark,UC Santa Cruz Fencing,C
Wayne,Bruce,Cal Poly Fencing,U
```

Fig. 2: An example of the format for in input test file. This is the data used in all example screenshots

Note that no quotation marks are needed in order for the file to work properly. However, each of the four values must not be empty. If a fencer does not have a club, they should be listed in the file as Unattached.

Manually Adding Fencers

To manually add a single new fencer, click the "Add Fencer" button on the event window. The resulting dialog window will have the same fields as listed above, and should be filled in with the desired information. You will then click "Add" to add a fencer with the filled in information, or "Cancel" to return to the previous screen.

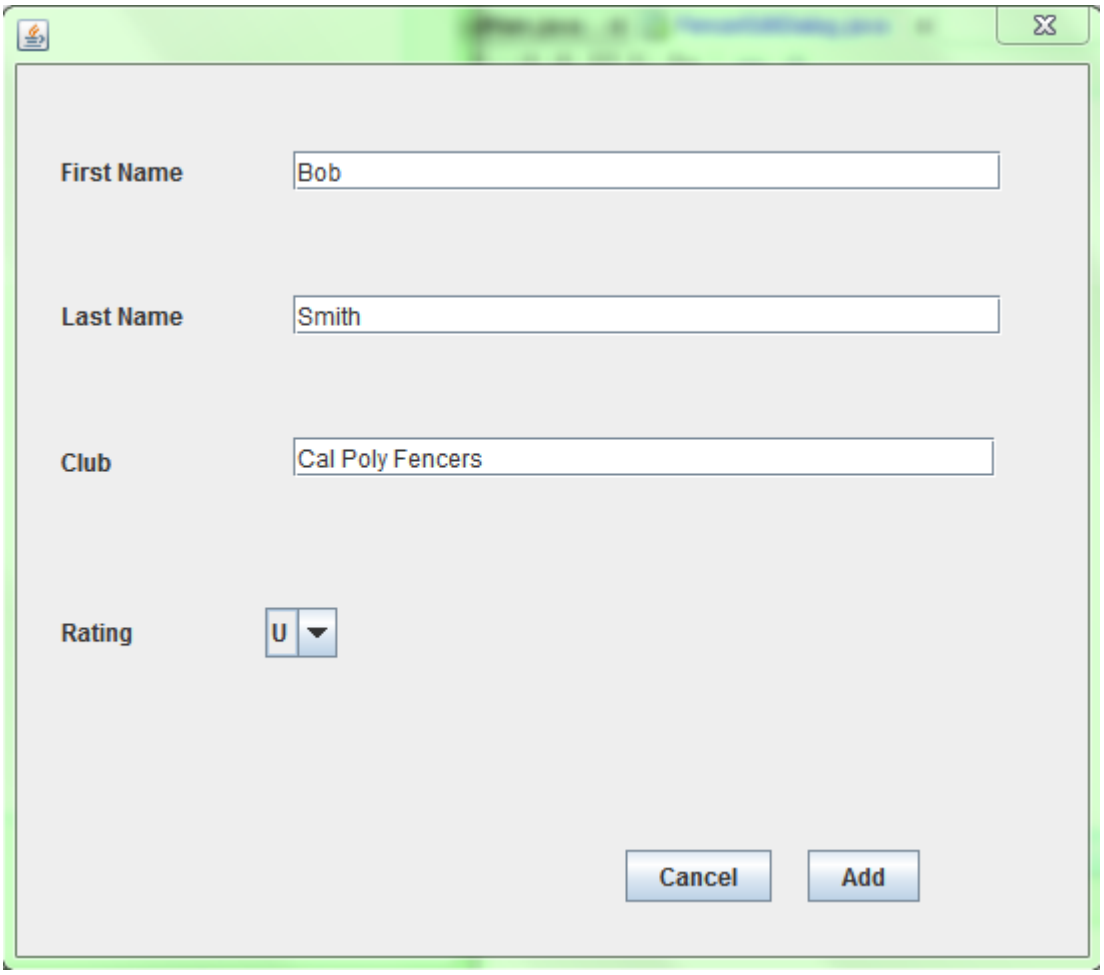


Fig. 3: The dialog for manually entering a fencer

Choosing Pool Size

The last set up step is choosing the number of pools. From the main event window, you will click "Begin Pool Selection". A dialog box will prompt you to choose how many pools to divide the fencers into in an even way based on their initial seeding by rating. This is important, as pools that are too large will take forever to run, and you don't want to be sitting at the desk all day. On the other hand, if the pools are too small, it doesn't accurately represent the talent, and some participants will be disappointed with how little they get to fence. In general, you should aim to have pool sizes between 5 fencers and 8 fencers (this decision will generally get easier with more experience running tournaments). Tournament Wizard will display at least one default option, with pools evenly divided and of the recommended size for you to choose from.

After inputting the desired number of the pools, text within the dialog will change to reflect the average number of fencers in a pool when divided among the number of pools chosen. Confirm the size of the pools is what you entered, and also confirm that the listed number of pools is acceptable.

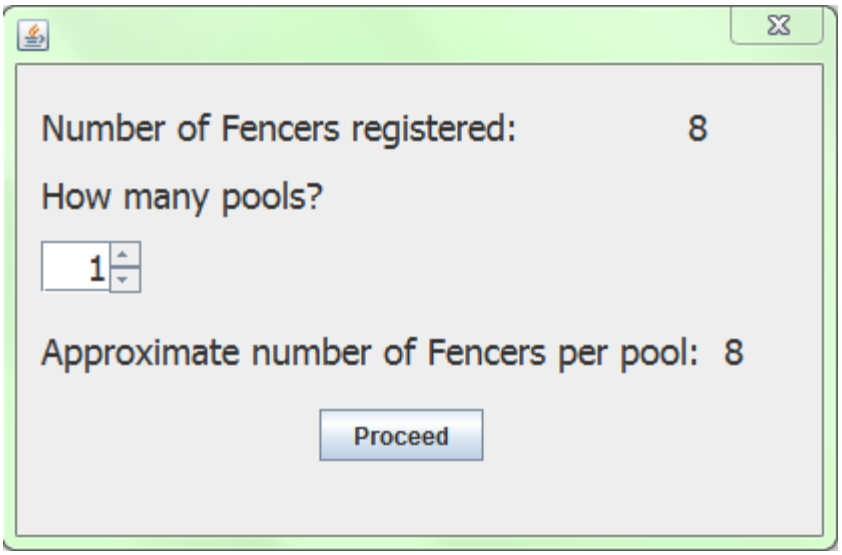


Fig. 4: Default value of the pool dialog window, before changing the desired number of pools

For example, if there are 8 fencers, and you select 2 as a pool size, the dialog text will say "Approximate number of fencers per pool: 8". See, isn't math is fun?

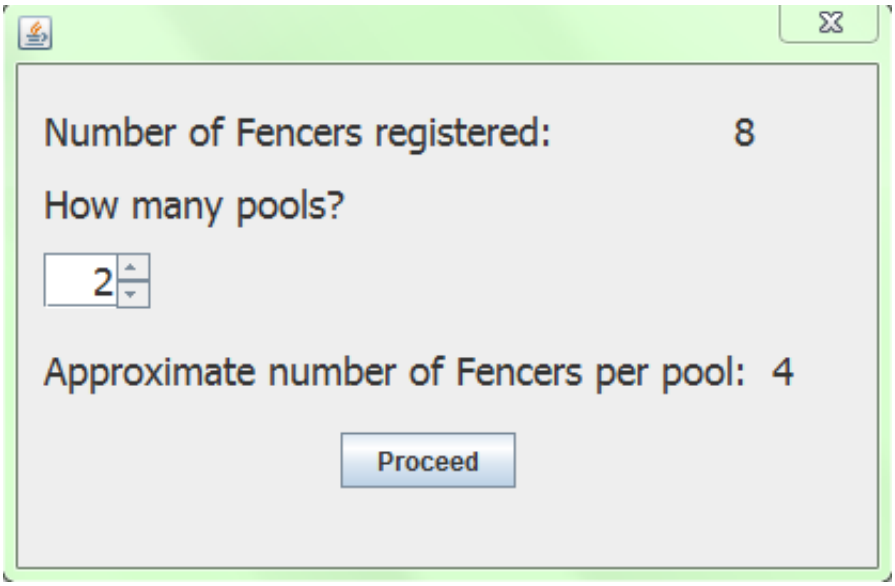


Fig. 5: Sample of the pool dialog window, after changing the desired number of pools

At this point, you can click "Proceed" to continue, or the "X" button in the top right corner to return to the event screen. By clicking proceed, Tournament Wizard will proceed into the pool phase of the tournament by assigning each Fencer into one of the available pools.

Chapter 2: Running Pools

At this point you have successfully started pools. Congratulations, you are now at the point where the tournament is officially under way. The point of the pools is to generate seeding for the fencers, which gets used in the next round to generate the tournament brackets. It does this in a fun way, letting fencers get short bouts against each other so everyone can get some excitement in before they possibly get knocked out in brackets.

The event screen will have changed, and will allow you to switch the view between the available pools. Each pool will show the fencers in it, arranged into a 2-dimensional grid. (See fig. 6). The referees will be using printouts of a grid like these to keep the scores of a bout.

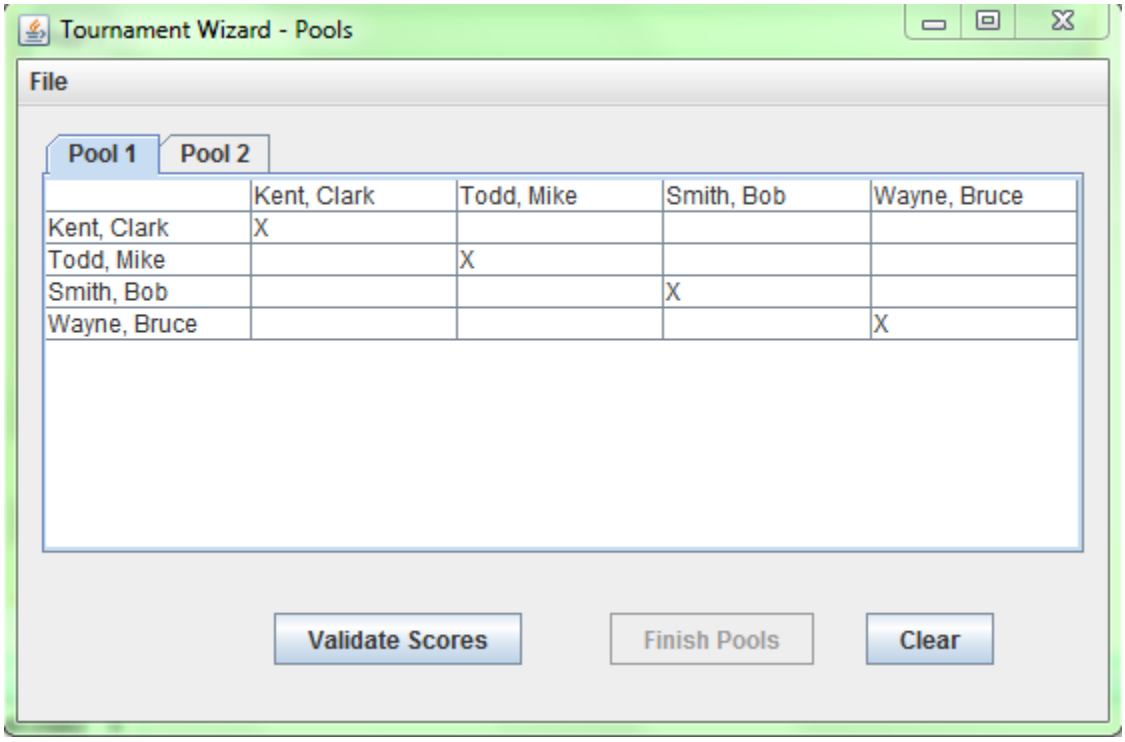


Fig. 6: A pool grid with 4 fencers. Note the tab to switch between pools to view)

If you are unfamiliar with how fencing pools work, it's not too important to know for the person simply running the information through the software. The basic premise though, is that each fencer will fence everyone else in their pool, and the results will be entered into this grid, where an individual cell will represent how the fencer on the left of the row fared against the fencer listed at the top of the column. Of course, a fencer can't fence themselves, no matter how hard they try, so no data will need to be entered on the diagonal.

The referees directing each bout will be in charge of overseeing the bouts and tallying the scores on score sheets. All you need to do as the software operator is simply sit

back and wait for numbers to be handed to you. The score sheets will even mirror the grids on the screen, so you just have to copy the numbers in the same order they appear for the correct pool. Starting from the open cell on the top most left, enter the scores in order from left to right down the row. Once you finish the row, you move down to the next row, and start from the first open cell in that row. You will not have to enter anything in the cells marked off with an 'X', as those would be instances of a fencer fencing themselves.

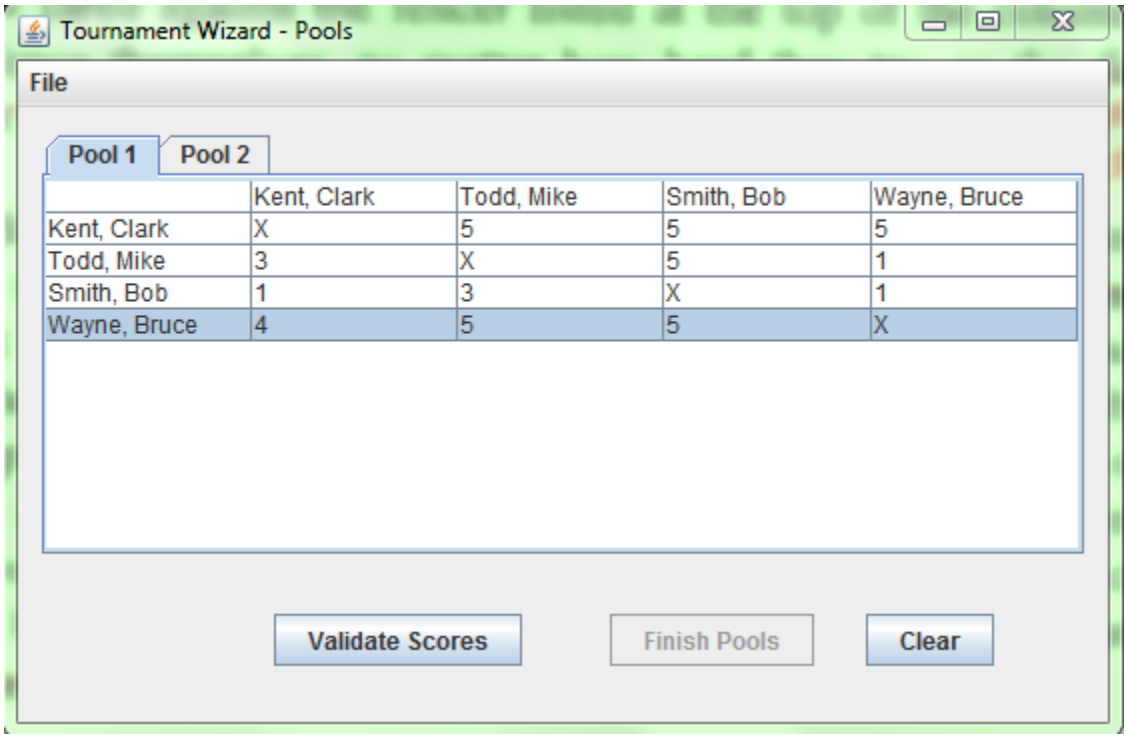


Fig. 7: An example of a grid after being filled in

Keep in mind, any blank cell will be considered a "0". If a fencer doesn't score a touch in a bout, you can either enter a "0", or leave it blank, and the software will fill it in. If you feel like you have made multiple mistakes, pressing the clear button will reset the grid to be a "0" in each cell. This will only clear the grid currently in view, so there is no need to worry about it affecting any other grids in the background.

Fig. 8: The error message that pops up when an invalid score is entered

Once the all the scores have been entered, click the "Validate Scores" button to ensure that the scores entered are valid. At this stage, the only way to have an invalid score is if both fencer have a score of "5", as that is the target score of a pool bout, and both fencers cannot win the same bout. If this happens, a dialog will pop up and tell you the names of the fencers that have invalid scores, allowing you to find them easier. If all scores in all pools are valid, the "Finish pools" button will become enabled, allowing you to proceed to the next screen to view the win percentages and indicators of each fencer (you don't need to know exactly what these terms mean, just know that they are numbers used to sort the fencers based on their performance in pools).

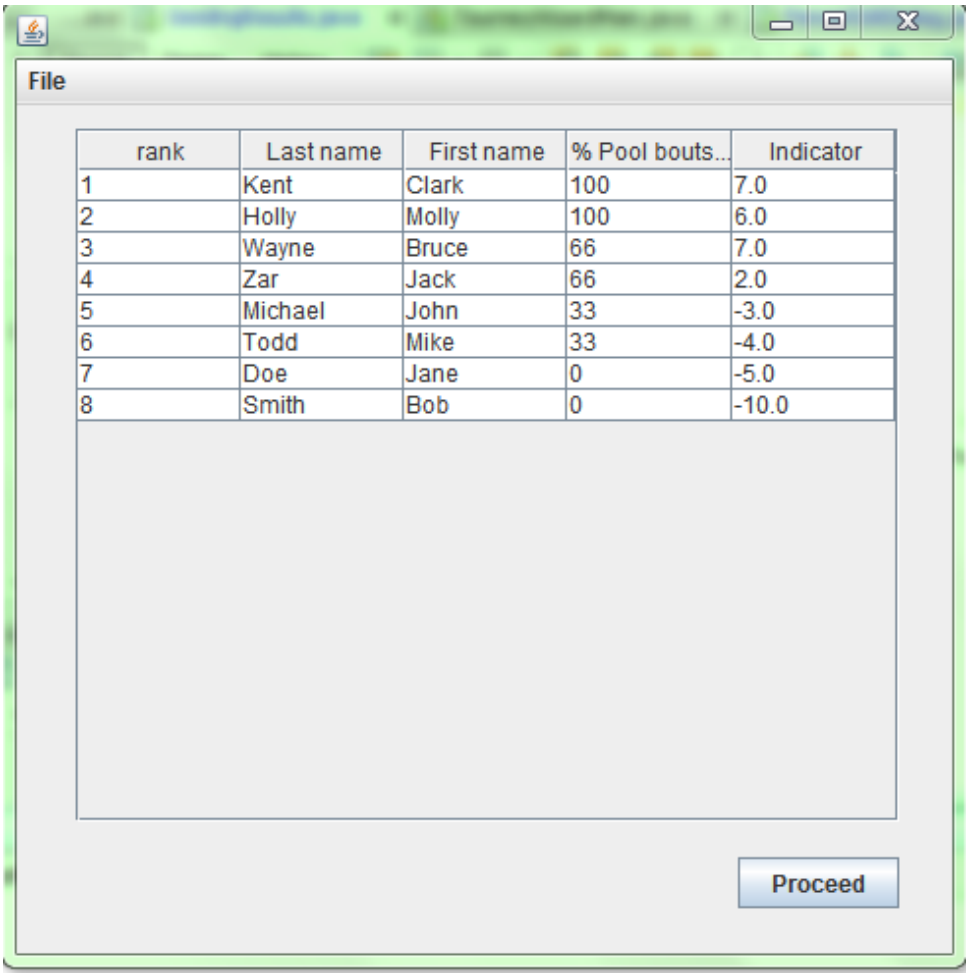


Fig. 9: The results after pools. The "Indicator" value is a term in fencing that means: touches scored in pools minus touches received in pools

Chapter 3: Elimination Round

This is the main attraction of the tournament. The final elimination round, where a loss means the fencer is no longer competing for the championship. Using the seeding generated from the pool round, the software will arrange the fencers into a bracket based on what their seeding is, with the number one seed facing the lowest seed, the number 2 seed facing the second lowest seed, and so on.

These bouts will also be officiated by the referees using different score sheets. Once a bout finishes, the score will be written on the sheet and signed by both fencers (this is a formality for them both to ensure the written score is correct), and will be brought back to you running the software to enter the results.

The screen will now display all the bouts in the current round of elimination. To enter the result of a bout, you will:

- Locate the corresponding bout listing the two fencers

- Click to open the corresponding combo box, which contain an item for each of the two fencers
- Click the fencer listed as the winner on the sheet that was provided to you
- You can also select the default "Undecided" option to not select a fencer if you accidentally clicked the wrong combo box

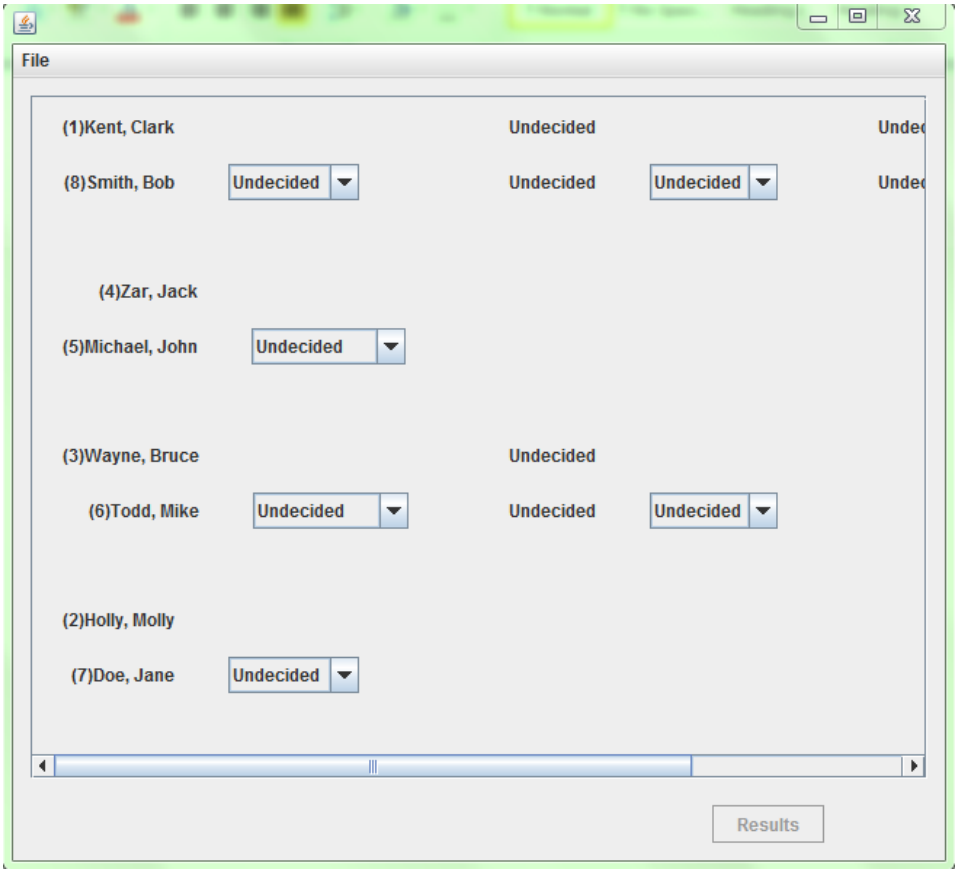


Fig. 10: Initial bracket. Note all the undecided bouts and the Results button being unavailable

The software will then advance the fencer you selected on to the next round of elimination. The fencer's name will appear in corresponding bout in the next round. However, if only one fencer in a particular bout has been decided, you will not be able to choose a winner in the combo box. Both fencers in a bout must have been decided for one to move on. Eventually, only one bout will remain undecided. This is the finals, where two fencers compete to become the champion. Once that decision is entered, the tournament will be over. As an important note, the "Results" button will only become enabled once the final bout has been decided. By clicking "Results", you can view the tournament results on a separate screen.

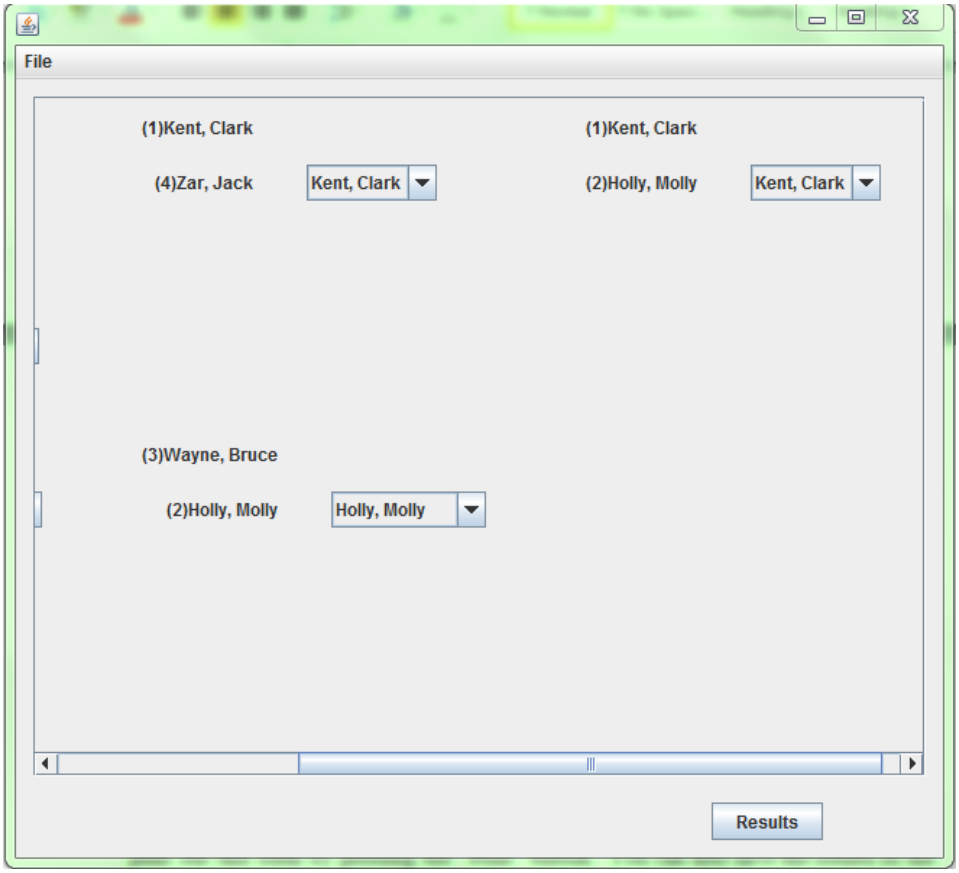


Fig. 11: The bracket after the finals have been decided. Note the combo box selections and the enabled button

Chapter 4: Results

This is what you've been working towards. You've made it to the end of the tournament, congratulations! From here, the event window will simply display the final rankings of the participating fencers in order. If you like, you will be able to save the results shown by pressing the "Save to file" button. This will save a list of the final results as a simple text document that can be viewed at any point after the tournament. As an extra feature to help differentiate multiple different result files, clicking the "File" menu will bring up a few menu items that can help you alter the results file. By selecting "Change result file name", you can choose the name that the result file gets saved as (if unspecified, default is "results.txt").

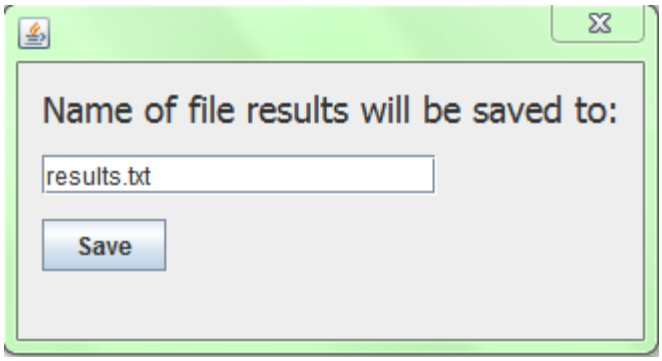


Fig. 12: The dialog that pops up to let you click "Change result file name" from the file menu

By selecting "Add result header", you are given a dialog that lets you enter a brief summary of the tournament. This can be whatever you choose to write, and will appear at the top of the file when you open it for viewing (if unspecified, there will be no header, and the result file will only show the results).

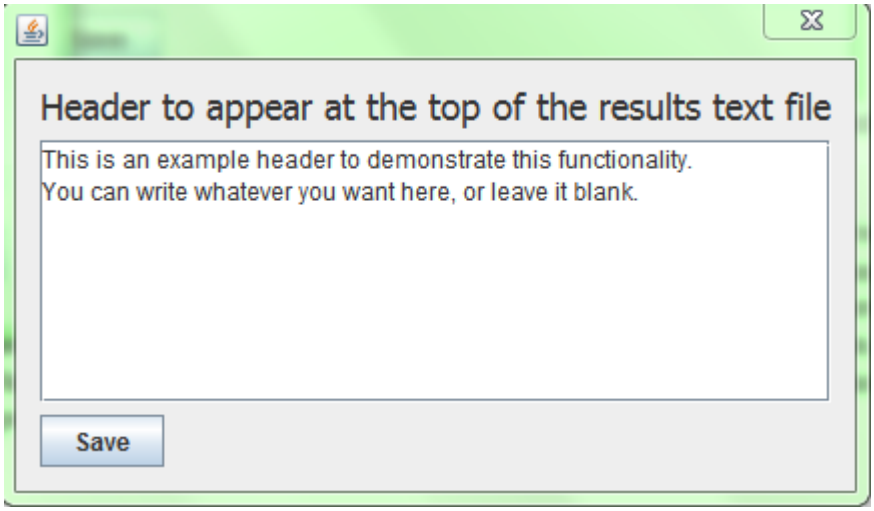


Fig 13: The dialog that pops up when you click "Add file header" from the file menu

When looking at the final results screen, you might notice that there will be two people listed as number 3, both having a "T" next to their rank. This indicates a tie, and is how fencing tournaments traditionally have 3rd place work. For all previous rounds, the fencers that get knocked out are ranked by what their seeding was, but for the two who get knocked out during the semi-finals, both get awarded 3rd place and share the podium.

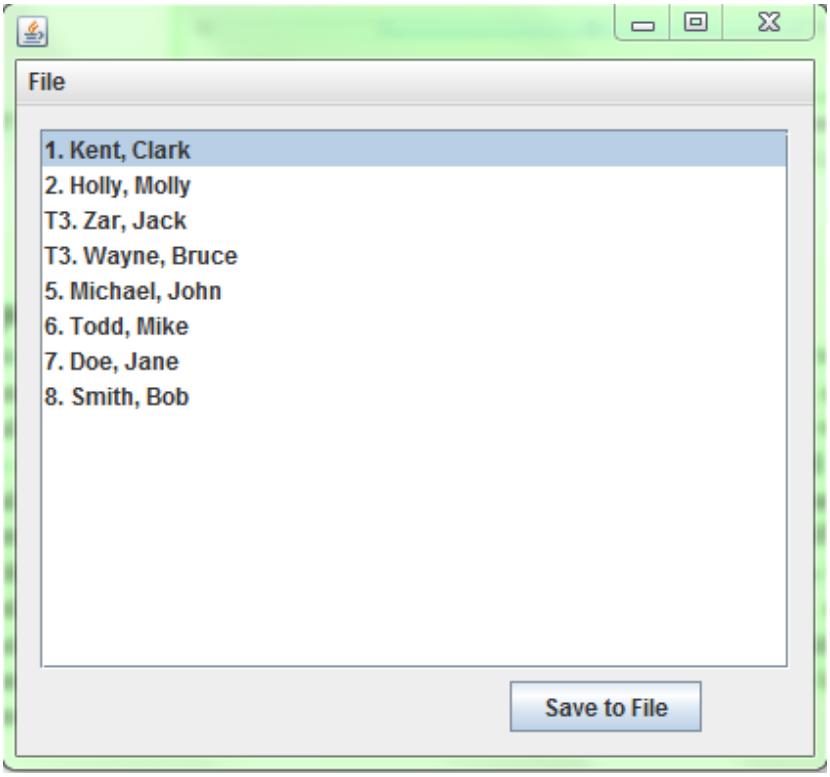


Fig. 14: The typical results screen. Note the two fencers ranked as number 3

```
This is an example header to demonstrate this functionality.
You can write whatever you want here, or leave it blank.


1. Kent, Clark
2. Holly, Molly
3. Zar, Jack
3. Wayne, Bruce
5. Michael, John
6. Todd, Mike
7. Doe, Jane
8. Smith, Bob
```

Fig. 15: An example of the output file with the example header entered in Fig. 10.

Glossary

I realize that for those of you unfamiliar with fencing, a few terms pop up that you might not know the exact meaning of. Don't you worry, we can fix that here.

Bout: A match where two fencers compete to see who can score more touches. Bouts are typically fenced to a score limit, or until time expires if that limit isn't reached

Direct Elimination: The main part of the tournament, bouts in this round generally go to 15 touches, and once a fencer loses once, they are knocked out of the event.

Indicator: Touches scored by a fencer in pools minus touches received by a fencer in pools. Used to seed fencers that have won an equal percentage of pool bouts.

Pool: A group of fencers in the first stage of the tournament. Generally, each fencer will have a bout to 5 touches against every other fencer in their pool

Seeding: The ranking assigned to a fencer based on their performance in pools. Higher ranked fencers will fence lower ranked fencers early on, to allow those high seeds the best chance to make it farther in the tournament.