Project Number. GFP0709

Enhanced DSL with Graphics

A Major Qualifying Project Report:

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

____

Christopher Woo

____

Victoria Zukas

____

Date: April 24, 2008

Approved:

____

Professor Gary F. Pollice, Co-Advisor

____

Professor Emmanuel Agu, Co-Advisor

1. Game Design
2. Rapid Prototyping
3. Domain Specific Language

Abstract

Modern game development can benefit from software that makes prototyping games easier. However, there is a notable lack of software for easily expressing concepts within the domain of computer board games. This project examines the domain component of rules in the context of strategic board games and what is required to facilitate rapid prototyping.

Acknowledgements

Thanks to all of the Students who participated in our user study.

Many thanks to Professors Gary Pollice and Emmanuel Agu for their time, advice, and guidance while advising this project.

Table of Contents

Abstract i

Acknowledgements ii

List of Illustrations iii

1. Introduction 1

2. Background 3

3. Methodology 7

4. Results and Analysis 28

5. Future Work 31

6. Conclusions 32

Appendix A 33

Appendix B 35

Appendix C 38

References 41

List of Illustrations

Figure 1: Parse Tree

Figure 2: UML diagram of the application

Figure 3: A sketch of the first design

Figure 4: The Create Objects Tab

Figure 4a: Create Object button window

Figure 4b: Create Properties button window

Figure 5: Rules tab

Figure 5a: See Code button window

Figure 6: Final Design. Create Objects tab

Figure 6a: Add Object window.

Figure 6b: Create a Value window.

Figure 7: Final Design. Create Rules tab.

Figure 7a: Add a Value window.

Figure 8: Final Design. Current Rules tab.

1. Introduction

Game developers without technical experience using computers as a platform for creating their games often times have difficulty prototyping their game quickly. This is because their lack of knowledge in the programming language that the game is being coded in. A tool that abstracted out the need for familiarity with a programming language would be very valuable to such developers. Using a domain specific language, it is possible to create such a tool.

A domain specific language (DSL) is a programming language that describes a specific problem domain. DSLs are usually limited in functionality but tailored to solve problems in the target domain. Java CC, for example is a DSL for generating parsers. Java CC is used to generate the Java classes needed to have a functional parser, allowing them to easily create applications with minimal amounts of code.

The goal of this project was to apply a user interface to the domain specific language component of rules of a turn based strategy board games. Input for the rules component is received via a custom Graphical User Interface designed so users can easily create and implement rules.

The background section of this paper includes an overview of domain specific languages, the use of game development tools and how they apply to domain specific languages, and visual programming as a user interface.

The methodology section of this paper includes an analysis of the scope of creating a domain specific language for strategic board games, rules as a component of the domain of strategic board games, representing rules as a context free grammar,

parsing a rule based on that grammar, and the design of a user interface as the method of providing input for that parser.

The results section contains the results of the survey. The survey consisted of participants using a working version of the user interface and filling out a questionnaire. The questions cover primarily ease of use and comprehension.

The future work section describes what improvements could be made to the project, including improvements to the user interface, expansion to other components of the domain of strategic board games, and the generation of a functional game.

2. Background

Domain Specific Languages

When solving a problem within a specific domain, aspects of the problem tend to be similar to other problems within that domain. Domain specific languages (DSL) are expressions of the knowledge held by a domain expert using the language and terminology of that domain with a computer. By using knowledge of the domain DSLs take advantage of the commonality within problems of the domain to improve the quality and efficiency of implementations of those problems by integrating solutions to the common issues a user encounters. DSLs also allow domain experts to create applications in their chosen domain more easily since a DSL is, by definition, designed to be used for the creation of applications within that domain. In contrast with general use programming languages a DSL can be used to make far fewer programs, but implements the programs that it can create very well. In essence a DSL must represent the structure and vocabulary of the domain it represents in order to actually do its job well. Overall Domain specific languages allow users to create applications in a domain by abstracting the commonalities of that domain and allowing the user to control the aspects of the domain that varies. (1).

Often Code Generators are used as part of an implementation of a DSL. Code generators take in input and use that input to create working software code. An example of a code generator is JavaCC. JavaCC is a code generator for creating parsers in Java. The program takes its input in the form of a grammar and generates Java classes that can parse input based on that grammar.

JavaCC is also a good example of a Domain specific language itself. It is useful in its domain of creating parsers. It always takes the same kind of input, a grammar and with it generates classes that are common with implementing a parser. The only concern for the programmer is writing the grammar correctly so that the parser can suit their needs.

Game Development Tools

Computer game development is a domain where tools to simplify it are very useful. There are many different roles with specialized knowledge required to create a game, such as artists, programmers, designers, and writers. Tools that can effectively reduce the time for these game developers are a valuable asset. This paper focuses on continuing to develop a tool for developers started by a previous project.

There are a wide variety of game development tools ranging from full programming languages that support specific common game concepts to applications for creating or modifying well known game types or genres. The primary difference between these tools is how programming oriented they are. (1).

An example of a more programming oriented tool is Torque 2D©. It provides many different function libraries for the creation of 2-dimensional sprite games. These libraries include abstractions ranging from sprite animation, physics, collision detection, networking, keyboard and mouse I/O, sound, and other common features of 2-dimensional sprite games. These libraries represent the Domain knowledge for 2-dimensional sprite games.

Another example, Game Maker© has a similar domain of games in that it focuses on 2-dimensional sprite games. However it provides a user interface that is more accessible to non-programmers. Users are able to drag-and-drop different components that represent actions or objects in a game program. This application is closer to the ideal product for this MQP. The difference however is the target domain. The domain being targeted by this project was strategy board games.

Strategy board games are a specific sub-genre of board games. Most board games are categorized by specific mechanics or themes. Common themes for example are economic, political, or military. Common mechanics are area-control, auction, race, or tile-laying mechanics. Many of these game types also have other common elements such as a turn structure, victory conditions, or movement and/or battle of pieces. (1).

Visual Programming

When designing our interface we wanted to allow users not familiar with programming languages to use our system. Therefore we decided to use visual programming. Visual programming is "writing programs in a language which manipulates information visually or supports visual interaction." (2). Programs using visual programming have some kind of drag and drop interface. Since most users would be familiar with the concept of dragging icons or text from one part of the screen to another we decided to use visual programming.

After doing research into different types of visual programming environments we noticed that many of them could be put into one of three types. The first type of environment had icons that the user dragged from one section of the screen into another

where the "coding" was done. These icons have an image that represents what part of the code they allow the user to create and/or edit. Also when one of these icons is selected options will appear on the screen, allowing the user to change certain properties. An example of this type of environment is LabView for LEGO Mindstorm's NXT development (3).

The second type of environment is similar to the first but requires the user have at least a passing knowledge of programming languages. In this type the user selects icons that represent a certain piece of code (such as Variable, If, and Calculate). These icons are then linked together using arrows to show the flow of the code. An example of this is the Microsoft Visual Programming Language (4) and Mindscript (5).

The third type of environment has multiple windows where the user selects a line of code and drags it to the main editing window. However, the lines of code are written in English. This type of environment was found in software that was designed to teach the user to program. An example is the program Alice (6).

After looking at these different types of visual programming environments we originally decided to create an interface similar to the first type of environment. However, after we decided to focus on the rules for a strategy board game we used a text based interface similar to the third type of environment. The user must know some basic programming terminology, such as the different between an integer and a boolean. However all of the "code" is written in simple English. Doing this allowed us to abstract out computer science concepts so that those not familiar with programming could create a rule.

3. Methodology

Analysis and Scope

The basic definition for the Domain specific language that this project intended to create was based on a previous MQP project, “Domain Specific Techniques for Creating Games”. The domain that was targeted was in fact strategy board games, the same domain as that project. The primary goal was to have a dedicated user interface that would target users with less programming experience while still having a robust language that a user could create their game in. This meant that our scope was smaller as we decided to focus on a specific subsection of that domain, namely rules.

To that end, this project focused on what defines a rule. We started by generating a list of common rules examples to define the key abstractions that represented what a rule was. The following is an example:

`If the terrain type is mountain, a piece cannot enter terrain`

This is a rule that is found in many area control board games, where pieces are restricted from entering a place on the game board based on a terrain type. The above format is a more formalized statement of that rule. A full list of these rules can be found in Appendix A.

From there we broke down the rules list into smaller components based on what was similar. These lists can also be found in Appendix A. We were able to find three specific commonalities that were consistent to all of the rules we generated. First, that there is a condition statement within the rule to determine if that rule should be executed

when it is check. Second, that there is an action statement within that rule that should be executed if the condition statement is true. Finally, that both the condition and action statements where based around either checking or changing specific properties of game objects. These three commonalities represented the basis of the rules grammar.

Rules Grammar

The rules grammar represents what composes a valid rule in a strategy board game. A user can represent a wide variety of rules specific to the domain of strategic board games based on this grammar. This grammar is in the format of a context free grammar since parsers use context free grammars as input by general convention. For a more complete definition of a context free grammar, see this website, (7). The following is the grammar itself:

- Rule → Condition Action
- Condition → Property ComparatorOperator Property | Property ComparatorOperator Value | Condition LogicOperator Condition
- Action → Property ActionOperator Value | Action AND Action
- *ActionOperator → = | + | - | * | /
- *ComparatorOperator → == | < | >
- *LogicOperator → AND | OR
- Property → Identifier: Value | Identifier.Property
- *Value → int | boolean | string | NULL

(Note: * denotes terminal symbol, in general follow order of operations and conventions of AND stronger than OR)

The grammar has all of the basic components of a context free grammar. First, it has non-terminal symbols such as Condition or Action. Second, it has terminal symbols such as ComparatorOperator or Value. Third, it has a set of relations based on those symbols. Finally, it has a starting symbol, in this case, Rule. The grammar, while being specific to strategic board games can, based on our analysis; represent a wide variety of rules within that part of the strategic board game domain.

A rule is composed of a condition statement and an action statement. The condition Statement is in general comparing a game object property with a ComparatorOperator, such as equality, less than, or greater than, with either another game object property or a simple Value. A Condition can also create more instances of basic Condition clauses using the final production and a LogicalOperator. In this way the grammar implements boolean logic to have complex condition statements. Between the two basic types of LogicalOperator, AND is stronger than OR. An action statement is very similar to a condition statement with two key differences. First, that the operators for a basic clause are different, representing changing game object properties as opposed to checking a condition. Second, that when there is more than one Action they are all connected by the LogicalOperator AND, since it's assumed that all actions will be executed.

A Property is a representation of a field of a game object. This field can be nearly any field that exists within the context of the game being created. The format of the Property production is such that any given property will be composed of a series of Identifier symbols representing the hierarchal class structure of game objects given by

the user, followed by a Value, a terminal symbol representing primitive types. The assumption of a hierarchical structure for game objects allows for both the use of the object oriented principle of inheritance, but also an easier way to organize various game objects that the user creates into categories.

Finally, the grammar supports three different operator terminal symbols. These represent the three types of operators needed at different times of composing a rule. First, is the ComparatorOperator that are used to make valid clauses of conditions. Examples include basic equality, greater than, and less than. Second, s the ActionOperator which are used to change properties if the condition associated with the rule is true. Examples include setting a value, addition, and subtraction. Finally, there is LogicOperator to connect smaller clauses within a Condition or Action together. These include the logical operators of AND and OR.

Parser

Any given rule would need to be checked to see if it was valid. To accomplish this task we created a parser based on the rules grammar in Java CC, a parser generator for Java that takes a formatted context free grammar as input. Java CC takes that input and generates the classes needed to have a parser with a parse tree based on the grammar. The parser takes the input of a rule as a string.

With this parse tree the various components of a rule can be checked in whatever order is needed to interpret the rule. The parse tree is traversed using the visitor design pattern. The visitor pattern traverses a parse tree by visiting the node in the tree that it is currently on and then calling the appropriate methods on each of its children. The order

in which children are called can be used to control what order nodes are traversed. This allows, for example, in the case of a parse tree for a rule, the condition statement to be checked first, followed by the action statement if needed. Figure 1 shows an example of the top levels of a rule parse tree.

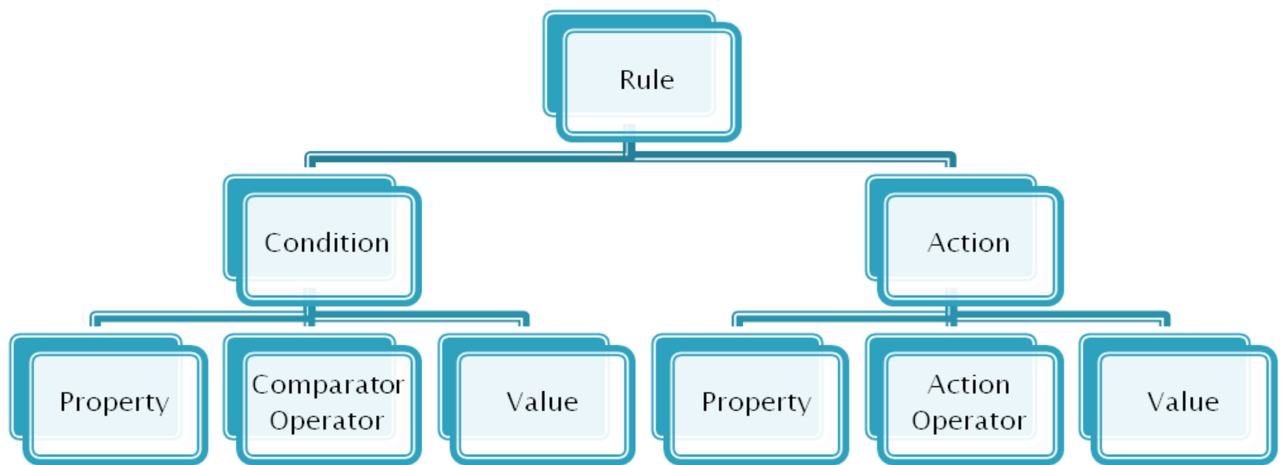

Figure 1: Parse Tree

This design is also very useful since the Template design pattern can be applied. The template pattern abstracts out what operations that do not vary from a coded procedure. In this case, the template pattern abstracts out the interpretation of parts of the rule parse tree. A rule will always have its condition statement checked first, and based on whether or not that condition is true, execute the action statement. In this case, what varies is how any given clause is interpreted in reference to the game being created. This

however is the only part of interpreting a rule that the user needs to do, and even then only if they are doing something different from the standard conventions of the rules grammar.

In order to properly execute this design, a number of custom classes were needed to be implemented. First, was the Value class to represent a Value object. A Value object is an object representation of a number of different primitives, in this case, string, int, and boolean. This object was passed up the visitor to the methods that checked the clauses. This allows the clauses to handle a variety of primitive types, as opposed to a single primitive type. Second, was the TemplateVisitor class. This class was the implementation of the traversal for a rule parse tree. Finally, there was an implementation of a basic interpretation of rules clauses, StrategoTemplate, based on the rules of the board game Stratego.

A UML Diagram of this package structure is shown in figure 2.

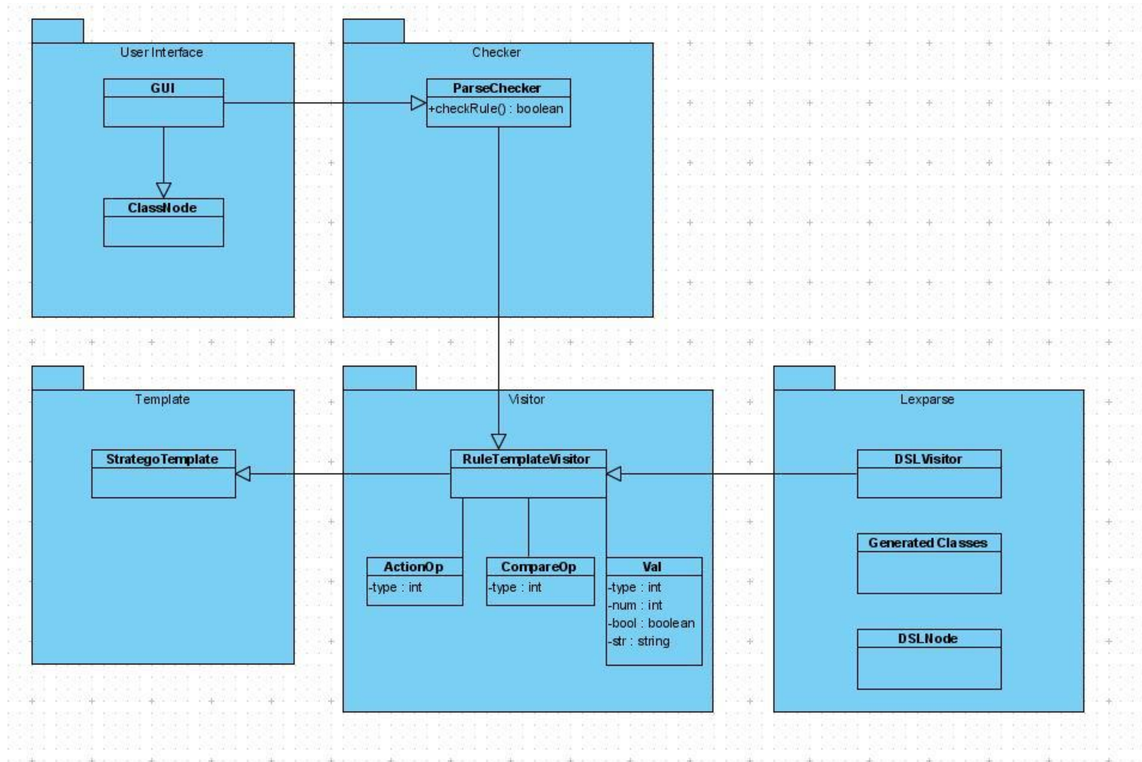

Figure 2: UML diagram of the application

User Interface Design

The user interface was one of the most important parts of the design of this project. Its purpose was to provide a way for the user to easily input a rule to be sent to the parser.

There were three different versions of the GUI design. The first design had three tabs, one for creating the board, one for creating pieces, and one for creating the rules of the game. The board and pieces tabs were nearly identical, except the board tab had extra options for defining the size of the board and the type of the board. Both the board and pieces tab were broken up into four sections. The left-most section had buttons to create

a new piece or tile (depending on the tab), a button to edit the tile or piece, a button to copy a piece or tile, and a button to upload a background image for the board. Above these buttons was space where an image of the created tile/piece would be displayed and on the top of that space, on the board tab, were options to create the board. If a tile or piece was clicked on the user would be able to edit the properties of that object. The middle section was space where the board would be generated. The user would drag the tiles and pieces from the first section into this section to place them on the board. The third section on the far right was for tools such as cropping and rotating a piece or tile. These would change an individual tile or piece. The last section was on the bottom of the GUI. This was where information would be displayed to the user, based on what they current had selected.

The Rules tab contained three lists and a separate space where the rules would be created. The first list contained the tiles and pieces the user had created, and their properties. The second had a list of operators, based on the list created by the previous project. These were pieces of code that would either combine rules (And, If, Or), or edit rules (Plus, Minus, Equals). The third list contained Constraints. Constraints were rules already created that could be grouped together to create longer more complicated rules. For example: If the user defined the rule " Tank.attack Equals 5 If Truck.defense Equals 6" then that would be saved as a Constraint and could be added on to later. The space on the far right was where the user would create the rule by dragging from each list. Figure 3 shows a drawing of the Board tab from the first design.

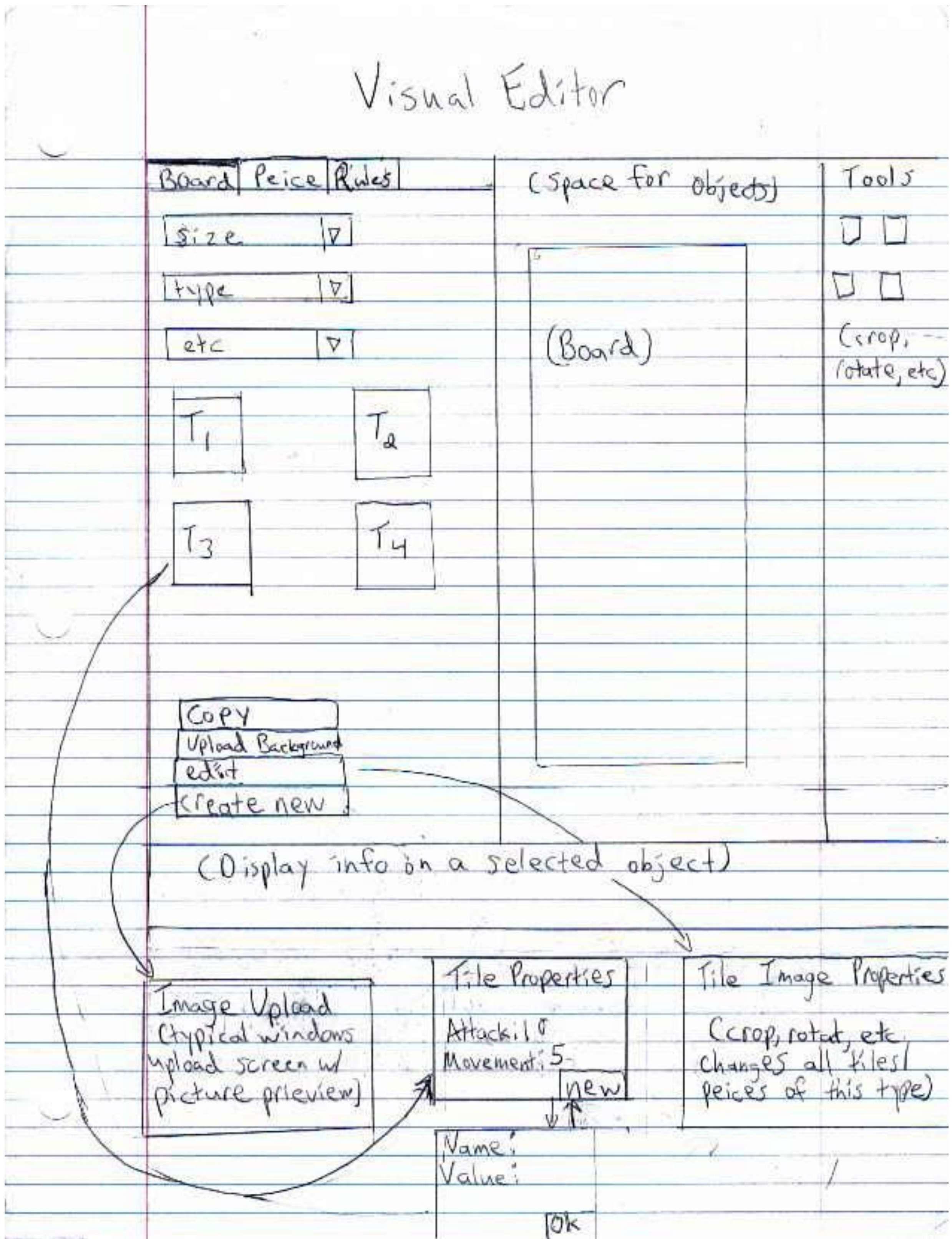

Figure 3: A sketch of the first design

After looking over this design we determined that the Rules tab could not do everything we needed it to do. While the objects and operators were fine, the Constraints were too limited. There weren't enough options to allow a user to program a game such as "Chess" or "Stratego", which we had been using as our baseline. Also, due to problems getting the previous project to work with our project, we decided to re-do the first two tabs and remove the graphical element, thereby focusing on creating a rules editor that could stand on its own.

The second design had two tabs, one for creating objects and one for creating rules. The Create Objects tab was a combination of the Board and Pieces tab from the previous design and was divided into two sections. On the top left were two buttons,' Create Object' , which allowed users to give an object a name and add it to the list of objects, and 'Create Properties', which allowed the user to add a property to an object. A property consisted of a name, a type (either integer, string, or boolean), and a value. Next to the two buttons was a tree object of the objects and their properties created and a space reserved for the board to be displayed and objects dragged onto it.

The Rules tab was greatly modified from the previous version. On the left side of screen were the object tree from the 'Create Objects' tab, and a list of Operators. Next to the Operators list were two text boxes. After looking at three well known strategy board games, "Stratego", "Chess", and "Checkers", it was determined that most rules for strategy board games can be made with If/Then statements. Therefore, the top box was for the "If" part of the rule and the bottom for the "Then" part of the rule. In order to prevent the user from creating an invalid rule, the tree of objects and the list of operators

was enabled and disabled based on what part of the rule was currently in the If/Then text boxes. Below the text boxes were two buttons. The “Add Rule” button saved the rule the user created. The “See Code” button opened a separate window that showed the syntax of what the user had created. This was important for advanced users. Since the text that the user interacts with is written in plain English those not familiar with programming languages can create rules. However, allowing the user to see the code behind what they are creating allows advanced users to save time and not have to go through unnecessary steps. Lastly, on the right side of the screen is a list of Actions. These are common events that happen in strategy board games and are already programmed for the user. If the user wanted to create their own unique events the “Create New Action” button is at the bottom of the list. Figures 4 through 5a show the second design.

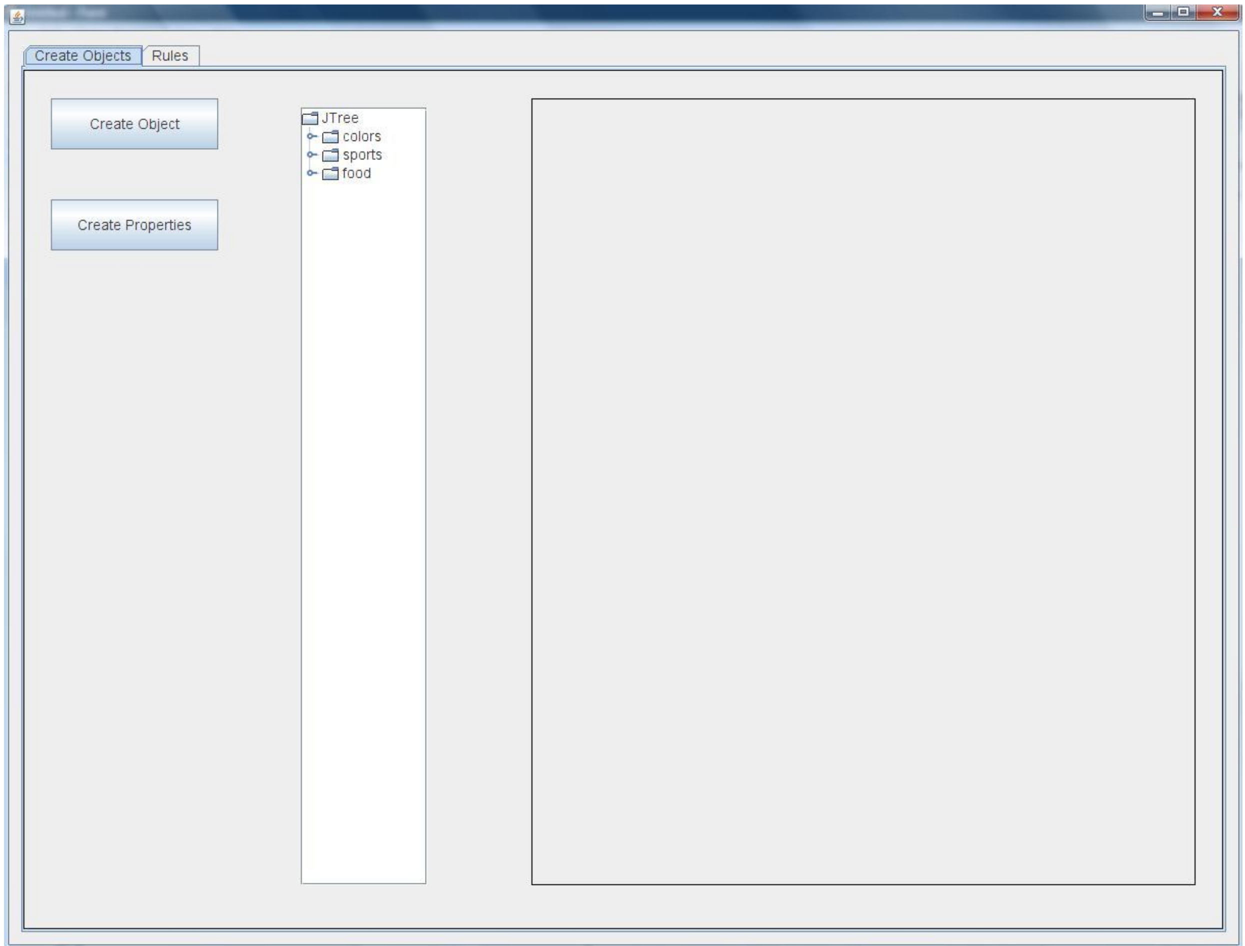

Figure 4: The Create Objects Tab

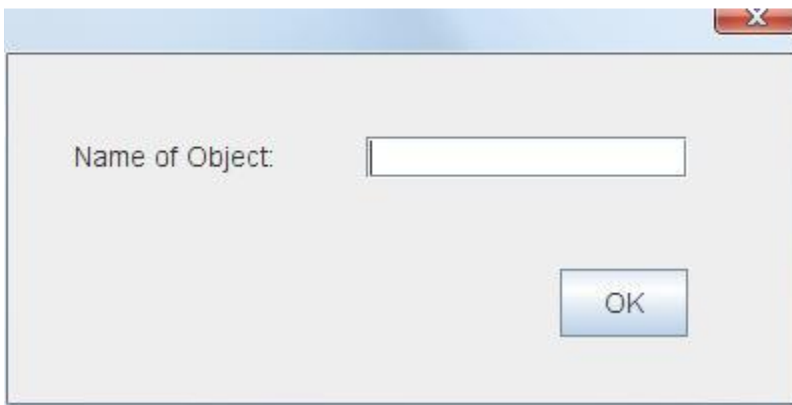

Figure 4a: Create Object button window

Figure 4b: Create Properties button window

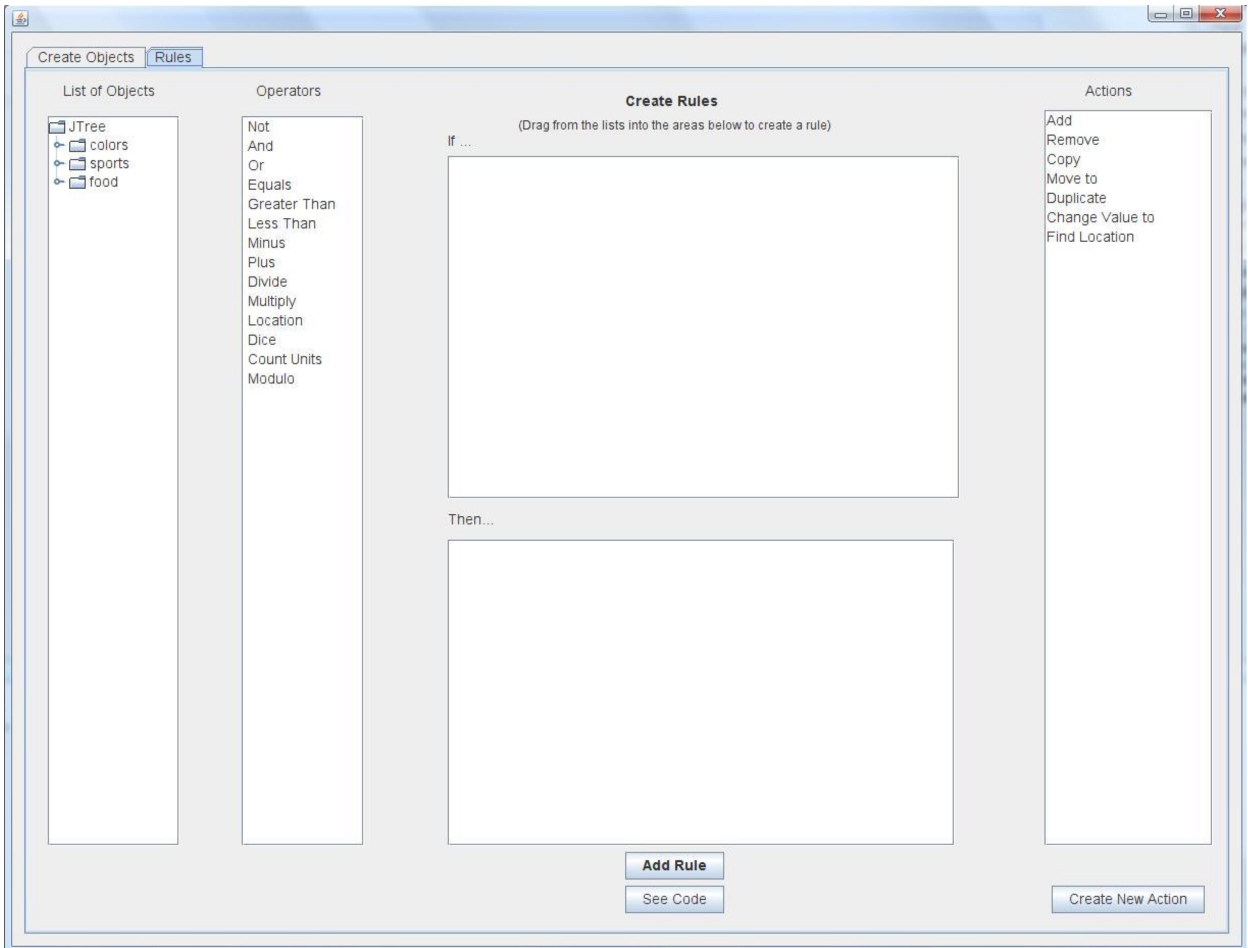

Figure 5: Rules tab

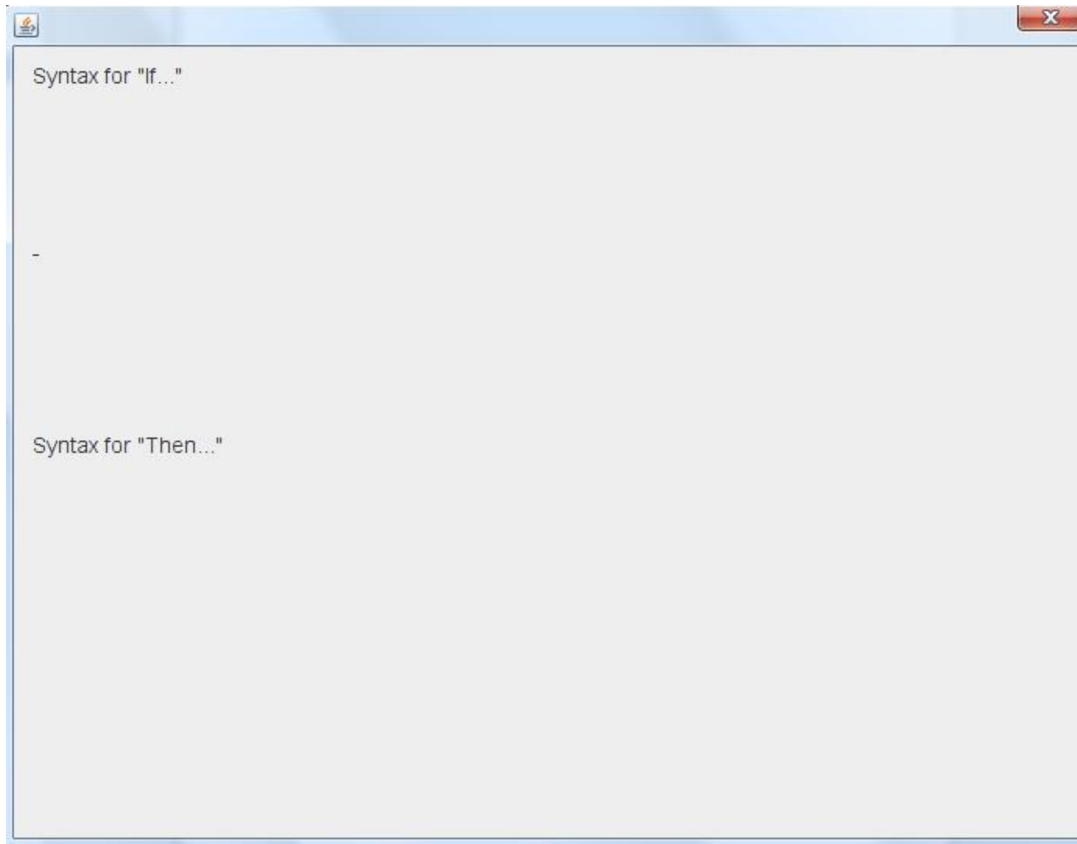

Figure 5a: See Code button window

This design focused on creating the rules of a strategy board game and removed the graphical element. While the objects tab was similar to the first design the rules tab was greatly improved. It contained all of the functionality we needed in an ascetically pleasing design that made process of creating a rule more user friendly. It also allowed for more kinds of rules to be created then the previous design.

In the final design we changed how objects and their properties were input into the UI and focused on guiding the user through our application. The Create Objects tab contains the same tree object that was in the previous design. However, due to changes in the grammar the first node in the tree is now ‘Object’. This way all pieces and tiles are considered objects. Next to it are three buttons. The ‘Add Object’ button creates an object

in the tree under the object that was currently selected. The user creates objects until they wish to assign a value. (Ex: To create a piece with an attack value the user would create a Unit object, a Piece object and an Attack object). The 'Create a Value' button allows the user to add a value to an object. A value can be either a String, an Integer, or a Boolean. (Ex: The user selects the Attack object and gives it a value of 5.) The 'Remove' button allows a user to remove a value and an object, as well as any values add objects underneath it. The 'Object' node cannot be removed. Lastly, the space reserved for the board was removed since by focusing on creating a rules engine we did not have the ability to create and display a board.

The 'Create Rules' tab is similar to the previous design but with key changes. The object tree will update as the user adds objects and values in the 'Create Objects' tab. The list of operators was broken up into three types of operators (logic, comparator, and action) to reflect the changes in the grammar. Also, by doing this we had more control over how the user would create a rule. Since certain operators are most often used in one part of the rule then another, operators are enabled and disabled based on which text box the user is currently working in. The user can switch between the If text box and the Then text box by clicking the 'Switch Active Section' button bellow the text boxes.

The list of Actions on the right side of the screen was removed from this design. After looking at the previous design we realized that the user could create all of those actions using the operators and created objects and having them listed separately was unnecessary. The last change was the addition of an 'Add a Value' button. This opens a window which allows the user to add a value that is not connected to an object. (Ex: To change a unit's status to "dead" the user would create in the Then section 'status Is' and

then use ‘Add a Value’ to put in the word ‘dead’). A value can be a String, an Integer, or a Boolean. In this design there is also a ‘Current Rules’ tab which allows the user to see what rules they have already created. Figures 6 through 8 show the final design of the interface.

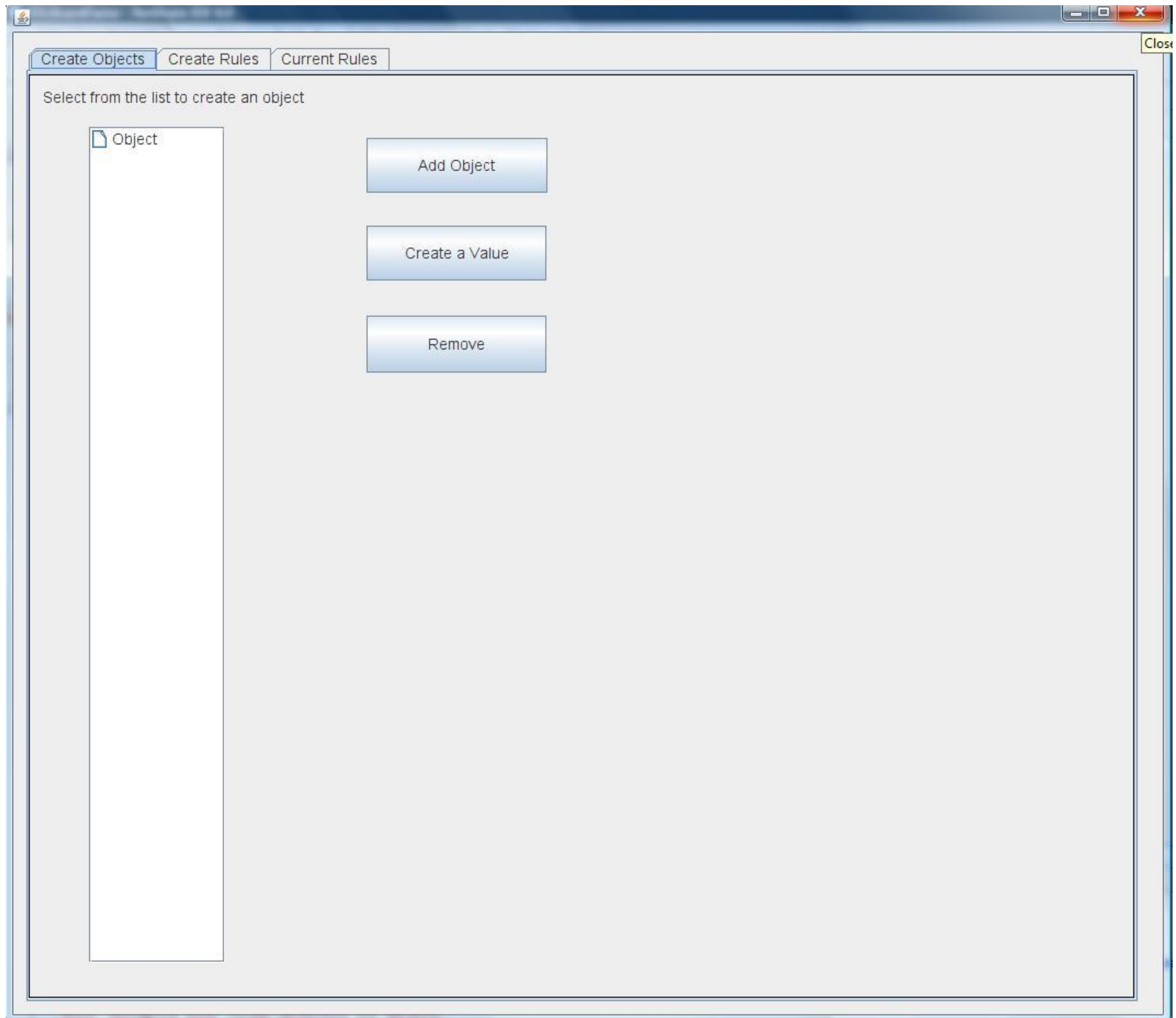

Figure 6: Final Design. Create Objects tab.

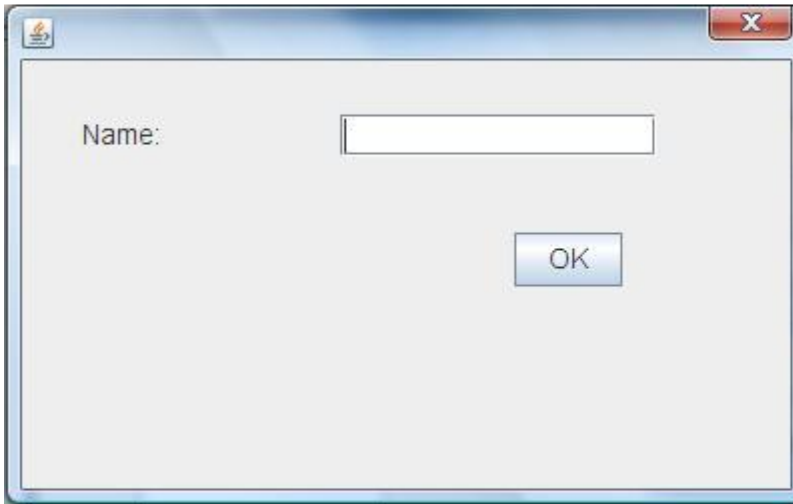

Figure 6a: Add Object window.

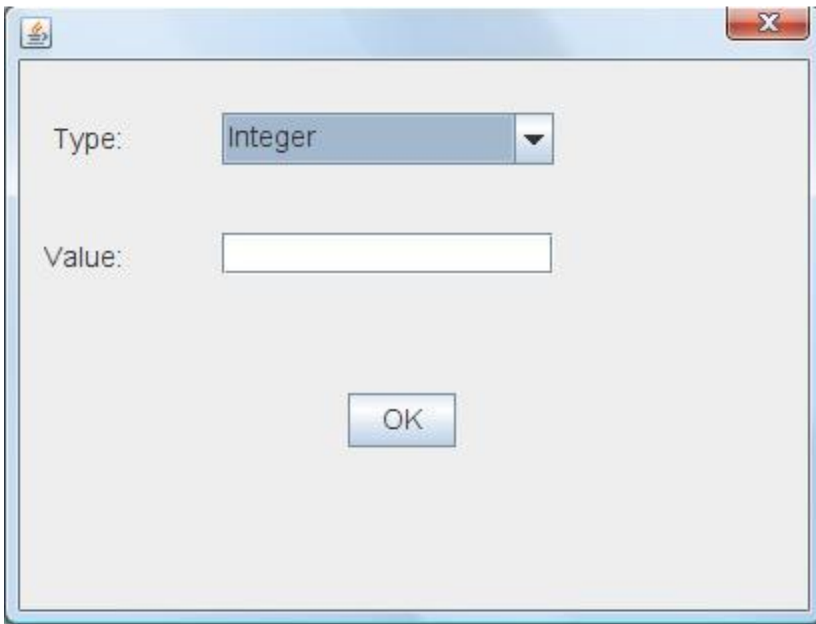

Figure 6b: Create a Value window.

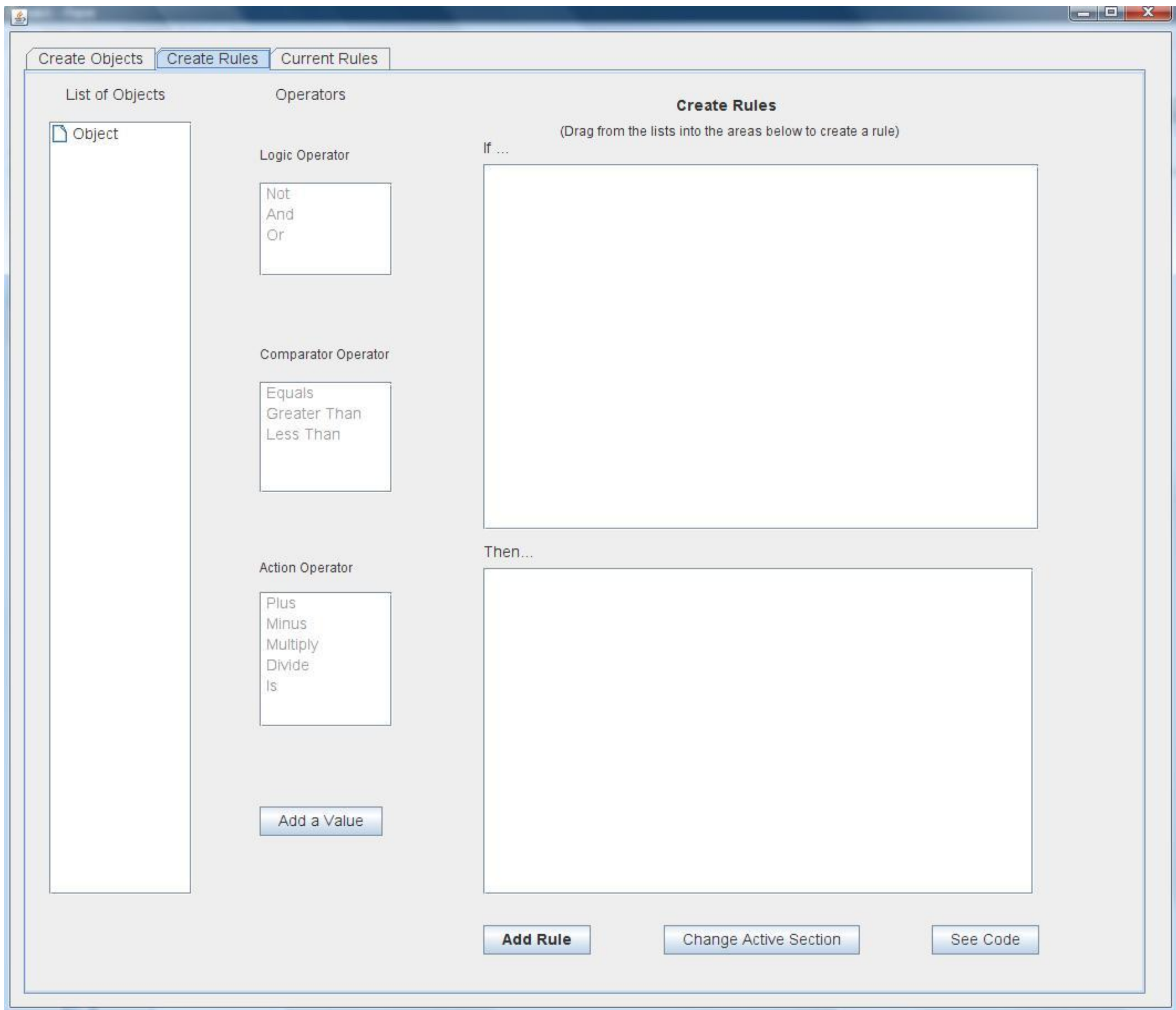

Figure 7: Final Design. Create Rules tab.

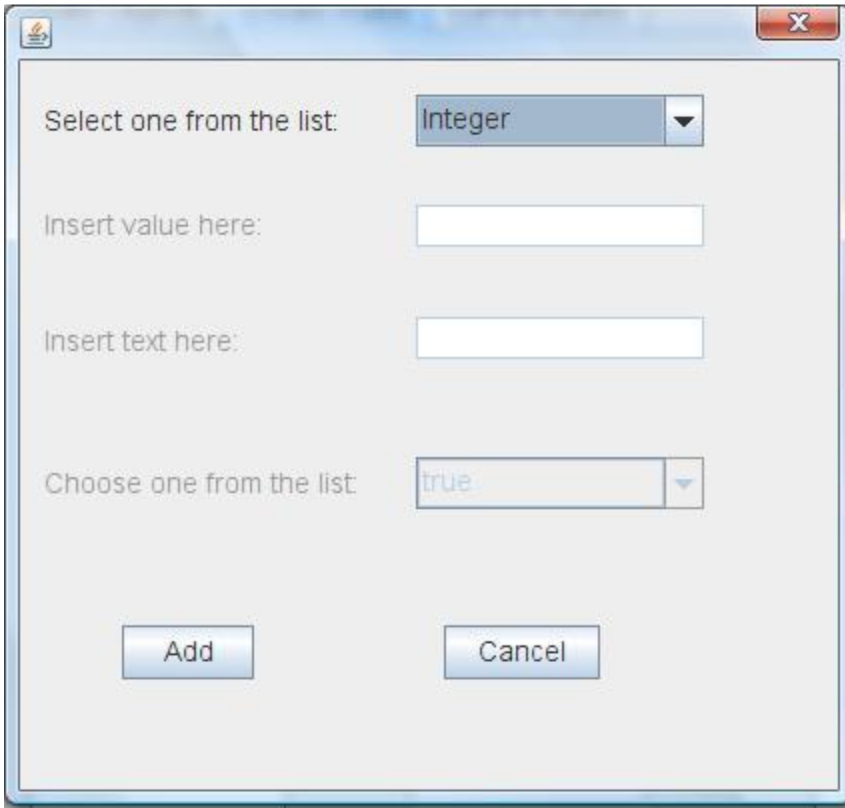

Figure 7a: Add a Value window.

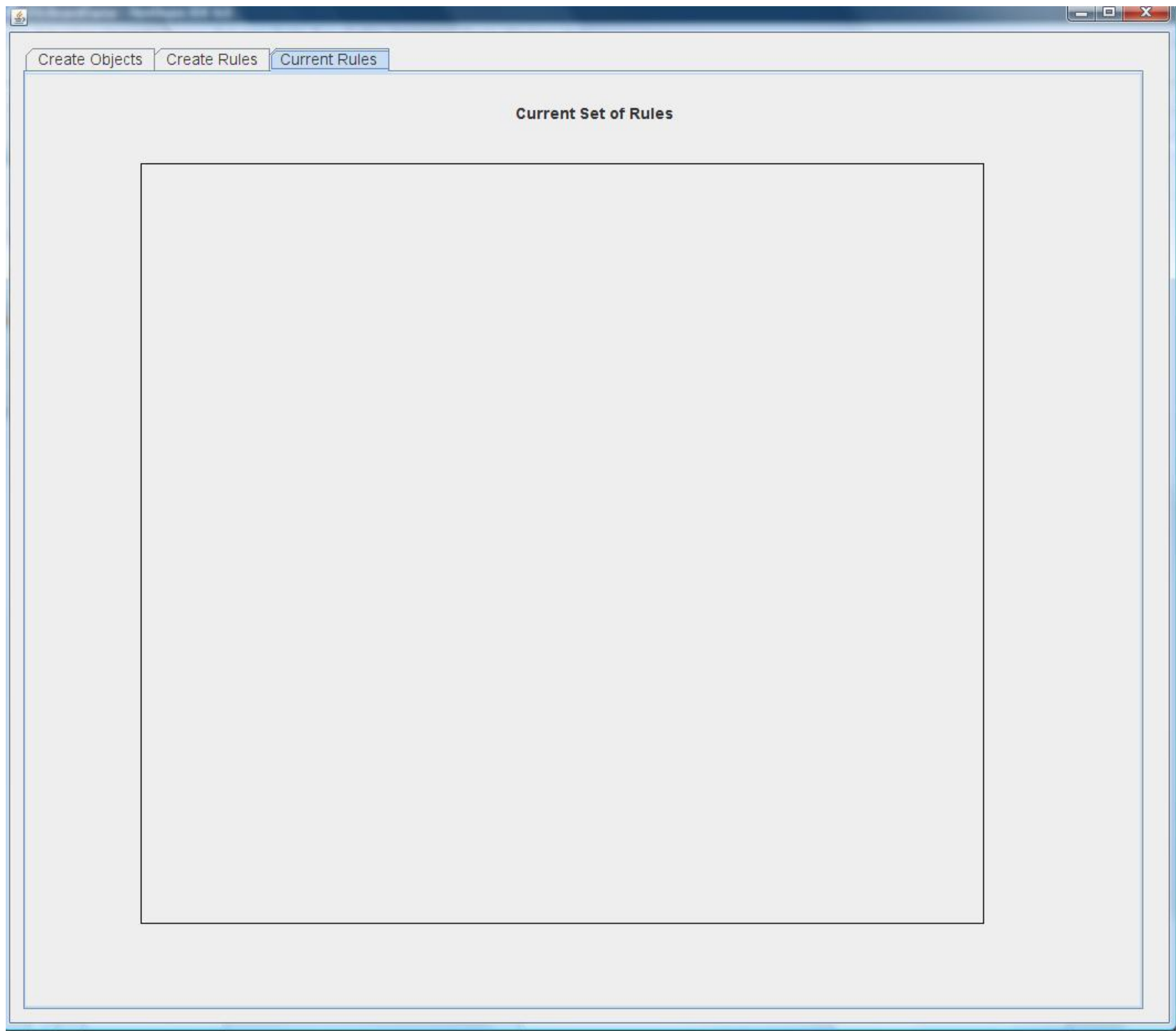

Figure 8: Final Design. Current Rules tab.

Throughout the different iterations we wanted to ensure our interface was following Human Computer Interaction principles. If the user gets distracted while using the interface it is important for them to be able to pick up where they left off. Depending on what part of the rule the user has created so far certain areas of the interface will be grayed out. The active sections are in black and the inactive sections are gray. This lets the user know what they should do next. Since the interface is texted based it is important

for the text to be legible. All of the text is in Arial 14 font, with the exception of instructions which are in Arial 12 font. Arial is a sans-serif font type. Sans-serif fonts do not have extra lines on letters (such as the two dashes that come off the top of a “T” in Times New Roman) and therefore are easier to read on computer monitors. Lastly, it is important that controls in an interface do what the user expects them to do. All of the buttons clearly state what they do and have an immediate response to the user clicking on them, either by changing the UI or by opening or closing a window. (8)

4. Results and Analysis

Metrics

There are 2 classes for the user interface, within GUI package, with 845 lines of written code and 1,028 lines of code generated by the Netbeans Form Editor. There is 1 class in the Checker package with 24 lines of code. There are 5 classes in the Visitor package with 247 lines of code. There is 1 class in the Template package with 78 lines of code. There is 1 custom class, 1 custom Java CC file, 23 generated classes, and 1 generated Java CC file in the Lexparse package. There are 17 lines of code written in Lexparse and 1,942 generated lines of code from JavaCC. Overall there are 1,218 lines of written code and 2,970 lines of generated code.

Survey

In order to test the usability of our interface we asked WPI students to test our application. The students were shown how to use the interface and then asked to create some of the pieces, tiles, and rules from the game 'Stratego'. The students were allowed to ask questions while creating the rules. After they were finished they filled out a survey asking different questions about how they interacted with the interface. The surveys filled out by the testers are shown in Appendix C.

Five students tested the interface; three were Computer Science majors, one was an Interactive Media and Game Development major, and one was a Biology major. We asked the Biology major to test our interface to see if the interface was simple enough that someone outside of the domain could use it. One of the Computer Science majors

and the Biology major were female. The other two Computer Science majors and the Interactive Media and Game Design major were male. The testers were able to create all of the objects and rules they were asked to create. The Computer Science majors and Interactive Media and Game Development major were able to finish the instructions in 20 to 26 minutes. The Biology major took 33 minutes, although she often paused to make comments and suggestions.

There were problems with the interface that the majority of the testers pointed out. While they were able to accomplish the set of instructions they were given, it was not always easy to do. Hitting the enter button on the keyboard they believed should cause the OK button in a pop-up window to be pressed; however it did not. While creating objects and properties the testers complained that they had to repeat the same steps several times and having the ability to copy and paste would make creating objects and properties much easier. Another complaint was that the pop-up windows did not appear near the main window and so seemed to pop up in a random place on the screen. Also, the testers were quick to point out that they didn't like having to press a button to switch between the If and Then boxes while creating rules. They believed that clicking on the appropriate text box should activate it. The last major complaint was that if the tester dragged an object or operator over into the If or Then text box the object or operator stayed highlighted. This was a problem because if they then went to type in part of the rule, such as 'dead' for changing a piece's status, the object or operator they just dragged over would be deleted.

One problem that all of the testers pointed out was that properties did not display the object they were connected to when they were dragged into the If and Then text

boxes. The testers were not sure if the application knew which property was tied to which object. Also, after the testers created a rule when they selected the ‘Current Rules’ tab to see what they had done so far it was very hard to tell which rule they had made. (Ex: The rule would say “Rank Greater Than Rank” instead of what they were expecting which was “Scout.Rank Greater Than Bomb.Rank”)

Despite the problems the testers had with the interface most of them said they would use our interface to design the game on some level. The Interactive Media and Game Development major and one of the Computer Science majors said they would use the interface to create the entire game, and that the interface would be faster to use then hand coding the game. One Computer Science major said he would use the interface to do design work and then hand code the game. The other Computer Science major said she would rather hand code the game. The Biology major had never coded before using our interface so she could not say. The set of instructions and the testing surveys are in Appendix B.

5. Future Work and Conclusions

User Interface Improvements

The usability of the user interface is as clear as it could be. Based on the survey on the user interface, there are a number of things that could make the interface more intuitive. These include improvements to functionality such as the ability to save and edit rules along with other improvements such use of the enter key to confirm action in open windows. While not a full project in of itself, this could be a useful part of an extending project.

Expansion of more components

Currently only the rules component of the strategy board game domain is implemented. There are more components to this domain, such as a board, full implementation of game objects, and a turn structure that need to be designed and created to have a complete domain specific language. The creation of the code generators for these components and extension of the user interface to encompass these new components would be a valuable addition.

Graphics and Prototyping

It is possible to include within the domain the generation of a working game itself. If given proper input of graphics and configuration information from the user, the commonality of the strategic board game domain would allow for generators to create working code that represents a running prototype of game input into the system.

6. Conclusions

There are many game development tools for the generation of computer games, but very few that focus on computerized versions of board games. This project was designed to create a tool for the use of game developers to rapidly prototype their game ideas with minimal effort and coding knowledge.

The rules grammar represents an easier way for developers to create rules for their games in a way a computer can interpret. The associated parser also provides functionality for executing those rules during runtime.

The user interface is designed to make programming a strategic board game more accessible to a developer by abstracting away many programming concepts that are unnecessary for designing the game itself.

In conclusion the rules grammar, user interface, and parser are a good start to a game development tool for strategic board games. However more work is required to make it a viable tool for developers. There are many more additions required. This project was a successful start on creating that tool.

Appendix A

A List of Common Rules in a Strategy Board Game

(Note: () denotes a specific instance of a game object)

- If the terrain type is () type, a piece cannot enter terrain
- If the terrain type is () type, change () piece's attack, defense, movement to ()
- If the terrain type is () type, remove piece from the board
- If () piece has not moved move the piece to () location on the board
- If () piece has not moved piece may capture (here).
- If () piece reaches the edge of the board replace () piece with () piece
- If () piece reaches the edge of the board remove the piece
- If () piece reaches the edge of the board duplicate () piece
- If () piece reaches the edge of the board change attack, defense, movement value to ().
- If () piece is () type then remove from the board
- If () piece is in () location, remove from the board
- If () piece is in () location, duplicate piece
- If () piece is in () location, move piece to () location.
- If () piece and () piece are in the same tile () piece is removed from the board.
- If () piece and () piece are in the same tile change () piece's attack, movement, defense to () value.
- If piece is of type () it cannot enter () tile.
- If piece is of type () it can enter tile.

- If piece has movement equal to, less than, or greater than () value () piece cannot move into terrain type.

Condition Statement Examples

- If tile type is A
- If piece has not moved
- If piece reaches edge of board
- If piece is of type A
- If piece is in location
- If piece A and piece B are both in location C
- If piece value equals A
- If piece value is less than A
- If piece value is greater than A

Action Statement Examples

- Piece cannot enter tile type
- Piece's attack value is A
- Pieces' defense value is A
- Piece's movement value is A
- Piece's location is (A,B)
- Remove Piece
- Duplicate Piece
- Add Piece

Appendix B

Testing Instructions

Pieces:

Create a Marshal –

1) Select "Object", Click on "Add Object", Type in "Unit", and click OK.
2) Select "Unit", Click on "Add Object", Type in "Marshal" and click OK.
3) Select "Marshal", Click on "Add Object", Type in "Status" and click OK
4) Select "Status", Click on "Create a Value", Select "String", Type in "Alive" click OK
5) Select "Marshal", Click on "Add Object", Type in "Rank" and click OK.
6) Select "Rank", Click on "Create a Value", Select "Integer", Type in "10" click OK
7) Select "Marshal", Click on "Add Object", Type in "Type" and click OK.
8) Select "Type", Click on "Create a Value", Select "String", Type in "marshal" , click OK.

Create a Sergeant -

1) Select "Unit", Click on "Add Object", Type in "Sergeant" and click OK.
2) Select "Sergeant", Click on "Add Object", Type in "Status" and click OK
3) Select "Status", Click on "Create a Value", Select "String", Type in "Alive" click OK
4) Select "Sergeant", Click on "Add Object", Type in "Rank" and click OK.
5) Select "Rank", Click on "Create a Value", Select "Integer", Type in "4" click OK
6) Select "Sergeant", Click on "Add Object", Type in "Type" and click OK.
7) Select "Type", Click on "Create a Value", Select "String", Type in "sergeant" , click OK

Create a Miner –

1) Select "Unit", Click on "Add Object", Type in "Miner" and click OK.
2) Select "Miner", Click on "Add Object", Type in "Status" and click OK
3) Select "Status", Click on "Create a Value", Select "String", Type in "Alive" , click OK
4) Select "Miner", Click on "Add Object", Type in "Rank" and click OK.
5) Select "Rank", Click on "Create a Value", Select "Integer", Type in "3" , click OK
6) Select "Miner", Click on "Add Object", Type in "Type" and click OK.
7) Select "Type", Click on "Create a Value", Select "String", Type in "miner" , click OK

Create a Scout –

1) Select "Unit", Click on "Add Object", Type in "Scout" and click OK.
2) Select "Scout", Click on "Add Object", Type in "Status" and click OK
3) Select "Status", Click on "Create a Value", Select "String", Type in "Alive" , click OK
4) Select "Scout", Click on "Add Object", Type in "Rank" and click OK.
5) Select "Rank", Click on "Create a Value", Select "Integer", Type in "2" , click OK
6) Select "Scout", Click on "Add Object", Type in "Type" and click OK.
7) Select "Type", Click on "Create a Value", Select "String", Type in "Scout" , click OK

Create a Spy –

1) Select "Unit", Click on "Add Object", Type in "Spy" and click OK.
2) Select "Spy", Click on "Add Object", Type in "Status" and click OK
3) Select "Status", Click on "Create a Value", Select "String", Type in "Alive" , click OK
4) Select "Spy", Click on "Add Object", Type in "Rank" and click OK.
5) Select "Rank", Click on "Create a Value", Select "Integer", Type in "1" , click OK
6) Select "Spy", Click on "Add Object", Type in "Type" and click OK.
7) Select "Type", Click on "Create a Value", Select "String", Type in "spy" , click OK

Create a Bomb –

1) Select "Unit", Click on "Add Object", Type in "Bomb" and click OK.
2) Select "Bomb", Click on "Add Object", Type in "Status" and click OK
3) Select "Status", Click on "Create a Value", Select "String", Type in "Alive" , click OK
4) Select "Bomb", Click on "Add Object", Type in "Rank" and click OK.
5) Select "Rank", Click on "Create a Value", Select "Integer", Type in "0" , click OK
6) Select "Bomb", Click on "Add Object", Type in "Type" and click OK.
7) Select "Type", Click on "Create a Value", Select "String", Type in "bomb" , click OK

Tiles:

Create a Ground Tile –

1) Select "Object", Click on "Add Object", Type in "Tile", and click OK
2) Select "Tile", Click on "Add Object", Type in "Ground", and click OK
3) Select "Ground", Click on "Add Object", Type in "Occupied", and click OK
4) Select "Occupied", Click on "Create a Value", Select "Boolean", and Type in "false" , click OK
5) Select "Ground", Click on "Add Object", Type in "Unit", and click OK
6) Select "Unit", Click on "Create a Value", Select "String", and Type in "None" , click OK

Rules:

1) A tile may only be occupied by one piece
   If: (Ground) occupied Equals false.
   Then: (Ground) occupied Is true And (Ground) unit Is Sergeant

2) If a piece attacks, the piece with the lower rank is removed.
   If: (Sergeant) rank Greater Than (Scout) rank
   Then: (Scout) status Is dead

3) If a piece attacks and both pieces are equal, both are removed
   If :(Sergeant) rank Equals (Sergeant) rank

Then: (Sergeant) status is dead

4) If a Miner attacks a Bomb, the Bomb is destroyed.
If: (Miner) type Equals miner And (Bomb) type Equals bomb
Then: (Bomb) status Is dead.

5) If a Spy attacks a Marshal, the Marshal is destroyed.
If: (Spy) type Equals spy And (Marshal) type Equals marshal
Then: (Marshal) status Is dead

Appendix C

Survey's Completed By Testers

Name: Sam
Major: Computer Science
Time: 20 minutes

Do you believe (based on the response from the UI) that you programmed the individual rules correctly?

- The rules seem correct although the UI is such that it might be hard to remember what objects the fields you are referencing are in after the fact if rules needed changing.

Did you find it easy to program the individual rules?

- Aside from some minor glitches it was easy enough. I would definitely add a way to see what objects your attributes come from and preferably a way to know when rules would be invoked and on what instances of objects.

Did you find it easy to create objects?

- Object creation is easy enough. Could use copy/paste and more inheritance stuff.

Did you find it easy to add properties to objects?

- Yes, see above.

Were there any properties you felt you couldn't add to the objects? Explain.

- No, provided everything boils down to strings, integers, and Booleans after a point.

Was the UI intuitive (Did the buttons, objects, etc. do what you thought they would do?)

- Hitting enter should probably substitute clicking the okay button.

If given the choice would you program the same game with this UI or would you code the game by hand? Explain why.

- I would probably use this interface to do design work for the game, but code it by hand.

Name: Mikey
Major: Computer Science
Time: 26 minutes

Do you believe (based on the response from the UI) that you programmed the individual rules correctly?

- More than likely.

Do you believe (based on the response from the UI) that you programmed the game correctly?

- Yes, nothing seemed to tell me I was doing it wrong

Did you find it easy to program the individual rules?

- No, the highlighting made typing a pain, and changing the field was a hassle

Did you find it easy to program the game?

- relatively

Did you find it easy to create objects?

- Yes

Did you find it easy to add properties to objects?

- Yes

Were there any properties you felt you couldn't add to the objects? Explain.

- A double or a float?

Was the UI intuitive (Did the buttons, objects, etc. do what you thought they would do?)

- The UI for the objects made sense, and while the buttons did what I expected them to do for rule generation, it was harder to do than expected

If given the choice would you program the same game with this UI or would you code the game by hand? Explain why.

- This UI since it seems like it would be quicker, assuming I did not make any mistakes programming it. Don't know how easy it would be to debug it all.

Name: Amanda
Major: Computer Science

Do you believe (based on the response from the UI) that you programmed the individual rules correctly?

- It added them. I couldn't tell if it was doing any error checking though.

Do you believe (based on the response from the UI) that you programmed the entire game correctly?

- The UI didn't give much response at all. I was able to add a rule that I'm almost certain was meaningless.

Did you find it easy to program the individual rules?

- Implementing double-click as well as drag and drop might simplify it.

Did you find it easy to program the entire game?

- It wasn't difficult, however in the rules, it was hard to tell what they meant, since it would only say "rank", and not what rank. You had to trust that the program kept track of what you meant, and also remember what you'd intended since you can't tell afte rit's there.

Did you find it easy to create objects?

- It was very time consuming. Duplicating previous objects would allow for quicker creation and more uniform designs.

Did you find it easy to add properties to objects?

- They were easy to add, but they didn't stand out as different from Objects. Also, type checking is good. I was able to create the integer "dfdf".

Were there any properties you felt you couldn't add to the objects? Explain.

- I couldn't tell. I didn't get much idea what properties would be used for.

Was the UI intuitive (Did the buttons, objects, etc. do what you thought they would do?)

- Add object also can add properties. That's kind of odd. Assign value might be more specific.

If given the choice would you program the same game with this UI or would you code the game by hand? Explain why.

- Given that I'm cs, I'd program it by hand. I'd feel like I have more control and more certainty that it will work as I expect.

Name: Drew
Major: Interactive Media and Game Development
Time: 20 minutes

Do you believe (based on the response from the UI) that you programmed the individual rules correctly?

- Nope.

Do you believe (based on the response from the UI) that you programmed the entire game correctly?

- No—I do think I did an okay job on a small portion of the game, though.

Did you find it easy to program the individual rules?

- Yes

Did you find it easy to program the entire game?

- The part that I did program, yes.

Did you find it easy to create objects?

- Yes.

Did you find it easy to add properties to objects?

- Sure.

Were there any properties you felt you couldn't add to the objects? Explain.

- No board interaction. There was a lack of actual piece movement in the game I programmed.

Was the UI intuitive (Did the buttons, objects, etc. do what you thought they would do?)

- More or less, yeah.

If given the choice would you program the same game with this UI or would you code the game by hand? Explain why.

- I'd use this, 'cause it does make things faster and does things for me, and I'm really lazy.

References

1. GFP0607, “Domain Specific Techniques for Creating Games”.

2. "visual programming." *The Free On-line Dictionary of Computing*. Denis Howe. 21 Apr. 2008. <Dictionary.com http://dictionary.reference.com/browse/visual programming>.

3. "LEGO Mindstorms NXT." National Instruments. 2008. National Instruments Corporation. 23 Apr. 2008 <http://www.ni.com/academic/mindstorms/>.

4. Microsoft Visual Programming Language "VPL Introduction." VPL Introduction. 2008. Microsoft Corporation. 23 Apr. 2008 <http://msdn2.microsoft.com/en-us/library/bb483088.aspx>.

5. "Mindscript Download." TopShareware.Com. 2008. 23 Apr. 2008 <http://www.topshareware.com/Mindscript-download-36457.htm>.

6. "Alice.Org." Alice.Org. 2008. Carnegie Mellon University. 23 Apr. 2008 <http://www.alice.org/>.

7. Hopcroft, John, Rajeev Motwani, and Jeffrey Ullman. Introduction to Automata Theory, Languages, and Computation. Second Edition. Boston: Addison-Wesley, 2001. 169-179.

8. Stone, Debbie, Caroline Jarrett, Mark Woodroffe, and Shailey Minocha. User Interface Design and Evaluation. San Francisco: Morgan Kaufmann, 2005.