A Case Study on Refactoring

Chris Andringa
Department of Computer Science
University Wisconsin – Eau Claire
andrincc@uwec.edu

Steven Ratering
Department of Computer Science
University Wisconsin – Eau Claire
raterisj@uwec.edu

Abstract

This project began with the implementation of the game of Risk in the computer language Java. Risk is a game where up to six players try to take over a simulated world using one's armies. Either a person or the computer can control each of the players. The computer-controlled players would then use one of several strategies. One objective of this project is to refactor the original implementation of Risk using two object-oriented design patterns. The Model-View-Controller pattern will be used to separate three closely related parts of the program: the model of the simulated world, the graphical view the user sees, and the handling of user inputs to control the game. The Strategy pattern will be used to encapsulate different strategies for the computer-controlled players. Finally, we want to find and apply other refactoring patterns.

Introduction

This project came about as a means to challenge us. Unsatisfied with Hasbro's computer implementation of Risk, and looking for a significant Java project, we designed and implemented the game of Risk in the Java programming language. Risk is a game where two to six players are in control of various countries and are given a specific number of soldiers to place in their countries. These soldiers are then used to both defend their countries and attack other players' countries. The outcomes of these battles are decided by rolling dice. One or both opponents lose soldiers in each battle. The player who controls all the countries on the board in the end is declared the winner. Our implementation allows each player in the game to be controlled by either a person or the computer. When the computer controls an army, the army is given some measure of artificial intelligence in the form of a strategy.

A Java program is composed of a collection of classes and one of the classes in our first implementation handled too much. It represented the model of the Risk world, the view the user sees, and the handling of mouse clicks and keystrokes which control the play of the game. This class also has a spot to plug in one of three different artificially intelligent strategies.

As object-oriented programming has become more popular, a number of design patterns have emerged. Each design pattern is a template that can be used to solve a class of commonly occurring problems. One of these design patterns, the Model-View-Controller Pattern, is perfectly suited for the large class in Risk mentioned above. We have split this class into three interacting classes: one to handle the model of the world, one to handle how the world is graphically displayed, and one to handle the user's inputs of mouse clicks and keystrokes. The new program runs just as the original, but the code is more modular and thus easier to modify.

Our current task is to repackage the strategies for the computer-controlled armies using another design pattern, the strategy pattern. In the current Risk implementation, the user can choose if an army is controlled by the computer or the user. An enhancement would be to let the user choose different strategies for computer-controlled armies.

This project is a case study in refactoring legacy code using design patterns. Most software development efforts in industry deal with legacy code. Design patterns enable software developers to use successful designs and architectures to make their code more flexible, elegant, and reusable.

The Transformation of Risk

Our original implementation of Risk was very simple in its form. Most of the program was contained in only one file. The new, refactored version follows the Object Oriented Programming (OOP) style program more closely than the original. Using Fowler's refactoring book [2] and the Gang of Four's design pattern book [1]; we were able to plan the refactoring. Using the Model-View-Controller (MVC) design pattern, we divided the large class into three distinct classes. Having three classes for these three components makes modifying or replacing one of these components much easier. After we achieved this first goal, we started adding other design patterns, such as the Strategy and Factory patterns to clean up the implementation further.

The Model-View-Controller Design Pattern

The Model-View-Controller design pattern was first conceived and documented in 1978 and was first used primarily in the programming language Smalltalk 80. Since then many programs have used this pattern but many programmers probably do not even realize they are using it. The basis behind the design is that there are three distinct parts of a program: the internal data (the Model), its presentation (the View), and the means to update the model (the Controller). These three parts work together to run one program. One would think that this means having three classes to house these parts, but Martin Fowler discusses using only two classes. Fowler says, “The gold at the heart of MVC is the separation between the user interface (the *view*) and the domain logic (the *model*)” [2]. This means that there are only two parts, as the view and the controller are combined into one part. This is how the original Smalltalk 80 utilized MVC and refactoring in this way is best accomplished in small steps. The class was first divided into 2 parts since it will then be easier to remove the controller from the view, making this a 3-tier system.

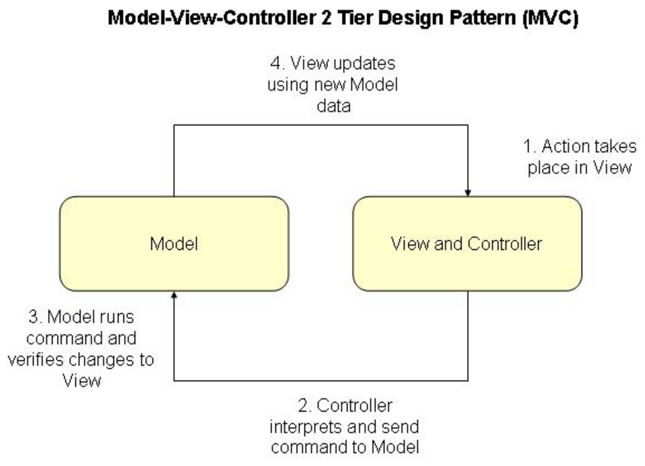
Model-View-Controller 2 Tier Design Pattern (MVC)
4. View updates using new Model data
1. Action takes place in View
Model
View and Controller
3. Model runs command and verifies changes to View
2. Controller interprets and send command to Model

Figure 1: 2-Tier MVC Design Pattern

The 2-tier design of MVC (Fig. 1) combines the controller and view into one file, while the data and model is contained within another file. As a 2-tier system, the client inputs an action into the view via mouse or keyboard and then the controller will interpret this action. After the action has been analyzed, the appropriate command is sent to the model. To accomplish this, the view has a reference to the model. When the model gets the command, it will execute the command, update its data accordingly, and notify the view of the changes. To accomplish this, the model has a reference to the view. Finally, the view/controller will update its information to show the changes. After this 2-tier system was developed, we evolved it into a 3-tier system. The only benefit to using a 2-tier to the 3-tier is that if you were teaching someone to use MVC, then the 2-tier would be more easily derived.

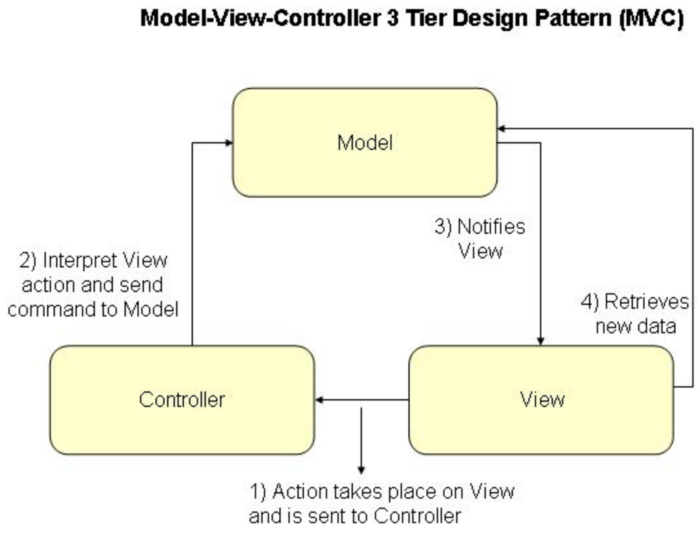
Model-View-Controller 3 Tier Design Pattern (MVC)
Model
3) Notifies View
2) Interpret View action and send command to Model
4) Retrieves new data
Controller
View
1) Action takes place on View and is sent to Controller

Figure 2: 3-Tier MVC Design Pattern

This 3-tier system (Fig. 2) follows the same path as the 2-tier but removes the controller from the view and makes it a new class. This allows the software developer to change either the controller or the view without the changing the other. This is the major benefit of using a 3-tier system instead of a 2-tier. It is in this 3-tier design that we are able to incorporate the Strategy and Factory design patterns into the Risk program without having to change the view or controller.

The Strategy Design Pattern

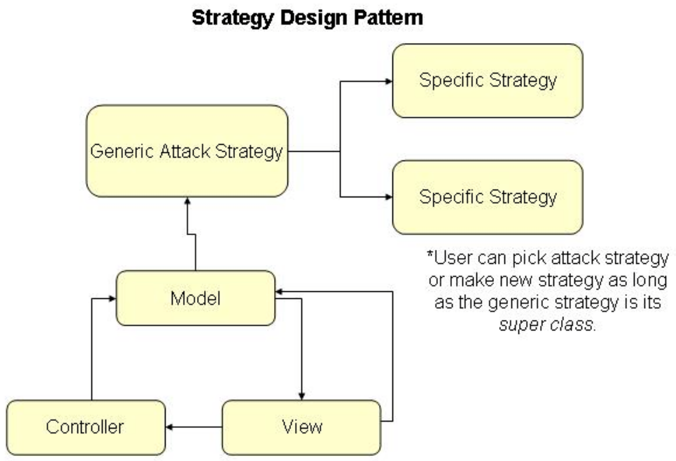
Strategy Design Pattern
Specific Strategy
Generic Attack Strategy
Specific Strategy
*User can pick attack strategy or make new strategy as long as the generic strategy is its *super class*.
Model
Controller
View

Figure 3: Strategy Design Pattern

The Strategy design pattern (Fig.3) is our way of changing the various attack patterns and decisions made by a computer player. We wanted a way for the software developer to be able to add new attack strategies to the game without having to rewrite a whole body of code in the model and this pattern gave us the idea to do so. By making the attacks used by the model a series of parent class commands--we can make a series of children classes to work under this parent. This is called inheritance and now a software developer can make as many children strategies as desired and only needs to change one line of code to test his or her strategy. As long these children classes contain the parent strategy methods then the game will run under these new strategies. This new strategy class would then control how the computer attacks and how the computer will adjust its armies in the various countries. The software developer can program famous army strategies of the past or come up with some of his or her own. This design pattern does not replace the original model class but rather enhances it by having the model call the strategy in a method call. After this project, we hope to expand more on these various strategies and add more artificial intelligence to the game.

The Factory Design Pattern

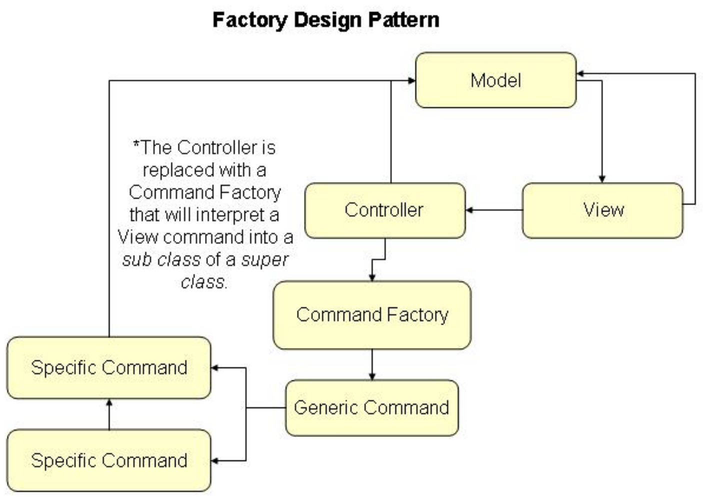
Factory Design Pattern
Model
*The Controller is replaced with a Command Factory that will interpret a View command into a *sub class* of a *super class*.
Controller
View
Command Factory
Specific Command
Generic Command
Specific Command

Figure 4: Factory Design Pattern

The Factory design pattern (Fig. 4) is one of the reasons that a 3-tier MVC design pattern is better than a 2-tier because we can more easily interchange the original controller with a new one. This design pattern consists of a factory class, a parent command class, and several children command classes. This begins by an action taking place in the view. The factory receives this action and compares it with the factory's own list of actions. If the action matches one of these commands then it "manufactures" a new instance of this command and the model runs the command. This pattern allows the software developer to make new actions in the view and all that needs to be changed is to make a new children command class. The developer would also have to update the list the factory uses to reflect this new action. The pattern also allows an author to debug the Risk program easier because if an action fails, then a default failure command can be created to notify the author without crashing the computer.

Use of Polymorphism and Inheritance

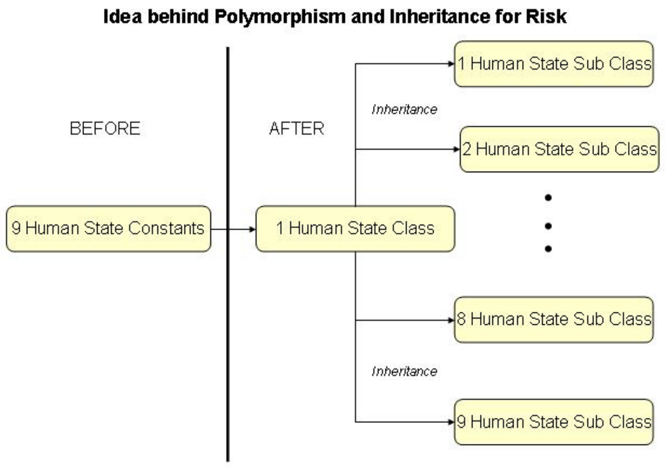

Figure 5: Polymorphism/Inheritance

Lastly, the use of polymorphism and inheritance was a large part of our refactoring. The original Risk used a variable called Human State to determine certain actions in the controller, but the controller did so in a series of if-then-else statements. This cluttered the code and made it hard to read. Using polymorphism cleaned up the code and made refactoring it easier. By making a generic Human State and a variety of children classes that “inherit” abilities from their parent class, we can move the code from the if branches into their respective child classes (Fig.5). This helps to remove repeated code and allows the software developer to simply add more HumanStates as desired. Polymorphism was also used by our Factory design pattern and our Strategy design pattern, which shows that polymorphism and inheritance play a large part in refactoring.

Conclusion and Further Work

Many conclusions can be drawn from our refactoring of Risk. This project was a chance to challenge us, but what if this project was given to a group of college students? A lot of work in industry is legacy code and students with experience working with legacy code will be more prepared when they become employees. This Risk program could be a way of training college students how to refactor code earlier in their careers, and then they would be more effective in the work force later. Second, we find that on average our files and their contents are smaller and more organized then those of the original program. By using these design patterns, we were able to make it easier for someone later to come along and possibly refactor our code farther. Finally, we find that there must be a logical end to how far one can refactor. A program can always be done more effectively, but through time constraints it may not be profitable for a company to continue refactoring a program. This is the case with Risk as we begin a new project to include more AI into the program to further the intelligence of the computer-controlled players. One could include a backtracking algorithm to guess the best possible move or another algorithm that could be faster, the possibilities are endless. Through this process, new design patterns may be formed from the combining of old ones in projects such as this. A new design pattern may come from our installing AI into the program. Another area for further work is to make the program a web application.

References

1. Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Massachusetts: Addison-Wesley.

2. Fowler, Martin. (1999). *Refactoring: Improving the Design of Existing Code.* Reading, Massachusetts: Addison-Wesley.