MQP: Design Patterns in Video Game Programming

Lukasz Zawada
5/21/2013
Contents

Abstract: ................................................................................................................................. 2

1. Introduction .......................................................................................................................... 3

2. Background Information: .................................................................................................... 5
   2.1 What are Design Patterns? ........................................................................................... 5
   2.2 Creational and Behavioral Patterns ............................................................................ 5
   2.3 Singleton Pattern .......................................................................................................... 6
   2.4 Factory Pattern ............................................................................................................. 7
   2.5 Memento Pattern .......................................................................................................... 7
   2.6 State Pattern ................................................................................................................ 8
   2.7 Strategy Pattern ........................................................................................................... 9

3. The Game ............................................................................................................................. 10
   3.1 Background and Design ............................................................................................. 10
   3.2 Reason for Use ............................................................................................................ 12
   3.3 Class Design ................................................................................................................ 14
      Singleton ............................................................................................................................. 16
      Memento ........................................................................................................................... 17
      Factory ............................................................................................................................... 18
      Strategy ............................................................................................................................. 19
      State .................................................................................................................................. 20

3.4 Assets ................................................................................................................................ 21

4. Observations ......................................................................................................................... 22
   4.1 Principles of Object Oriented Programming ............................................................... 22
   4.2 Usefulness of UML ...................................................................................................... 23
   4.3 Modularity of the project ......................................................................................... 23
   4.4 Deep copy vs Shallow copy ....................................................................................... 24
   4.5 Artificial Intelligence ................................................................................................. 25

5. Results and Conclusion ....................................................................................................... 28

References .............................................................................................................................. 30
Abstract:

We built a game, Reversi++, to identify the advantages and disadvantages of using Object-Oriented Design Patterns in a video game environment. We evaluated the patterns’ usefulness when designing the game. With the end result we demonstrated that the design patterns are great tools to create a video game because its design is flexible for revisions as it is being built.
1. Introduction

Software design patterns are techniques used by programmers to achieve robust source code that can easily be tested, fixed, added onto, or even replaced. With Reversi++, a game roughly based on the classic board game Reversi, we hope to weigh the significance of why should these design patterns be used when creating a video game. The game was created using C# and Microsoft’s XNA libraries not only because its libraries simplify many of the low level aspects of programming, but also because the combination was used in numerous titles for games on Xbox 360 and Windows from indie developers to ones that were published. Reversi++ was programmed with the design patterns in mind and the impact that they have on the game through its lifecycle.

For Reversi++ we used as many patterns as we deemed necessary for its design, but never used one for the sake of using it. Every pattern has a justifiable use, and some were used multiple times, while others were only used just once. One pattern, called the State Pattern, was cut altogether because it was not a perfect solution to a problem we encountered.

Reversi++ is actually a collection of two game types. Game 1 is more straightforward of the two. Players take turns putting a game piece on an empty tile that is next to an opponent’s piece to capture it. All the adjacent tiles of the newly acquired tile that belong to the opponent are then flipped and captured as well. Empty tiles are not captured by the flipping motion. The game ends when at least one of two conditions is met. First, if a player fills a diagonal row completely with their game pieces, that player then becomes the winner. This condition supersedes the second game ending condition in case both are met at the same time. The second condition under which the game ends is when the entire board is filled and neither player can put a game piece down. At which point the player with the most amount of captured tiles is the winner if the first condition is not met. On a side note, capturing all of the opponent’s pieces before the game ends is not a significant condition within the game’s rules.

The second type of game introduced in Reversi++, also known as Game 2, is basically Reversi played on a hexagonal board. The two main differences, between the two games is simply that in this version the way one captures opponent’s pieces is to surround them from two opposite sides, and the only way to win is to have more pieces when neither player can make a
move. A more detailed description of the original game with images can be found in section 3.1 and we urge to review that section for better understanding.

Throughout the iterations of the game, we saw that the design patterns were best utilized when coupled with object-oriented design principles. These principles were identified by the four authors of a major design patterns book. In addition, neglecting either of the two led to not only adding of additional classes but also revising existing classes that may not work with the new additions.

End result is that Reversi++ is a playable game and the code is flexible, modular, and easy to understand. The patterns helped bring the game creation to a more iterative building style because each responsibility was extracted to its own class or function. Each class was tested and integrated without much worry that it would cause other parts of the code to break because the dependencies were on the functional existing classes, and never the other way around. In addition, even though the game is complete, the modularity of the game allows for expansion in many ways such as more game types, more players, more new board designs, and completely new AI algorithms. Thanks to the patterns many classes also became decoupled from the rest of the project and could become a library which then could be used in other projects.
2. Background Information:

2.1 What are Design Patterns?

Design Patterns is an idea that a system can be broken down into modular parts of a project. An architect named Christopher Alexander originally conceived patterns in his book on architecture A Pattern Language: Towns, Buildings, Construction [1] published in 1977. The book was written under the impression that much of urban planning had a need for answers to common problems. He describes areas using general terms such as a waiting room, despite the actual terminology may depend on the building type and use. By 1987, Reid Smith [2] wrote a paper asking for a new way of thinking in terms of program design based on Alexander’s work. Later that year, Kent Beck [3] gave a workshop at OOPSLA on just that. By 1994, four skilled programmers named Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides authored a book called Design Patterns: Elements of Reusable Object-Oriented Software where they do just what Reid Smith spoke of.

As with architecture, where different rooms and their respective needs can be assembled to create a building, programs can also be reduced to parts with certain responsibilities. A waiting room in a building can be equivalent to a state of the program where it waits for input. The design of which rooms can be bathrooms (which often depends on where the main water source is on each floor) can be comparable to something called a Factory Pattern where arguments passed result in a specific implementation of an interface to be created. Here, a Room is an interface and Classroom and Bathroom would be instances of the interface.

2.2 Creational and Behavioral Patterns

Design Patterns come in four different categories: Creational, Structural, Behavioral, and Concurrency [4]. Creational and Behavioral Patterns are used in Reversi++. Creational patterns are ones that manage the creation of instances and what implementation is created. Behavioral patterns are used when certain classes need to fill a certain role and act a certain way. A good example to contrast the two is a Factory pattern and Memento pattern, respectively. The Factory pattern listens to requests and creates different types of instances based on the request. The
Memento pattern only deals with saving already created objects and reading from them. Note that a class can use both creational and behavioral patterns together. It can also use multiple patterns of each at once, however, the object oriented design is best used when a class has as close to one responsibility as possible. This decoupling of responsibilities helps keep the entire system modular and takes full advantage of the object oriented design. Below are the Patterns used for the design of Reversi++, they are later shown where and how they are used in Section 3.3.

2.3 Singleton Pattern

Singleton Pattern creates and manages an instance of a class where only one is allowed to exist. It achieves that by making the constructor private, reducing the scope of the object down to a static local. Because the object is static to that class, the only way to gain access through it is by use of a function getter. The end result is one object universal to the entire project.

![UML diagram of Singleton pattern](http://www.oodesign.com/singleton-pattern.html)

**Figure 2.3.1: Singleton Pattern. Retrieved from http://www.oodesign.com/singleton-pattern.html**

In Figure 2.3.1, the Singleton class has a function `getInstance()` that returns the instance of the class object. Note that in this UML diagram, the class points to itself because the implementation of the class is limited to itself.
2. 4 Factory Pattern

Factory Pattern creates an object at runtime based on some arguments and returns the result.

![Diagram of Factory Pattern](http://www.oodesign.com/factory-pattern.html)

**Figure 2.3.2:** Factory Pattern. Retrieved from http://www.oodesign.com/factory-pattern.html.

In the Figure 2.3.2, the Client makes a request from the Factory to create a Product. The Factory then returns a Product Implementation based on the interface of said product. All products should implement the same interface because the client uses the interface to know what functions all products have in common. All products should work with what the Client expects.

2. 5 Memento Pattern

The Memento design pattern allows for an object to create a copy of a state of an object that is saved and can be referenced to at any point, like a checkpoint. To create a memento, first the Originator class must contain a state. Using the `createMemento()` function, the Originator copies the state of an object in an instance called a Memento. The Memento object contains the state, and is able to set its state to another instance or returns the current state it has. The last class, the Caretaker holds all the Memento instances. Whether its capacity is one or many is up to the implementation and need. The Originator uses the CareTaker for a Memento. Again,
whether the Originator can ask for any state as opposed to most recent is dependent on the implementation of the CareTaker which can have a List of Mementos or a Stack, or any other way of holding the information.

Figure 2.3.3: Memento Pattern. Retrieved from http://www.oodesign.com/memento-pattern.html.

2.6 State Pattern

The point of using the state pattern is to alter its behavior during use depending on the context at runtime. The big picture is that a State class tends to act as if it changes classes based on a request. It is often an abstract class with different implementations that function the same way, but act entirely different from one another. To switch its behavior, a Context redefines the state object to another implementation. This is done with inheritance as opposed to interfaces.

Figure 2.3.4: State Pattern. Retrieved from http://www.dofactory.com/Patterns/PatternState.aspx.

In Figure 2.3.4, a Context class includes an object of a State. It uses the Request function to call the state object’s Handle function. Inside that function, it may redefine the object itself to another State implementation should it be necessary. ConcreteStateA object may redefine context.state to ConcreteStateB object and vice versa.
2.7 Strategy Pattern

The strategy pattern is used to define behavior or algorithm that is used based on a specific context. The Context class holds an implementation of the Strategy, which it uses on its data. The context does not know what the implementation does. A client delegates which strategy should be used by the Context. The client then gives an implementation to the Context prior to Context needing to use it. Frequently, the client then uses the Context to manipulate the data as opposed to operating on the data by itself. Sometimes, it is another client that uses the Context after the first client sets it up for use.

![Strategy Pattern Diagram](http://www.oodesign.com/strategy-pattern.html)

Figure 2.3.5: Strategy Pattern. Retrieved from [http://www.oodesign.com/strategy-pattern.html](http://www.oodesign.com/strategy-pattern.html).
3. The Game

3.1 Background and Design

The game described in this paper is a modified version of Reversi [5]. Reversi is a two player game, played on an 8 by 8 board. Each player is represented by a color, and they take moves placing a game piece of their color onto the board. The rules of placing a piece on the board is that a player must surround a line of opponent’s pieces. It can be vertical, horizontal, or diagonal; the line segment results with two of player’s pieces at end points with arbitrary amount of pieces of an opponent in the middle. When the turn ends, the player who just moved gets to replace the game pieces inside the segment with his or her color. If the pieces are arranged in such a way that the current player cannot put down a piece and flip the opponent’s pieces, then the opponent gets to move again. It is possible that neither player can put a piece down. The game ends when all spaces on the board are filled or neither player is able to make a move. The player with most pieces on the board at the end of the game wins.

Below images were taken from http://www.pressmantoy.com/instructions/instruct_othello.html and are quite helpful in explaining the more complicated instances of the game.

![Figure 3.1.1](image1)

![Figure 3.1.2](image2)

By placing a white piece in the lower right corner in Figure 3.1.1, seven black pieces are flipped with the white player winning the game. The result of this move is seen in Figure 3.1.2; we can see that every black piece has been flipped and therefore no matter where the black player puts a piece down, he will not be able to surround a white piece. Also note that creating rows in multiple ways does flip pieces in every way possible, not just one of the 3 directions.
In Figure 3.1.3 we see a black piece being placed over the row of white pieces. In Figure 3.1.4 we see that flipping pieces downward does not cause a chain reaction to the right even though the white pieces are now being surrounded by black pieces.

Note to people who have played *Othello*: Reversi and Othello are practically the same game. The difference is that Othello is a trademarked name of Pressman Toy Corporation [6] with rules as specified by the company.

The game designed and used for this project is called *Reversi++*. The major difference is that it uses hexagonal board spaces allowing for six different directions of flipping the disks. The rules of this game depend on the mode the player selects. The first mode simply flips the adjacent disks to that of the player’s color. The second mode is a direct translation of Reversi onto this new board type. The objective for both games remains the same.

Thanks to the principles of object oriented design *Reversi++* is designed to have as much modularity as possible which allowed for extensive use of the design patterns. Modularity here means separating responsibilities and functionalities between classes. Object oriented design allows for multiple classes implementing the same interface to be swapped in a larger system and still work. These classes would therefore be known as modules. In *Reversi++*, implementations of different rules would be the modules that the game uses. The game does not care which module is used so long as the module attached works as expected and properly.

Design patterns allow for certain aspects of the game to be swapped out at runtime. This is great because it allows for creating modules for small algorithms that are abstract and do not
merit a whole object to be created. For example, one would probably not create an Adder class in a calculator for simple function of addition, then another for substitution. Instead, one would create a function that allows for such functions to be executed at runtime with use of some deciding algorithm. Artificial intelligence in Reversi++ is of this nature because of the multiple game types and levels of artificial intelligence. There is an AI player class and its decision algorithm is not defined when it is created. Instead, it is fed the appropriate algorithm as a callback function that it will then use for making moves. This way, there is only one AI player class as opposed to multiple classes that only differ by one function. Likewise, the class that has an AI Player only needs to know one function of the class: move(). All move functions have the same responsibility and return the same type of data. How they go about coming to that conclusion is not considered by the game so long as the data works with what the game expects.

Using these callback functions, while definitely has great advantages, could be considered bad design. In Reversi, the AI player could not function by itself. One cannot instantiate the without passing in the algorithms for it to function. This need of another class is called a coupling. Generally speaking, coupling is not a desired trait of a design because it makes refactoring and isolating responsibilities harder. If used where the coupling can be considered as one whole such as AI player and its algorithms, and they are in no way coupled with another class, the advantages can outweigh its disadvantages. The problem arises when multiple classes are coupled and a major change of design is needed. Replacing and moving classes can create problems with scope and needing objects that are no longer created in proper order, causing instantiation problems.

3.2 Reason for Use

We decided to use Reversi++ for this project in order to demonstrate how good object oriented design and principles make creation of a video game easy. A simple copy of Reversi would show that these design patterns work with Reversi alone. Expanding on Reversi in multiple ways allows for experimentation. It should prove that both object oriented design and design patterns work well when designing a game from scratch, just as it does when adapting an existing one.
An architect named Christopher Alexander originally conceived patterns in his book on architecture *A Pattern Language: Towns, Buildings, Construction* published in 1977. The book was written under the impression that much of urban planning had a need for answers to common problems. He describes areas using general terms such as a waiting room, despite the actual terminology may depend on the building type and use. By 1987, Reid Smith wrote a paper asking for a new way of thinking in terms of program design based on Alexander’s work. Later that year, Kent Beck gave a workshop at OOPSLA on just that. By 1994, four skilled programmers named Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides authored a book called *Design Patterns: Elements of Reusable Object-Oriented Software* [7] where they do just what Reid Smith spoke of.

Gamma, Helm, Johnson, and Vlissides’s book contains a compilation of patterns that the four authors found and all the common characteristics which they coin *principles of good object oriented design*. The four programmers, known as Gang of Four, point out the idea of programming to an interface, not to an implementation, which allows for multiple implementations to work with one another class that expects an implementation of an interface. To clarify, when creating an expansion of Reversi the rules of this new game did not matter – only things in common between the different sets of rules mattered. This meant we could experiment with different sets of rules, or implementations, on the fly so long as each implementation shared the same functionality as the interface. Thus the interface, in this example, included checking for valid moves, having a player move, and checking for win conditions – this was bare minimum that is needed. Note that from this interface, one does not even know it is a Reversi game. This is because the implementations need to conform to the design of the game, as opposed to the game conforming to pre-made implementations.

The focus should be directed at the design of the game while keeping the first principle and patterns in mind. If we used a large project to show the different pattern types we would have to use many of them at once. Instead, we are focusing on a few patterns and showing that they are flexible and useful which fortifies the aforementioned principle which states that one does not need to worry about the implementation. Reversi++ gives multiple opportunities for each pattern to be used. The Singleton pattern is used for the mouse handler and the instance of a
game. The Factory pattern is used to create a rule set, and could be expanded to create different board types. The state pattern is used to change how the game updates – whether a person is in a menu state, or a game state. The smaller size of the project allows for a smaller interworking of the patterns which helps isolate and explain each pattern individually. It is just simpler to pick apart a small project and identify where the patterns were used in isolation, and where they are used in conjunction, building off of each other.

3.3 Class Design

![Full Class Design Diagram](image-url)

Figure 3.3.1: Full Class Design Diagram.
Figure 3.3.1 is the full diagram of the Project demonstrated using Visual Paradigm 10.0. Before getting started with each individual parts, it is easier to walk through the program itself. The game starts in an XNA generated class named Program.cs. It creates a game object and that’s where the main loop takes place. What we did was create multiple menu classes that use the GameState singleton to determine which menu is the player currently in. From the user’s perspective, the active class is the one being drawn onto the screen, and the one player’s input are matched with.

After the player moves from the main menu, he faces an instruction page where he can hover over buttons which determine a small screen with each of the modes’ rules. These buttons, when clicked, initialize the specific game using a factory, and change the state to the game. The game begins promptly after.

The game class is the ReversiGame singleton and it holds all the information about the game. It is separated from the menu system with only shared classes being singletons and the Button class. Reversi++ game instance contains (moving clockwise) an AIPlayer which in turn has an AIInterface implementation, a user interface implementation provided by a factory, a ruleset specific to that game, a game board that works with the game type, and lastly, a caretaker class which saves the instances of the board as part of the checkpoint system.

Other useful information include that within the diagram, we did not specify explicitly using UML that every class that gathers mouse input uses the mouse handler, and that whenever a class has art it needs to load, it uses the content loader singleton. This information in the diagram was purposely left out to minimize the amount of lines thus complicating the diagram needlessly. It should be pretty clear that whenever a user needs to click on a button to proceed that the mouse handler is then used to capture the information.
Singleton

Figure 3.3.2: The Singleton Patterns are identified by the green rectangles.

This diagram shows singletons in the game. We chose the GameState to be a singleton because it allows the game to know which menu to draw and update – including the game itself. CurrentPlayerSingleton is a singleton because some rules need to know which player is being used. Other rules classes may handle that internally. For example, in game 1, each player is guaranteed to make a move, in game 2, one of the players may have no valid moves and the opponent will take consecutive moves until the first player can move. FontWriter class is used to print letters onto the screen of the game. MouseHandler class captures mouse input, and ContentLoader loads 2D Textures into memory.
Memento pattern is shown in this diagram as the three classes fully covered by the pink rectangle. The ReversiGame singleton contains an instance of a Board. The Caretaker does not initially start with a board of its own. Instead, the ReversiGame passes an instance of the current board to the Caretaker and may at any point ask for it back. The implementation of the Caretaker in Reversi++ is stack based. It does have additional functionalities such as the ability to clear the entire stack when the game is reset.
Despite the usefulness of a factory pattern in a modular system, there were only two instances where a factory class was completely justified to be used. The first was the Reversi++ game instance. The game can come as a two player game, or an AI. There are also different levels of the AI. The second instance is with the UI. The user interface may not have an undo button when playing a two player game. Just imagine playing with a person who clicks to undo every move that results with the person losing.

There are instances where a factory was not needed. One good example is the board. Since each board have one definition of how to instantiate themselves, there was no need for a factory. Each game mode just created an instance of the board it needs. A factory is used when
the instance is needed based on known parameters that can change. In this game, the parameters of the game remain static, thus a factory class would have been superfluous.

Strategy

Figure 3.3.5: The classes that make up the Strategy pattern are highlighted by the yellow rectangle.

Strategy pattern is best used on algorithms, and there is one good implementation of that in Reversi++. In this diagram, the orange rectangle shows the four classes that make up the strategy pattern implementation. The artificial intelligence of the game is split with an AIPlayer being the object that the player faces in a single player game. However, by itself, it does not know what to do. It is the two AI implementations, each game mode specific, that dictate the algorithm that is going to be used. Each of the AI implementations also has three different
algorithms for each of the difficulty levels. If all this information was within AIPlayer class, then this one class would be both large, and have too many responsibilities that could be extracted out. Now, one class does all the actions which in turn are defined outside using a system of algorithm implementations.

State

State pattern was originally going to be used with the game’s menu system however it did not fit the game’s implementation. The problem comes with State pattern’s definition. It allows an object to change how it operates based on an internal state changes. This could have been possible with just the menu system and probably still is. That being said, certain menus have buttons while others don’t; and different menus lead to multiple others which quickly became a complicated juggling act of keeping track of multiple variables and functions which differed by a
few lines. So instead, each menu page has its own class, and Instruction page has buttons which utilize the Reversi++ factory class. Afterwards, the game state is changed to game and the class Game1 draws and updates the game itself.

3. 4 Assets

The Assets of the game were made from scratch. Originally, we opted for placement art before it was replaced by final art. Final art was created by Pawel Zawada who is an older brother of the programmer of Reversi++. Technical assets for Reveri++ were also programmed from scratch with use of UML diagrams of various sources (see references) for guidance. The game is written in C# using XNA and .NET frameworks for Windows. The framework libraries allowed for easy display of textures and input reading.
4. Observations

4.1 Principles of Object Oriented Programming

We noticed that throughout the project certain object oriented principles [8] were very useful when programming. Though they are not the focus of this project, they were still a very important part of design patterns. The principles do have their advantages and disadvantages, and we stumbled upon a few instances that highlight what happens when a programmer neglects the principles or follows the principles too strictly.

The principle of programming to interfaces, not implementations, kept adding new classes relatively easy since they all work exactly the same even if their functionality is not all that similar. Neglecting to use this principle, however, hindered the speed of development. A good example is when the board for game one was being developed. Unlike game two, where the board is relatively easy to create as it is square with every other row being offset, board for game one is diamond shaped. Originally, there was one Board class, and when another board needed to be created, this called for a Board interface. Creating the interface was rather easy; the problem that arose was that every class which referenced a board needed to reference the interface. This reliance on a specific class not only added a dependency, but also kept us from working on a board design straight away.

At times, sticking to a principle caused the classes to become large. The Ruleset classes have that problem because different classes were designed to require different information. For example, a regular game loop only needs to know whether a valid move is found to make sure a player’s turn is not skipped. An AI would need a list of possible moves so that it is much faster to determine which move is the best choice. Different instances need for overall same redundant process with different information at the end. Open Closed principle, or the need to never modify a function in use once it is written, allowing for only adding to a class creates large classes. While it ensures that every other section of the program referencing the old functions will work, it leaves a lot of room for extraction and refactoring. Another negative outcome is that because a new function is written in one class, if it is an implementation of an interface, now all other implementations need that function as well. A good work around is to have an abstract class that
has all the common functions in one place. However, with different boards and set of rules, while function names remained consistent between the games, unfortunately, the code wasn’t.

The single responsibility principle did not pose any problems. The principle states that every function and every class should have only one responsibility to ensure as much modularity as possible. It is this principle that is the essence of extracting functionalities into additional functions and classes. The design pattern of Memento needs a Caretaker class and a Memento class, for example. The caretaker takes care of mementos, which in turn hold data. Both of which only have one responsibility though act together to get a larger task accomplished.

4.2 Usefulness of UML

Unified Modeling Language, or UML, was used to help manage the relationships of the objects, and to integrate the different design patterns, but not without a few problems. All design patterns used in the creation of the game, have long since been documented in UML. Unfortunately, UML is not as standardized as we originally believed. Different books, websites, and programs use different notations. In fact, since the design patterns themselves are a way of tackling a problem, there are multiple implementations that get the job done based on their respective description. Each, of course, has a different UML model. We tried to stick to the least amount of sources such that all the notations carried over between illustrations. The tool we used for modeling Reversi++, Visual Paradigm 10.0, does not contain arrowheads for direction of associations. These were added in by use of a two line segments at the appropriate endings.

4.3 Modularity of the project

With the help of design patterns and principles, the game was programmed to be very modular which had many advantages. For example, the testing of the game was easy. All of the classes had different responsibilities, and whenever a problem arose, the class that is closely related to the problem is most likely the culprit. Debugging then became even easier since each function can be tested one by one for proper output. A lot of functions have helper functions which helped further narrow down the problem.
Adding new features is very often a natural evolution of a game, and Reversi++ was no different. Despite the problems with adding a new Board type, there were many instances where adding new features was relatively easy. Originally, the games were created from the main menu once the option to play them was selected. An additional menu which explained the games and gave each game a different set of options (two player, or versus a certain difficulty of AI) to choose from was needed to clean up the menu system. Plugging the new menu in was simply a process of creating the class, adding a new game state to the game state singleton, and then connecting a button that redirects the player from main menu to the new class by switching the game state.

The creation of different types of games was also easy. Since there are many implementations based on interfaces and factories attached to them, creating a game consisted of requesting proper versions of the game from the factories and setting the result as the active game. Then again, the system switches the game state to start the game itself.

### 4.4 Deep copy vs Shallow copy

Aside from object oriented design and principles, another important lesson that was learned was the difference between deep and shallow copy in C#. When copying an object to another object, only the reference is copied. So the programmer has two pointers to the same object. To resolve the issue, there is a MemberwiseClone function that allows for a new object to be created and then non-static variables be copied over. The problem remains that that’s all that was copied. In Reversi++, copying a Board object meant that the turn number was being copied but not the list of Hex objects that the board was comprised up of. For that, a deep copy was needed. Deep copies are not inherently available to a programmer in C#. The workaround is to create a clone using MemeberwiseClone function, then create another clone of each object that your clone needs to have a reference to. The result is a chain of cloning until everything is copied over depth-wise.
4.5 Artificial Intelligence

The artificial intelligence is a very integral part of Reversi++, and finding proper strategies was very important. Since different game modes have different set of rules and board types, the strategies do not carry over from game one to game two. Reversi++ also features three different AI levels for each of the game types.

The easiest AI for both games was simply to have the opponent pick one of the available moves at random. This was surprisingly a very good way of testing. While we tried to make sure everything we expected to work was being tested, the computer would make an illegal move, such as placing a piece in a corner, isolated from every other piece. This signaled that the function that checked that a piece is separated from all others was incorrectly written because there was no way to determine whether non-existing pieces were empty or not. The AI’s sporadic moves highlighted problems that would have taken additional time of random clicking to find.

The medium difficulty AI was a bit harder for game one. For this we created a greedy minimax algorithm. It looks at a set amount of steps forward and checks which outcome takes the computer to the point where there are more of its pieces than the player’s. The reason it is greedy is because it does not take into consideration whether or not any of these steps cause the computer or the human player to win. By looking only to win, we noticed the computer became very predictable and repetitive in its preferred move order.

The natural progression of the medium AI in game one to create the hard AI was to get rid of the heavy heuristic put on placing the most tiles. The algorithm will stay away from any piece placement that results in human player advantage. It also ranks its winning moves the highest in preference, so that if it sees an opportunity to win, it will take it.

Game two of Reversi++ is a bit more complicated, so while easy AI was left to be just the same as in game one, medium AI could not just look for most amount of pieces. This is where we utilized a Strategy Guide [9] for regular Reversi because Reversi++ is more or less a direct translation and the hypothesis is that most strategies should carry over – albeit harder to implement. A good strategy to start with is a mobility battle. The idea is to pick a move such that the opponent has the least amount of options. This hinders their mobility. The game, in theory,
should stay balanced until one of the players is forced to make a costly move that opens up for large gain of stable pieces by the opponent.

Other good strategies considered such as the “evaporation” strategy. In this strategy player tries to have the least amount of pieces so that they are surrounded by opponent’s pieces. Then it is easy to capture the surrounding pieces from the edges with the opponent having no way to recapture them. The problem in utilizing this method was trying to figure out when is a good time to start capturing the inner pieces, and the fact that the board has hexagonal pieces, the opponent has more opportunities to recapture his or her pieces.

“Frontiers” strategy was considered, but ultimately not implemented. The premise is very similar to that of evaporation strategy where the goal is to have the opponent surrounding the player’s inner pieces. Then slowly capture pieces that do not increase the frontier, and when they are placed on the outside, the opponent has no way or capturing them because they do not have opponent’s pieces on any side. This restricts the opponent’s mobility to the point that the player may get consecutive moves. The problem, again, is the board. Since the tiles are hexagonal, it is very hard to find pieces which cannot be captured. It is also hard, as previously mentioned in the evaporation strategy, to keep the inner pieces. None of the two strategies are optimized for Reversi++.

As with game one, game two’s hard AI utilizes a positional strategy heuristic. The difference between the medium and hard AI is that this approach takes positions into consideration and is discouraged from putting pieces in ways that would benefit its opponent. So putting a piece one tile away from a wall or a corner may lead to the opponent from capturing the advantageous corner tile. Instead, the AI will put a piece in a safer position in hopes of capturing these stable tiles itself.
In figure 4.5.1, in a regular game of reverse, putting a piece in the corner gives a value of 99, meaning it is the most favorable move. Notice that the move next to it is -8, that is because putting a piece there is risky as it may lead to a capture of the corner by the enemy. The tile to the left of that is more favorable because with it, one can capture the corner. The same train of thought is applied to the diagonal tiles away from the corner. In Reversi++, the hard difficulty AI looks for wall and corners.
5. Results and Conclusion

By the end of the project, the game that was finished ended up with a few features cut, but nevertheless could still be considered a completed game. If we had time, features such as online multiplayer could have been added. Different board types for the two games could also have added a long term value to the game.

Reversi++ is quite modular, which allows for many of the classes to be transferred to another project if desired. A lot of low complexity classes such as Hex, Button and even singletons such as MouseHandler, or FontWriter which detect mouse movement and clicks, and write text to the screen respectively, can be used in another project altogether without any dependencies that would cause errors. In fact, the entire menu system could be reused with slight modifications whether the game is created and then information extracted for the game over screen. In its place any other game can be put instead.

The code itself could also be cleaner in a few areas. Large classes could have functions extracted out into smaller functions which would make each function have one responsibility. We’re pretty sure some things that were programmed could have been simplified or have a more optimized equivalent in terms of time. Increase in processing speed would help AI computer consider more instances and would result in a smarter opponent. In a way, one could consider Reversi++ to be in late beta form where only polish and optimization is left.

In its current form the program is flexible enough to allow for new features. Features that would be very easy to implement are new AI algorithms, new board designs, and as previously mentioned, whole new game types. These game types do not need to follow suit in the rhythm of the two featured games (as defined in the class ReversiGame, which compiles all the aspects that make up both of the games and puts them together). Even online multiplayer is quite feasible, though not as easy.

If one was to add online multiplayer option to the game, outside of a server, substantial additions would have to be added in the game singleton because as it stands, the primary player is always first. While that does not cause many problems, the fix is not only in one place. As it stands, certain AI assume they are red in color – or player two. There are a few ways to fix this
but the easiest one would be to hold the data of which player is what color and then determine the order internally. As the game is turn based, it can do all the calculations without specifically looking at colors. It will instead use a variable to compare colors of tiles and players. The problem is very reminiscent of the addition of a new board problem, though in this instance, the culprit is hardcoding values not classes. The problem persists because modifying turn order was simply not part of the features listed.

The last, and probably most important, end result is the fun factor. Fun is very subjective because whereas some people like a slight challenge but still enjoy winning most of the time, there are people in this world that have fun setting themselves up for almost guaranteed failure with a very low chance to succeed then perhaps prevailing eventually by skill if not pure luck. The AI and multiplayer should satisfy a wide range of people with different ideas of what a fun game is. The two different game types may also reach out to different types of players but being that the games are very similar, that may not be the case.

There are many factors outside of design patterns that are just as important in programming a game, but the design patterns do have an irreplaceable use. These patterns work especially well with the principles of object oriented design and together they achieve a class design that is easily recognized by anyone familiar with them. Other advantages include decoupled classes that provide a modular design. Alone, they still provide a great way of organizing classes that suit certain needs; however, without tenets such as programming to the interface, the modularity suffers in our observations.
References


