REFACTORING PROJECT

Design Refactoring Documentation
Prepared by TEAM 2:

- Eric Lin <exl5325@rit.edu>
- Jarrod Cummings <jdoa@rit.se>
- Moisés Lora <mal3941@rit.edu>
- Jack Corrigan <jmc4216@rit.edu>
- Abdulrahman Alfahad <aa8356@rit.edu>
- Doanh Pham <dnp4736@rit.edu>

Product Overview

Checkers shall allow two people to play a checkers game locally or over the internet. The game only coordinates the play of the two human players and does not handle computer operated players. For each game, players shall set their name and select one of three game modes: local, host, and join. For join mode, player shall provide a valid IP address. Checkers uses built-in rules system to keep track of player's moves and gives feedback to user if it is illegal.

Domain Model

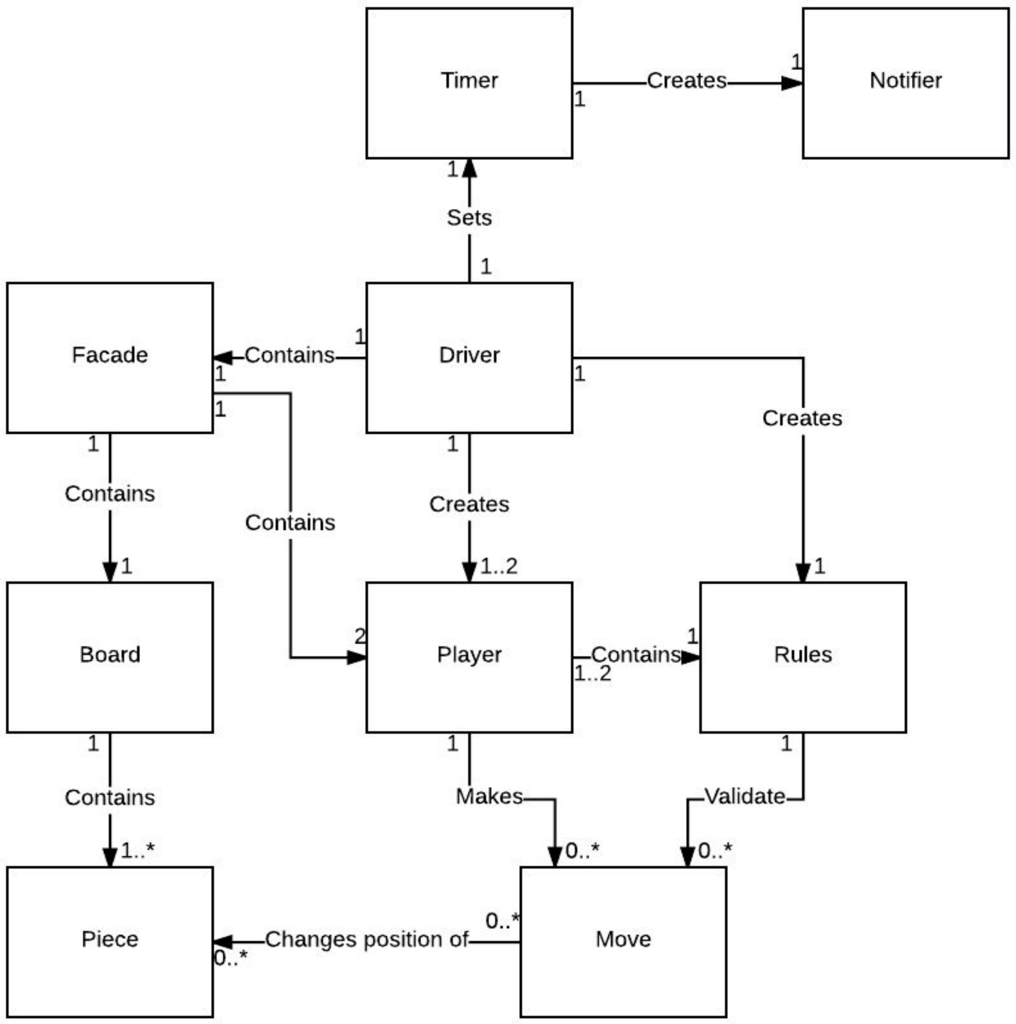


Figure #1 Domain Model

Analysis of Original Design

Design Weaknesses and Strengths

Weakness: The main weakness of the system is the amount of duplicate code and the lack of cohesion in classes. For example, the GUI builder CheckerGUI had 64 instances of duplicated code. Each new button was code that was simply copied over and over. Lack of cohesion was also a big issue in the code. For example, the Pieces object had no knowledge of how they moved, instead the Rules and Moves class both handled possible moves and illegal moves. Another example was the GUI builder. The methods were often large, unreadable methods which handled completely unrelated functionality. In short, the two biggest weaknesses were low cohesion and duplicate code.

Strength: The strength of this system is the attempted use of inheritance, although flawed. Both regular pieces and king pieces both extend from piece to promote (very little code reuse). The same goes for players: local players and network players extend from player. A refactoring would make these inheritances more useful - such as adding king moves and regular moves to the pieces. In addition, the separation of GUI windows - SecondScreen, CheckerGUI, and FirstScreen, makes it very easy to isolate potential bugs in the GUI. The design could be improved with some extract methods, code cleanup and extract classes, but otherwise, it is a decently good start.

Use of design patterns

Name:		**GoF pattern: Observer**
Participants		
Class	**Role in pattern**	**Participant's contribution in the context of the application**
Facade	Subject	The Facade class can have action listeners attached to it which will be notified when its generateActionPerformed method is called.
CheckerGUI	Concrete Observer	Attaches itself to the Facade as an action listener when calling the Register method from its constructor.
Action Listener	Observer	Interface from the AWT events package. Interface for listeners that receive action events.
Deviations from the standard design pattern: No deviations		

Subsystem and Class Structure

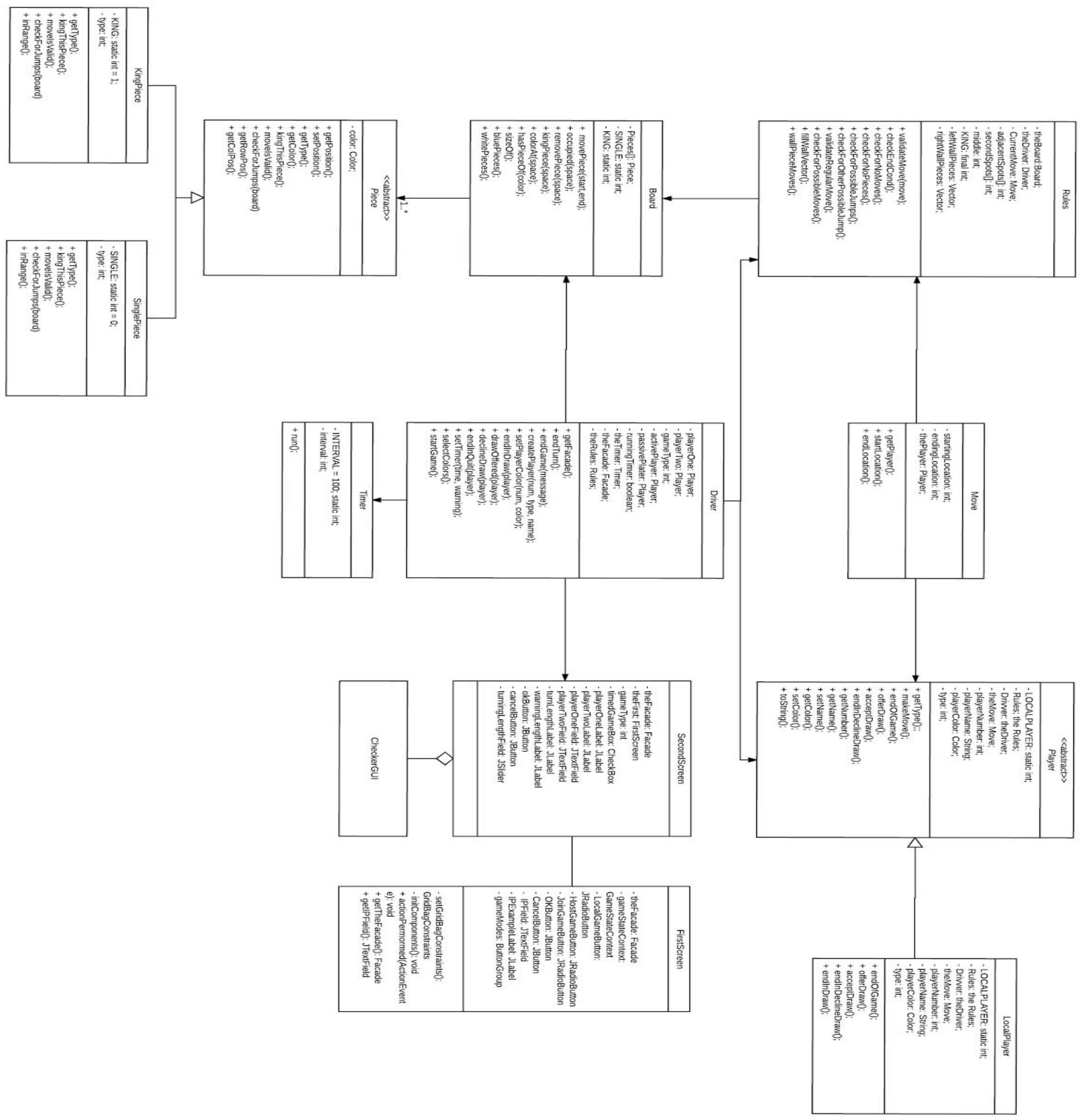


Figure #2 Original Class Diagram

Sequence Diagrams

GUI Sequence Diagram - Old

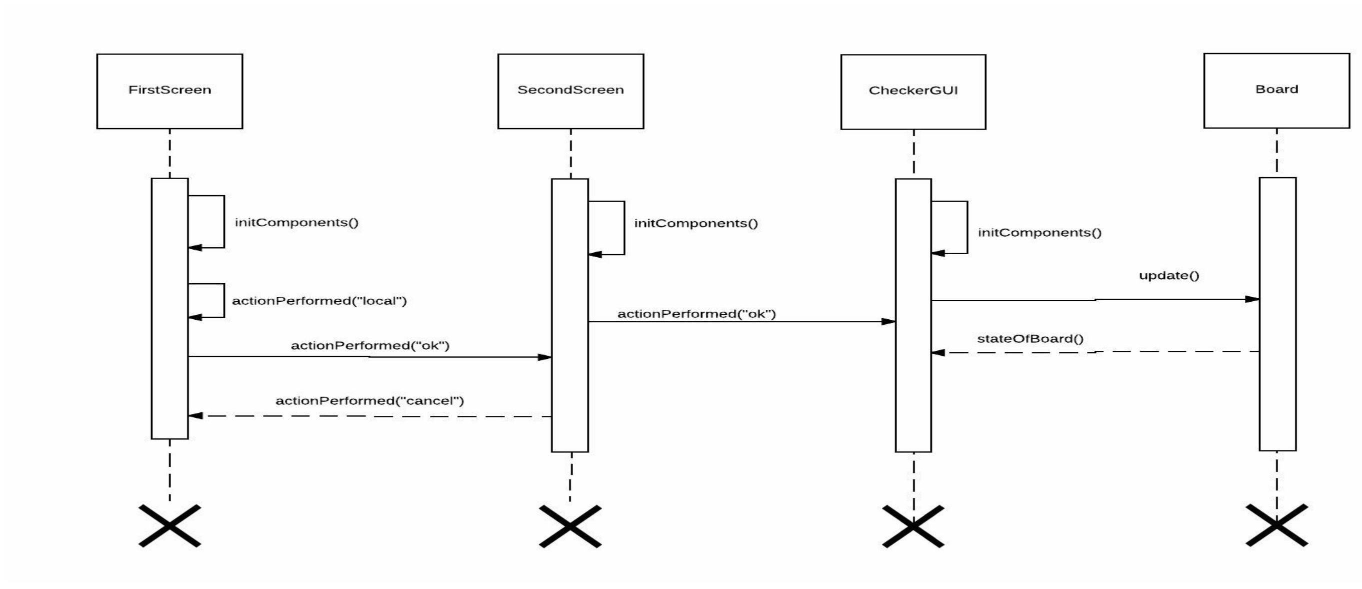


Figure #3 GUI Sequence Diagram (Original)

King Piece Wall Move Sequence Diagram - Old

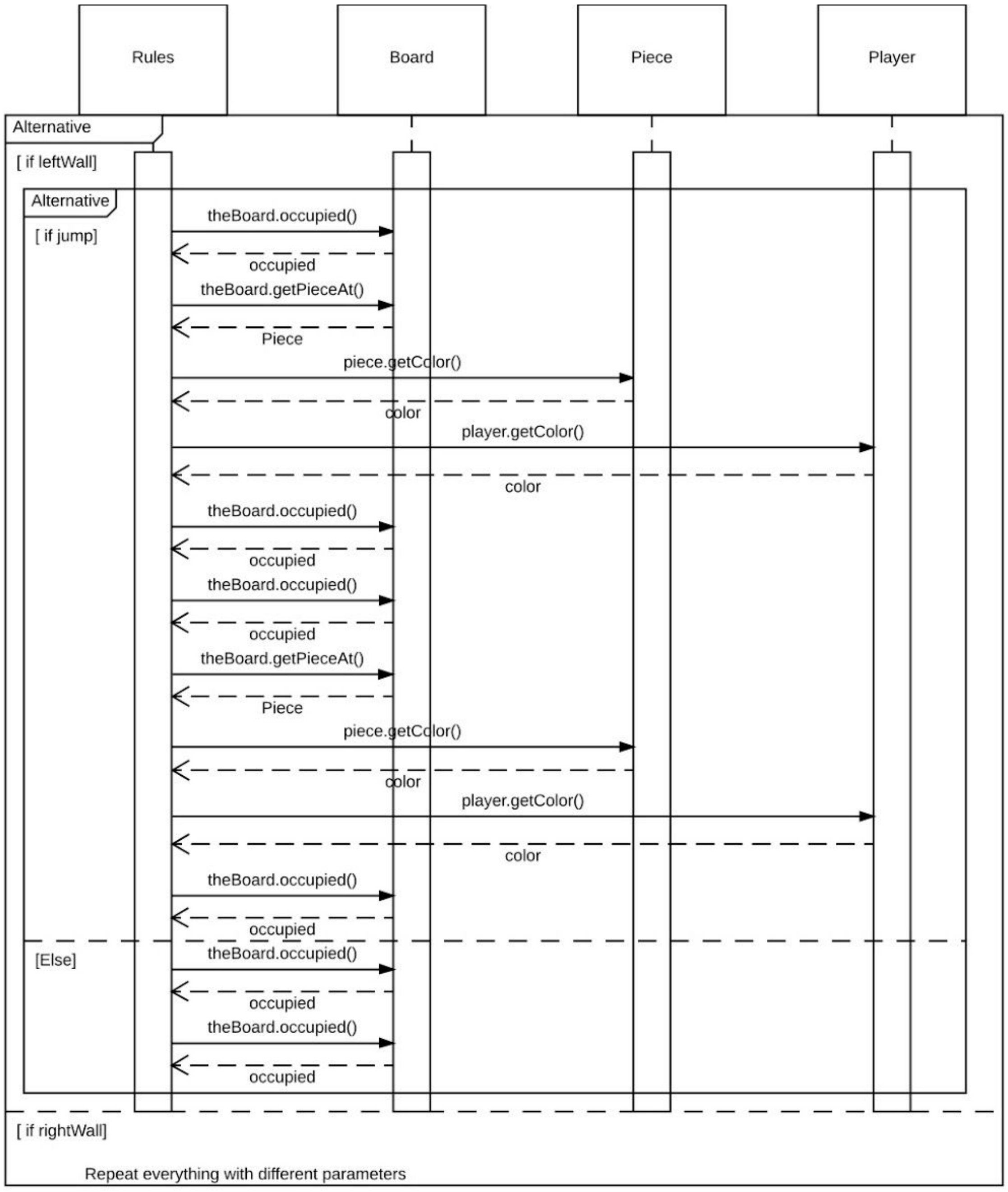


Figure #4 King Wall Move Sequence Diagram (Original)

Metric Analysis

Lines of Code: The most basic metric is the number of lines of code in a system. One of the most noticeable aspects of this system was amount of duplicate code. Rather than using shortcuts (such as loops or method calling) to quickly perform duplicate functionality, a lot the same code was simply copied and pasted over and over in multiple areas. As a result, there was a large number of lines - according to Git, 6609 lines of code in the original system. The amount of duplicate code and large LOC value meant that there was a lot of potential to refactor and remove duplicate code, resulting in a smaller number of lines of code.

Cyclomatic Complexity: Cyclomatic complexity is defined as the number of linearly independent paths in a module. This metric is positively correlated with the number of errors - the higher the complexity, the more likely there are to be errors. Two of the cyclomatic complexity values that stood out were the action handlers in the Firstscreen and Secondscreen class. Firstscreen's actionPerformed method had a cyclomatic complexity of 11, while Secondscreen's actionPerformed() method had a cyclomatic complexity of 13. This meant that there were 11 and 13 independent paths through these methods. This was supported by visual evidence - both methods had large conditionals with nested try catch blocks, making it a prime candidate for refactorings such as extract method, extract class, decompose conditionals, etc.

Weighted Method Complexity: The weighted method complexity is equal to total cyclomatic complexity in the class. The Rules class WMC was valued at 127, which is the highest of all classes in the system. This shows that there is probably a large method that requires a method extraction in the class, or even an extract class to remove unrelated functionality

Average Design Complexity: Design complexity is a metric related to cyclomatic complexity - the number ranges from 1 to the cyclomatic complexity, indicating how "complex" a system's design is. The design complexity is 3.33/3.80, which is a high value. This indicates that the code base could probably be reduced in size. In addition, introducing a design pattern should reduce some of the complexity values

Number of Cyclic Dependencies: Cyclic dependencies are the number of classes or interfaces a class directly/indirectly depend on. The average cyclic dependencies was 3.05, which is a lot lower than expected. This is probably because each class has so much functionality contained within it that dependencies are very low. After refactoring, when new classes are introduced, this value will most likely increase.

The Refactored Design

The refactored design adheres to principles such as separation of concerns, high cohesion, and extensibility. The state and decorator refactorings attempt to follow separation of concerns. With the state refactoring, all of the game mode functionality was moved into individual states, with each state concerned about its own functionality. With the decorator refactoring, all of the functionality relating to piece movements was moved into the piece classes. This also ensured cohesion, because each piece contained information of how they were able to move. The pattern refactorings also ensure extensibility because they are designed to be easily modifiable. The state pattern allows a new game mode to be easily added in at any time. In addition, the decorator pattern makes it easy to change each piece's movement, since they are now controlling themselves rather than being controlled by a giant blob class.

The metrics did show that coupling between objects increased in the refactored system. This is a tradeoff. Since more objects were introduced in order to adhere to separation of concerns, coupling inevitability increased due to the fact that more objects had to communicate with each other. The trade off was appropriate because it made the code more maintainable (less blob classes) and each class handles its own functionality, rather than multiple unrelated functionalities (such as the GUI handling malformed URL exceptions).

Refactoring identification	Checker Piece Decoration
Metric evidence	
Other evidence	The original checker classes were basically identical. They contained only getter and setter methods and only information on their color and type.
Description of the refactoring	This refactoring was heavily related to the Rules Simplification refactoring mentioned below. It's purpose was to move the responsibility for checking for possible movements to the piece classes. This would help the piece classes to serve a purpose and dramatically reduce the lines of code and responsibilities of the Rules class. The checker piece classes were worked into the decorator pattern because a king piece essentially does everything that a single piece does, but with some extended functionality.
Classes involved	Piece, SinglePiece, KingPiece, Rules

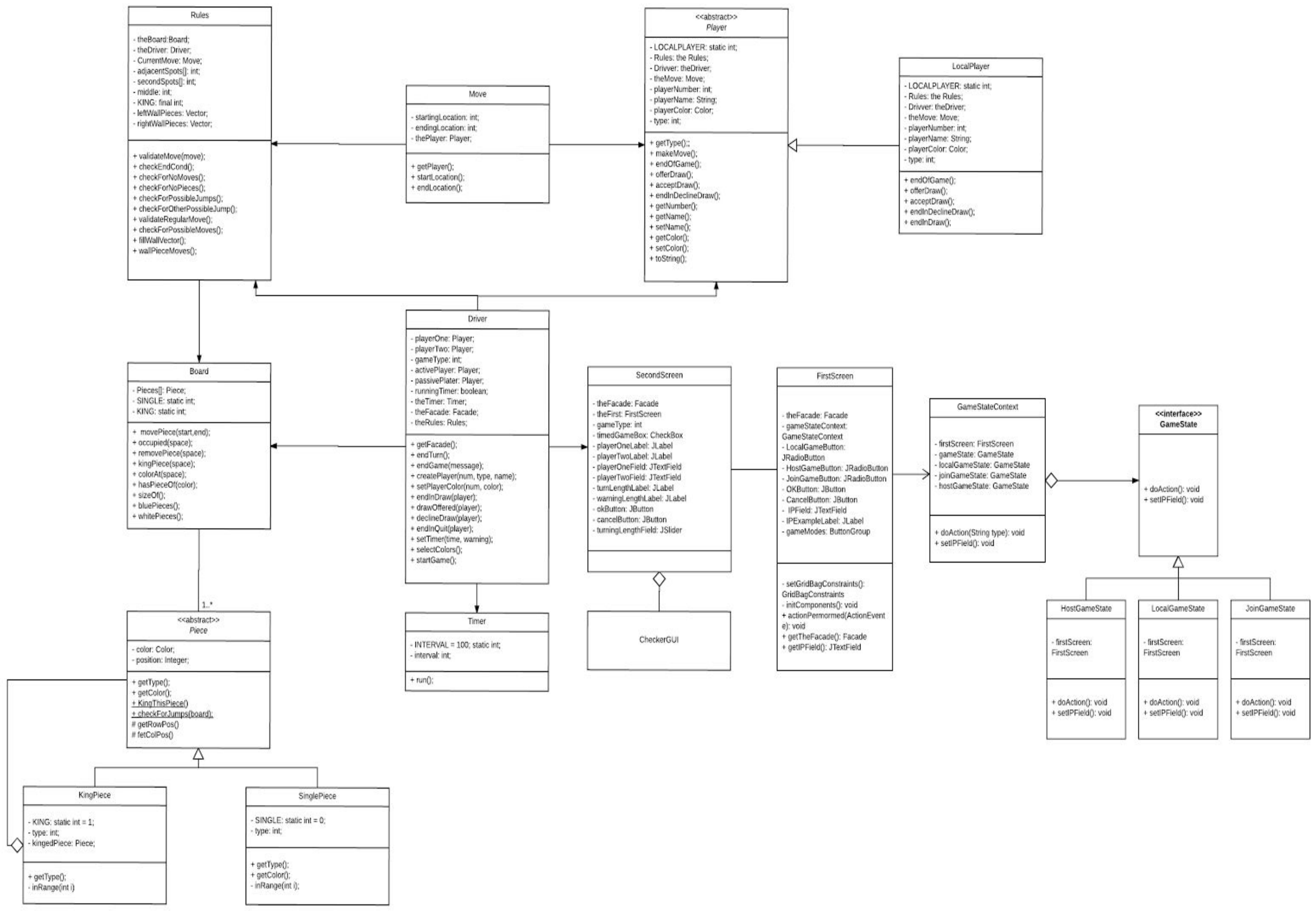
Refactoring identification	Rules Simplification
Metric evidence	Several of the methods for handling jumps had multiple hundreds of lines of code.
Other evidence	The jump checking methods were massive series of complex and difficult to understand if-else statements.
Standard refactoring pattern (if any)	Decompose Conditional, Extract Method
Description of the refactoring	The method used an unnecessary conditional with code that was repeated exactly with different loop starts, so the duplicated code was extracted to a different method and the loop start was used passed as a parameter into the method.
Classes involved	Rules

Refactoring identification	State Refactoring
Metric evidence	
Other evidence	Nested Try Catch
Standard refactoring pattern (if any)	Replace Conditional with Strategy (State in this instance) Extract Class
Description of the refactoring	This class uses a huge conditional to determine which game mode to start. In addition, it uses a large, generic try catch block to catch whatever non specific exception that might be thrown as a result of starting the game. This can all be replaced with a state pattern - each state is a type of game (ie. LocalGame, HostGame), with each state handling any functionality and exceptions relating to that game mode.
Classes involved	FirstScreen.java

Refactoring identification	GUI Refactoring
Metric evidence	Cyclomatic Complexity
Other evidence	Duplicate code, blob methods
Standard refactoring pattern (if any)	Extract Method Replace Exception with Test
Description of the refactoring	The initComponent() method in both Firstscreen and Secondscreen has a large amount of duplicate code that can be reduced with method extractions and by converting local variables into state. In addition, like Firstscreen, Secondscreen's event handler has a large, nested try catch block. In this instance, the block is used to catch issues with invalid timer values. This can be refactored by replacing the exceptions with test - instead of

	passing invalid values to the setTimer method (which throws the exception), just test to make sure the value is valid in the first place.
Classes involved	FirstScreen.java, SecondScreen.java

Figure # 5 Refactored Class Structure



Design Patterns

State Pattern- When the PlayCheckers class is started, a GUI launches that gives the user the option to pick between a local game, host game, and a network game. Depending on the game chosen, a large conditional handles the execution of each type of game. Because each game mode has its own exceptions, this conditional is wrapped by a try catch block that catches generic exceptions. This part of the GUI was refactored to use the state pattern. Each type of game mode was a state. Each state handles the functionality associated with the game mode, and handles its own specific exceptions related to that functionality.

Name: Game Mode		**GoF pattern: State**
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**
HostGameState	Concrete State	State that handles the functionality of a host game
JoinGameState	Concrete State	State that handles the functionality of a network game
LocalGameState	Concrete State	State that handles the functionality of a local game
GameStateContext	Context	The context that maintains the instance of three concrete states and defines the current state
GameState	State Interface	An interface which defines the methods required in each of the concrete state
FirstScreen	Client	The GUI window where the user selects which game mode they want to play in
Deviations from the standard pattern:		

Decorator Pattern - The checker Piece classes Single and King are each responsible for the same things, with king pieces simply adding further functionality to a single piece. Because of this, the KingPiece can act as a decorator for the SinglePiece when a SinglePiece is kinged and requires that added functionality. As a further benefit, the pieces can then be treated generically without a need for casting.

Name: Checker Pieces		**GoF pattern: Decorator**
Participants		
Class	**Role in GoF pattern**	**Participant's contribution in the context of the application**

Piece	Abstract Component	Abstract class for checker game pieces. Declares several abstract methods that handle things such as when a piece is kinged or checking for possible moves.
SinglePiece	Concrete Component	A standard checker piece. Creates a KingPiece to wrap itself when it is kinged. Finds possible moves in its forward facing direction, depending on color.
KingPiece	Concrete Decorator	When a SinglePiece is kinged it is wrapped in a KingPiece object. The check for jumps method of the kinged piece uses the wrapped single pieces methods to find the jumps in one direction and then adds functionality to find jumps in the other direction
Deviations from the standard pattern: The Component is an abstract class rather than an interface. There is no decorator interface.		

Sequence Diagrams

GUI Sequence Diagram - New

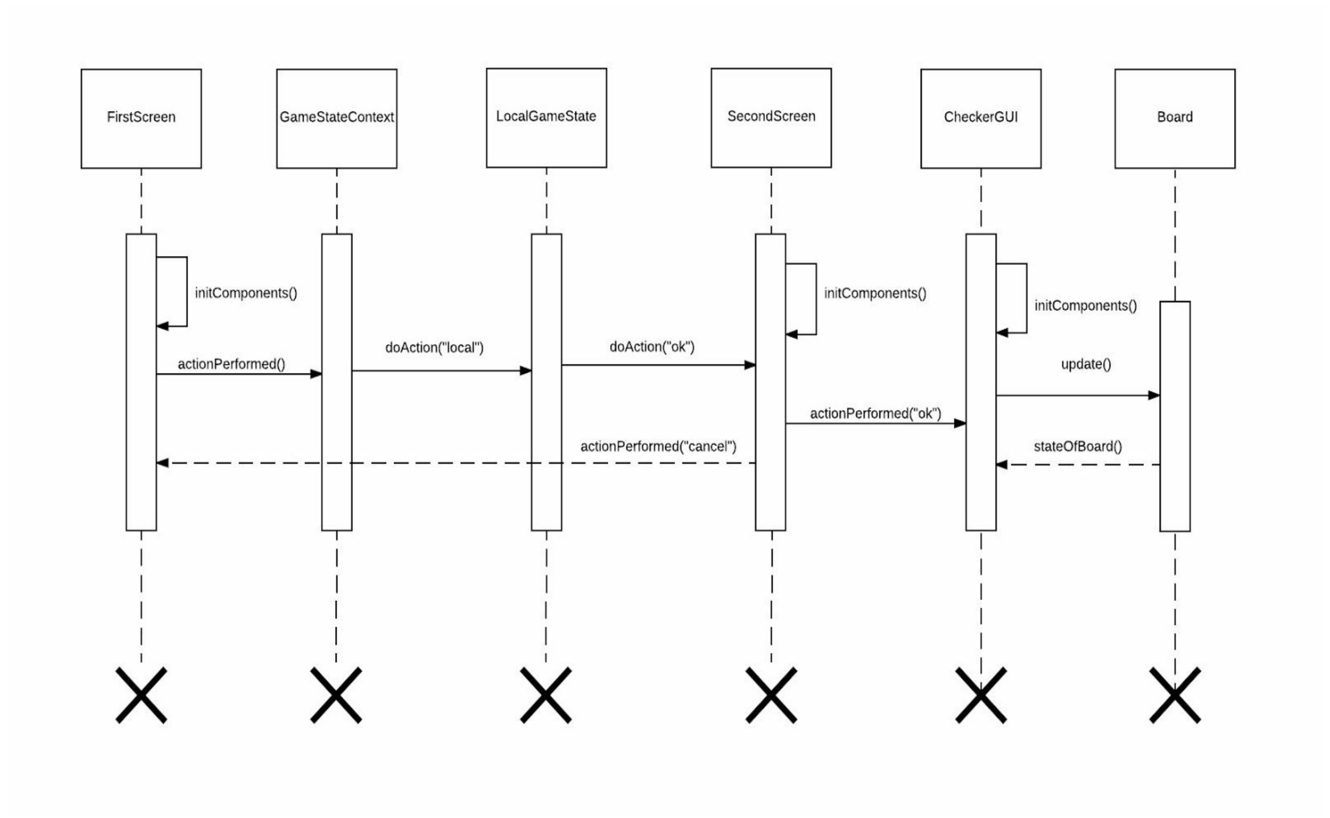


Figure #6 GUI Sequence Diagram (Updated)

King Piece Wall Move Sequence Diagram - New

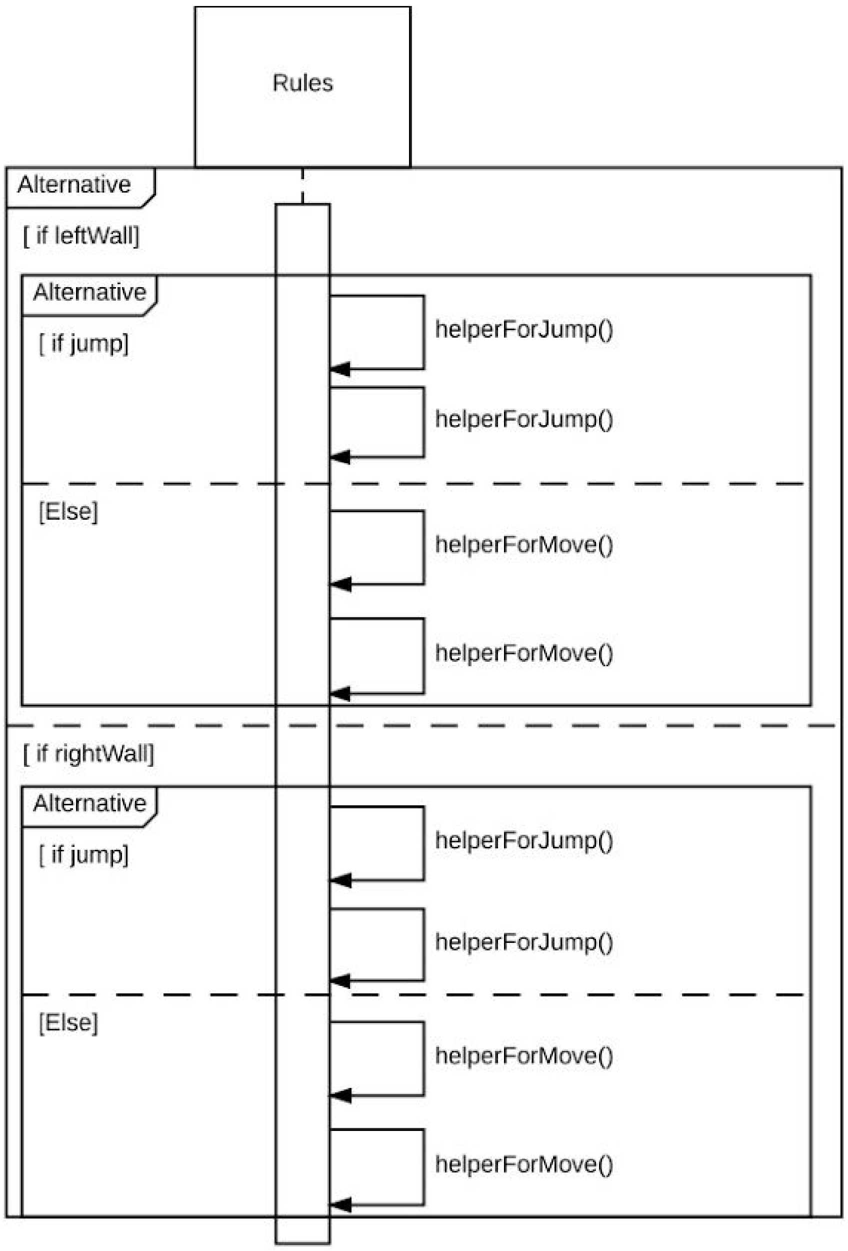


Figure #7 King Wall Jump Sequence Diagram (Updated)

Implementation

State Refactoring: In FirstScreen.java, the event handler method (actionListener()) has an enormous try catch block, inside of which is an enormous conditional. This block handles almost everything - it determines what kind of game is being played (local, network, host), it sets the game mode, it catches exceptions for any possible errors in these game modes, it starts the next GUI, etc. This enormous block is extremely unnecessary and cluttered.

Instead of having this one giant block, the state pattern is implemented. Each state has their own exceptions and starts the GUI in its own way. Since there are three possible games, each has its own state. Each state handles its own functionality and any exceptions that may arise. For example, a network game might throw a MalformedURLException, but a local game would not. Doing this makes the functionality of each game mode clearer, reduces the size of the event handler blob, and it is easier to make a change if a new game type is added in the future.

GUI Refactoring: In the CheckerGUI, instead of manually adding all 64 JButtons in the initComponent, we can reduce a huge amount of code by using a for loop and some helper method. The CheckGUI also had the checkEndConditions() method. We decided to move this method to the Rules to follow the Single Responsible Principle. The update() method of CheckGUI had several duplicated code and nested if else, so we created some helper methods(updatePiece, checkTypePieceToUpdate, and updateImageIcon) to remove those duplicated code in the update() method to make it is more readable, understandable, and cleaner.

In both FirstScreen.java and SecondScreen.java, there was a lot of duplicate code used in order to build the GUI. Changing each GUI component to private state allows every component to be accessible throughout the class. In addition, making the most common operations into methods allows us to call the methods over and over instead of duplicating the same code 10 times in one method.

Also, SecondScreen.java has a nested try catch block similar to FirstScreen.java, relating to invalid timer values. However, this nested try catch can be fixed easily by replacing the exception with a test, since its functionality is much smaller than FirstScreen.java's nested try catch. Instead of having nested blocks, we can simply test for valid timer values and send an error message in the GUI if the test fails.

Check For Jumps Simplification: In the Rules class, for the purposes of validating piece movement, there were several methods such as Validate Move, Check For Jumps, Check For Other Jumps, Check For Wall Moves, and so on. All of these classes were packed with complex, nested conditional statements with hard to follow logic and there were many instances of redundant and repeated code throughout. For this reason we decided to extract the repeated code out into methods and to simplify or eliminate the instances of conditional statements. As a result the Rules clast was greatly simplified, becoming more readable and much shorter.

Checker Piece Decoration: In conjunction with the Jump Simplification refactoring, we determined that some of the responsibilities for handling movement could be handled by the Piece classes. By doing this we would be able to further simplify the Rules class. In addition, we determined that the decorator pattern could be used because the behaviors of king pieces are mostly extensions of standard pieces. The decorator pattern also allowed for us to treat the Piece classes generically so that Rules was no longer coupled to the specific piece subclasses anymore.

Metric Analysis

What are the metrics for the refactored code base? This only has relevance for the areas of the refactored design that you implemented. How did your refactoring affect the metrics? Did your refactoring improve the metrics? In all areas? In some areas? What contributed to these results?

Weighted Method Complexity: Weighted method complexity (WMC) is a metric that shows the total cyclomatic complexity of each method in a class. The Rules class was refactored to remove the jump responsibilities to each individual piece. The pieces themselves now control their individual moves, rather than Rules. This reduced the weighted method complexity of the Rules class from 127 to 110, which is a 17% decrease. In addition, the cyclomatic complexity of Rules' checkForPossibleJumps method decreased from 27 to 3, an 89% decrease

Cyclomatic Complexity: As mentioned in the initial metric analysis, the Firstscreen and Secondscreen classes both had high complexity values - 11 and 13 respectively. In the Firstscreen class, the actionListener was refactored by replacing the conditional with a state pattern to handle the different game modes. This reduced cyclomatic complexity from 11 to 6 - a 45% decrease. This is because when a game mode is selected, only the functionality of that state matters, greatly reducing the number of independent paths.

In Secondscreen's actionListener, cyclomatic complexity was reduced by 54%, from 13 to 5. This refactoring involved replacing the nested exceptions with tests. The GUI now checks for invalid exceptions and stops execution, instead of passing on invalid values and catching exceptions. This reduces the number of paths possible, and as a result, reduces cyclomatic complexity.

Number of Transitive Dependents: A transitive dependency is a class which is relied on directly/indirectly by another class. In the refactored system, the average number of transitive dependent increased from 9.84 to 12.43. This is because in the original system, a lot of different functionality was contained within large classes. However, the refactored system divided responsibility to new classes more specific to their functionality. While dependency went up, it was simply a tradeoff to increase cohesiveness of the system.

Coupling Between Objects: Coupling between objects (CBO) is a metric which calculates the number of classes/interfaces a class is coupled with. In the original design, the CBO average was 4.74, but the new system's CBO value is 5.74. This was to be expected because the new design introduced new objects in order to delegate functionality. Like the dependency metric, this is a tradeoff where coupling was increased in order to increase cohesiveness.

Lines of Code: Some of the minor refactorings included removing duplicate code and promoting code reuse. For example, rather than making 64 buttons using duplicate code, a method was called over and over again. These refactorings lead to a sizable reduction in the number of lines of code - down from 5444 to 4466. All of the refactorings combined wiped out 978 lines of code (or 18%) from the system without changing any of the functionality. A large reduction indicates that complexity of the code is decreasing while readability increases.

Reflection

After being supplied with the original source code, our initial process of examination was for each member of our team to read through the files. We also used a plug in for our development environments to show us various metrics and statistical information about the files. This visual inspection helped us to familiarize ourselves with the structure of the project and how the various files interacted with each other. The metric analysis plug in was very helpful for this process because it provided a quick and easy visual reference for parts of the code that were hotspots with a high likelihood of poor design. It was a useful tool that helped to provide direction beyond simply blindly reading through the code and searching for visual evidence. After having gone through the refactoring process, our team's general opinion is that it can be a difficult and tedious process. Despite this, it is exceptionally rewarding in the long run, by improving the design and readability of the code it helps to ensure that less drastic and time consuming refactorings will need to be made later on.