# DigiCheckers

# By

# Victorious Bovine

**Team Members:**

Abdulaziz Almerdasi

Josh Berk

Austin Cowan

Hussain Mahkareem

Jon O'Brien

Submission Date : May 8th 2015

# Table Of Contents

| | |
|---|---|
| **Overview** | |
| **Original Design Overview** | |
| **Refactored Design** | |
| **UML Class Diagrams** | |
| **Interaction Diagrams** | |
| **Class-Responsibilities-Collaborators (CRC)** | |
| **Metrics Analysis** | |
| **Reflection** | |
| **Aspect of our Refactoring** | |

## 1. Overview:

DigiCheckers is a checkers Java Applet that allows you to play checkers on your computer with friends locally or over the internet. For each game, players can select a name and two game modes, timed and normal. For timed games, players are given a time-limit per turn and an overall time-limit for the game. For LAN games, the user must select either to host a game or to join one at a specified server IP address. DigiCheckers uses a built-in rules system to keep track of player's moves and make sure that they are all legal. DigiCheckers is refactored version of another product, Checkers.

## 2. Original Design Overview:

### Implementation Weaknesses:

When our team began working on refactoring the original design, the most noticeable design flaws were apparent upon running the program. These included a bug where upon starting a game of checkers, all of the pieces were hidden. Fixing this problem only required changing a few lines of code referencing the image resources for the Graphics User Interface.

Other weaknesses included weak implementations of inheritance, specifically for the Piece classes and the Player classes. For instance, Piece, King Piece, and SinglePiece, did not share attributes describing their type. Instead, KingPiece and SinglePiece had their own individual attributes to describe their type and the abstract class Piece only described their color.

Probably the most notable weakness is the bloated Rules class. The rules class is supposed to function as a constraint on the kinds of moves players can make during gameplay. However, because of the data structure used for the board, a one-dimensional array, and the fact that all of the rule constraints are implemented in a single class, the Rules class is very large and hard to break down for potential future refactorings.

**Implementation Strengths:**

The first thing you'll notice when using the original design of Checkers is that the interface is rather intuitive. The steps used to set up the game are straightforward and easy to understand. There is a logical sequence of steps, starting by selecting how the players will interact to play the game, whether by local play on one machine or over the internet, and moving to setup the actual game with player names and time constraints. Beyond the initial setup, playing checkers is also very easy. You simply click the space containing the piece you would like to move and then click the space where you would like to place it.

In regards to the internal structure, there are clear attempts to use inheritance to allow for modular changes to the software, for instance using different piece classes. However, these piece classes, KingPiece and SinglePiece do not share much functionality and the code is at times redundant.

**Design Documentation:**

- Weaknesses:
    1) The requirement file doesn't have a description for the requirements, such as visualization constraints, interaction  constraints.
    2) The documents don't have any interaction diagrams, such as sequence and state diagrams.
    3) Class description in the analysis class diagram file is really similar to the code's comments.
- Strengths:
    1) The use cases in the requirement file are really strong. They are organized and detailed very well.
    2) The UML class diagram is clear and organized very well.
    3) Strong description for the non-functional requirements
    4) based on strategy implementation, can expand checkers game into chess

**<u>Use of design Patterns (original design):</u>**

- **Facade - Manages the game (partial implementation)**
- **Proxy - network player**

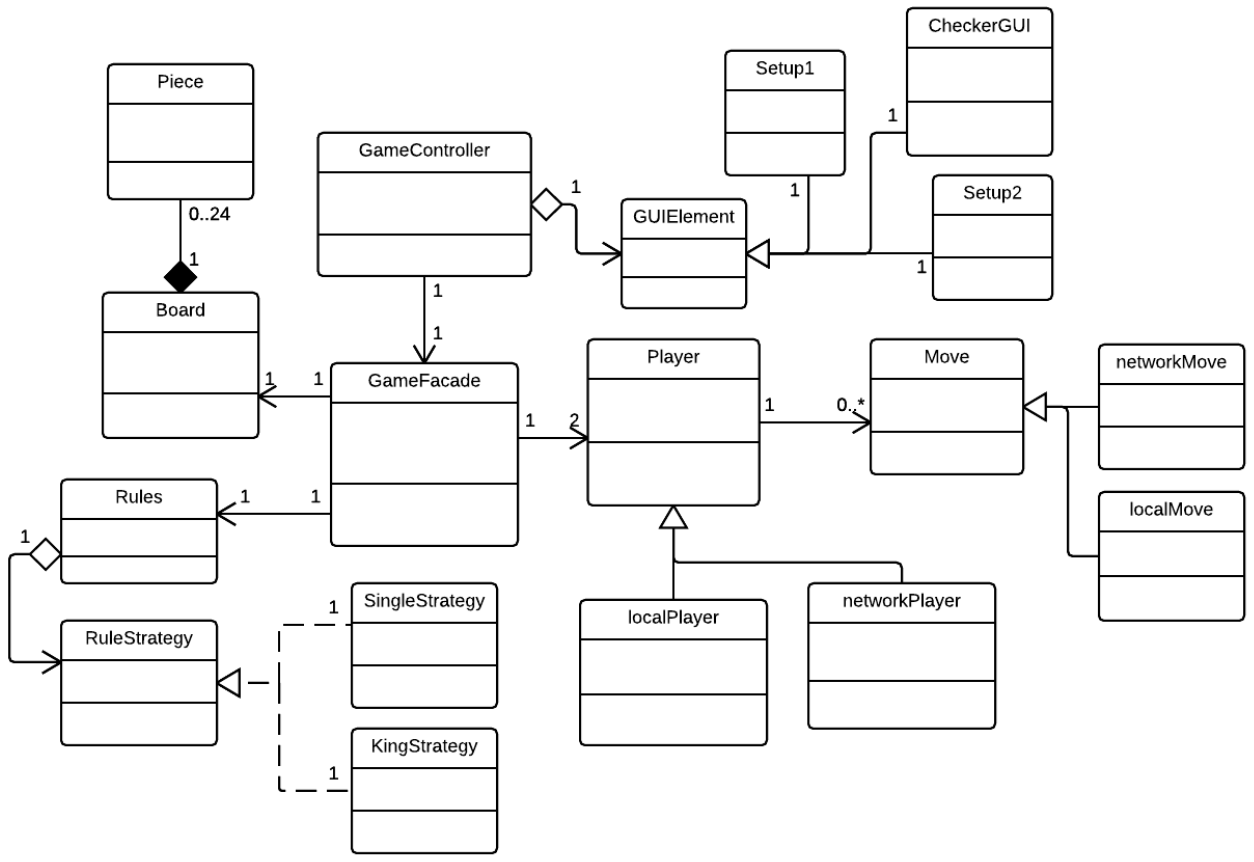
## 3. Refactored Design

The most vital change that was made to the design of the game was a re-implementation of the facade used in the game. Many of the other classes had to be de-coupled from each other in order to finalize a proper implementation of the façade. In the new design, the game controller has access to the façade while the façade provides a higher-level implementation of operations such as telling the GUI to tell the next player to make a move and retrieving, validating, and executing that move in the board. Additionally, the game can now update the GUI board using the Board object state.

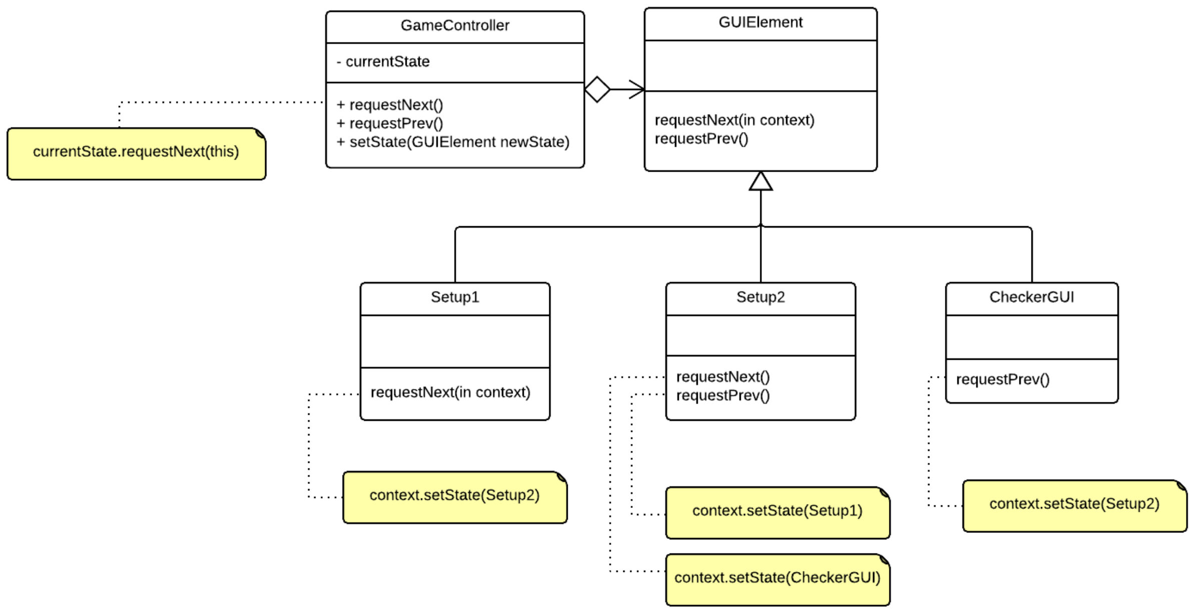
**<u>Use of design Patterns:</u>**

- **Facade**: manages game subsystems, move making/validation
- **Observer**: observer for board changes and implementation of board pieces
  - uses **strategy** to check piece placement
- **Strategy**: rules, makes things simpler to compute, knowing which move was made could lead to future move calculations being simpler
- **State**: Handles gui events to allow for restarting the game instead of just ending the program at the completion of the game

## 4. UML Class Diagram

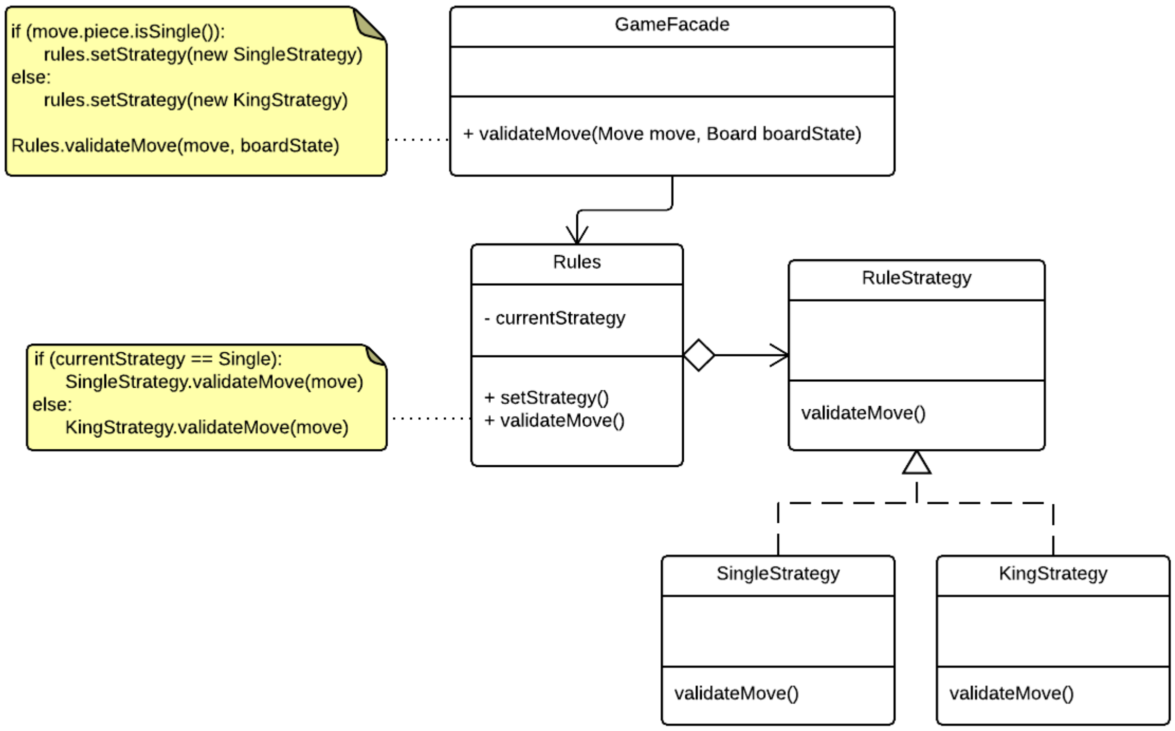


State Pattern Implementation:



For the graphics user interface, our group decided it would be best to use a state pattern. This works well in this context because users will navigate each of the screens in a very predictable and precise manner which can easily be specified with this pattern. Above, the diagram shows the classes involved. Here, the GameController class is the context, keeping track of which state the GUI is in. The GUIElement class is a superclass of all of the GUI states. The controller has a currentState attribute that stores an instance of the current GUI state being used. The controller can change its state by calling the respective requestNext() or requestPrev() method. Each of these methods calls the repective method on the currentState and the currentState invokes the proper state change on the controller. The possible state changes are also shown in the diagram.

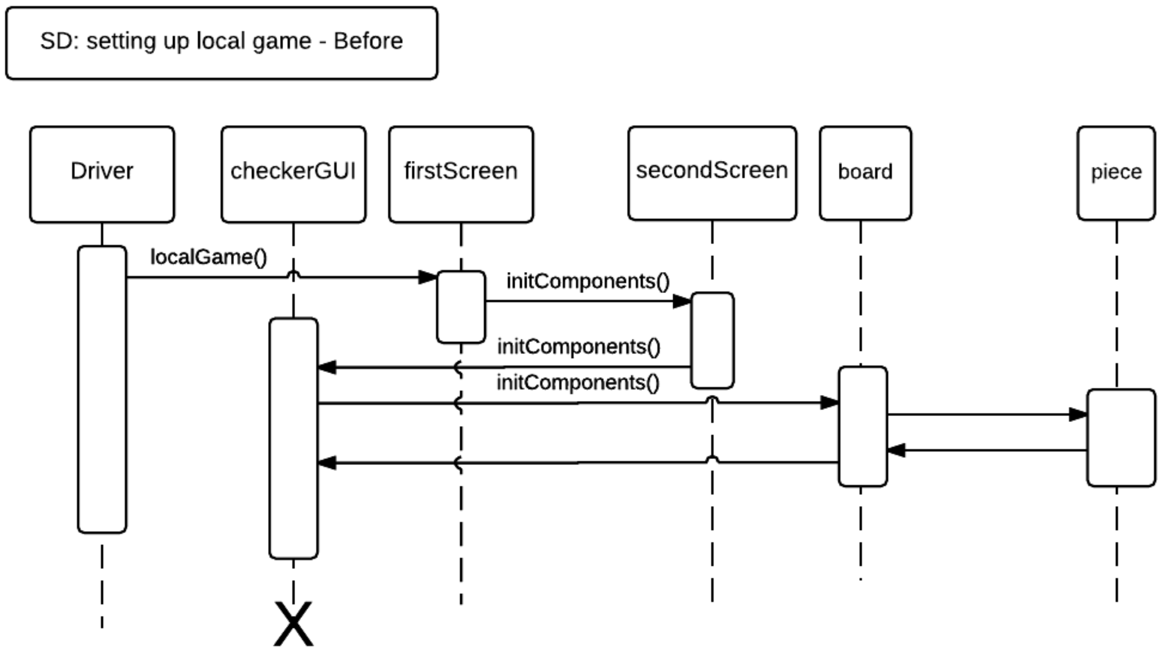
## Strategy Pattern Implementation


if (move.piece.isSingle()):
rules.setStrategy(new SingleStrategy)
else:
rules.setStrategy(new KingStrategy)
Rules.validateMove(move, boardState)
GameFacade
+ validateMove(Move move, Board boardState)
Rules
- currentStrategy
+ setStrategy()
+ validateMove()
if (currentStrategy == Single):
SingleStrategy.validateMove(move)
else:
KingStrategy.validateMove(move)
RuleStrategy
validateMove()
SingleStrategy
validateMove()
KingStrategy
validateMove()

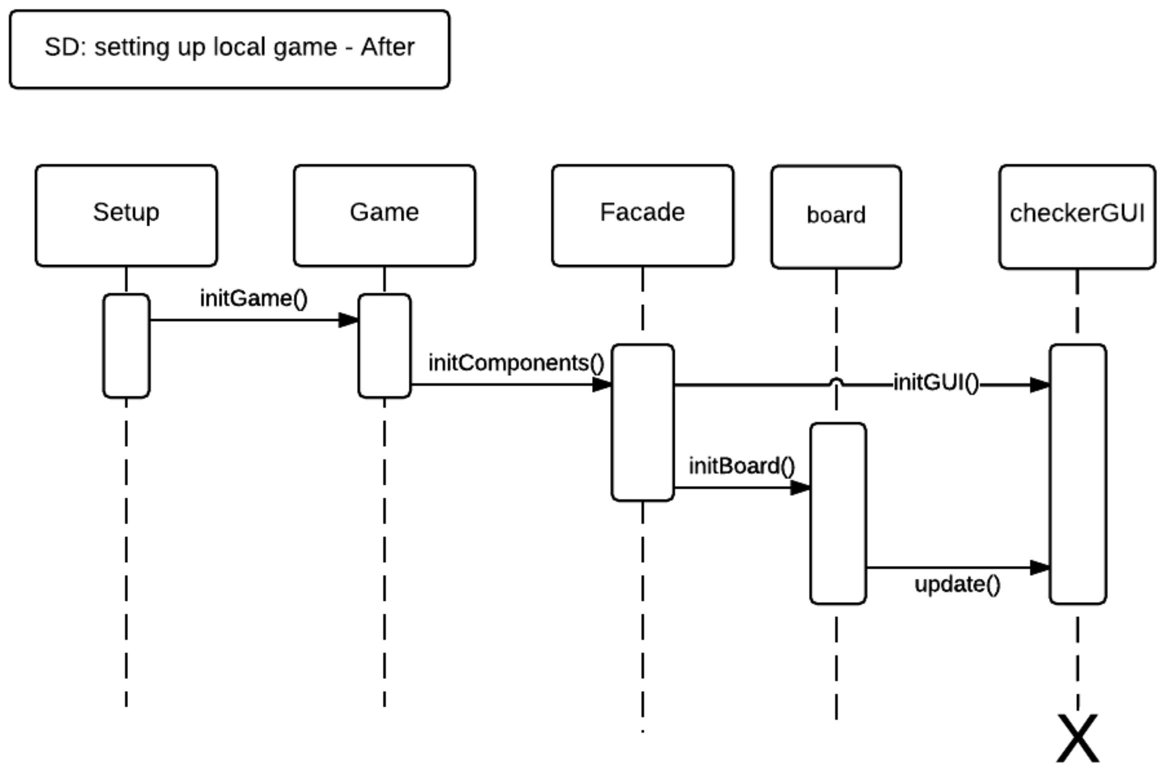
For the strategy pattern, the GameFacade represents the context, the Rules class represents the client, and the RuleStrategy class is a interface which is implemented by all of the rule strategy classes. These strategies  are separated out in this way to simplify the algorithms used to validate a given move. For instance, in the case of checkers, single pieces have a more strict regimen because they can only move toward the opposing side, thus requiring a more strict ruleset for move validation. However, a King piece does not require this check since it can move toward either side of the board.

In order to choose a strategy, the move is checked by the GameFacade once it receives the move from a player to see which type of piece the move is being executed upon. From here, it chooses which strategy to use for validation and sets that strategy in the Rules class. Then the validation is invoked upon the Rules class's currentStrategy instance which subsequently returns a boolean value representing the move's validity.

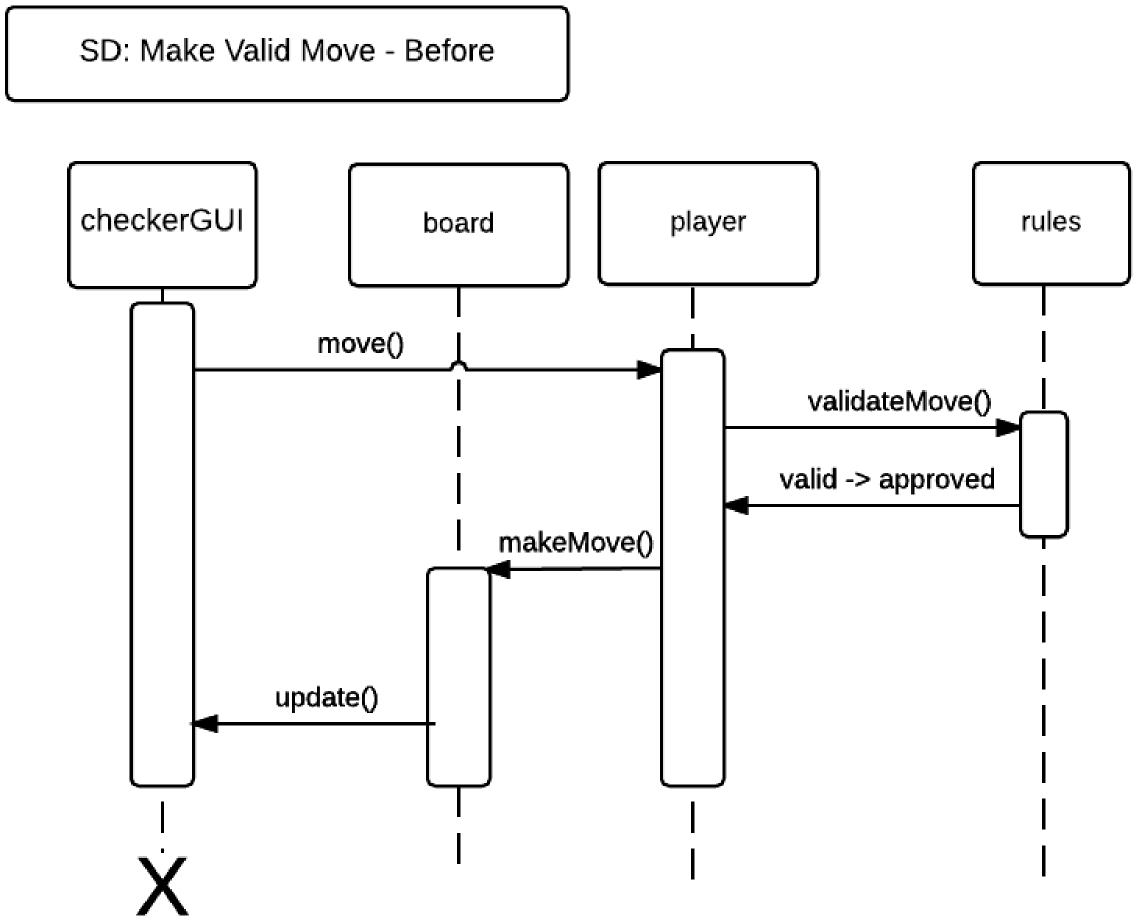
## 5. Sequence Diagrams



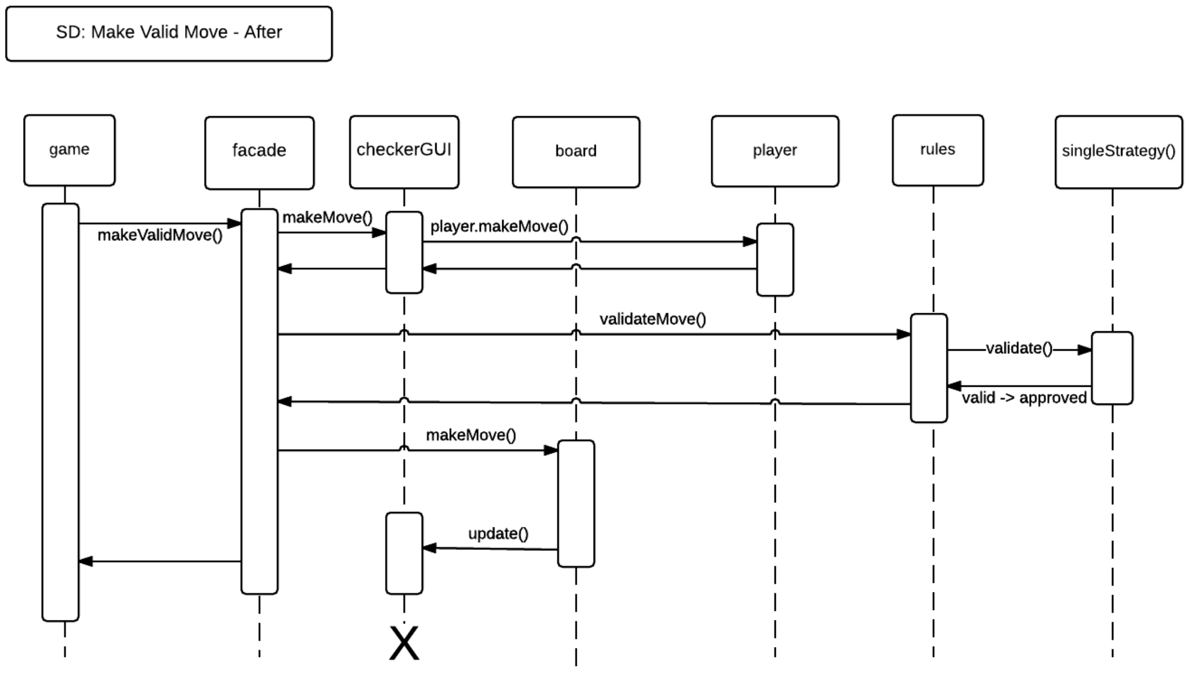
Setting up a local game under the original design consisted of a call from the driver to the checker gui class, from the first and second screens. InitComponents was called on them and the gui was populated from the board, which was filled with 64 pieces from the piece class.



Setting up a local game after the refactoring, using the new design, consisted of a call from the setup class, which was responsible for the settings for the game, the player names ,and turn length, game information, etc, and a call to the game class, which calls initComponents on the facade, which handles the game management. The facade calls initGUI on the CheckerGUI which is then populated by the board of pieces, and its other buttons required for complete game functionality.



Making a valid move under the original design consisted of calling the player to make a move from the CheckerGUI and going through the board to the rules class, validating the move as valid or invalid, alerting the player of any invalid moves, then making that move on the board itself, then updating the gui with the respective changes.



Making a valid move under the refactored design consisted of the calls from the game class, which drives the game and directs the facade to manage the game, to the checkerGUI to makeMove() which queries the current player to make a move, once the player makes a move, the move is validated in the rules class and the facade manages the updating of the gui based on the move being valid or not, if the move is invalid, the player is queried for another move. The completion of the valid move is when the game driving class is returned to for the next operation in the running of the refactored implementation.

## 6. Class-Responsibilities-Collaborators (CRC) Refactored

| | | |
|---|---|---|
| Class | checkerGUI | |
| Responsibilities | It’s the actual board which contains a lot of action listeners on it. | |
| Collaborators | Uses | Used By |
| | Board | Game |

| | | |
|---|---|---|
| Class | Piece | |
| Responsibilities | This is an abstract class representing any piece that knows about it's color and possible moves and captures. There are subclasses KingPiece and Single Piece. | |
| Collaborators | Uses | Used By |
| | | Board |

| | | |
|---|---|---|
| Class | Rules | |
| Responsibilities | This class is used to check the validity of the moves made by the players.  It also checks to see if the conditions for the end of the game have been met. | |
| Collaborators | Uses | Used By |
| | | Facade |

## 7. Metrics Analysis

- **Original Code Base Metrics:**

| Metric | Value | Hot spot area in code base | Potential refactoring |
|---|---|---|---|
| Cyclomatic Complexity | 48 | Rules.java.wallPieceMoves | Re-implement rules with more iteration and simpler methods. |
| Cyclomatic Complexity | 41 | CheckerGUI.java.actionPerformed | Break up the method and reimplement them. |
| Cyclomatic Complexity | 18 | TestingKernel.java.testInvalidMoves | Break up the method and reimplement them. |
| Cyclomatic Complexity | 13 | Secondscreen.java.actionPerformed | Break up the method and reimplement them. |
| Cyclomatic Complexity | 11 | Firstscreen.java.actionPerformed | Break up the method and reimplement them. |
| Nested Block Depth | 8 | Rules.java.checkForOtherPossibleJump | Remove limitations on possible moves. Use different data structure to represent board and simplify search for movies. |
| Nested Block Depth | 6 | Secondscreen.java.actionPerformed | Break up the method and reimplement them. |
| Nested Block Depth | 6 | CheckerGUI.java.update | Break up the method and reimplement them. |
| Method lines of Code | 300 | Rules.java.checkforposisiblemoves | The method should be broken up for readability and maintainability. |
| Method lines of Code | 160 | Secondscreen.java.wellPeiceMoves | The method should be broken up for readability and maintainability. |
| Method lines of Code | 139 | CheckerGUI.java.initComponents | The method should be reduced and redundant sections be restructured. |
| Method lines of Code | 104 | TestingKernel.java.wellPeiceMoves | The method should be broken up for readability and maintainability. |
| Method lines of Code | 90 | Firstscreen.java.initComponents | The method should be broken up for readability and maintainability. |
| Method lines of Code | 58 | NetworkPlayer.java.processCommand | The method should be broken up for readability and maintainability. |

- **New design metrics (based on refactored project):**

| Metric | Value | Hot spot area in code base |
|---|---|---|
| Cyclomatic Complexity | 15 | CheckerGUI.java.update |
| Nested Block Depth | 6 | CheckerGUI.java.update |
| Method lines of Code | 122 | CheckerGUI.java.initComponents |

- **What did these initial measurements tell you about the system?**

The initial system architecture was very much a collection of classes with a semi-thought out design, but not much actual implementation of patterns or thoughts about coupling/cohesion. The system was functional in most of the given requirements, but some of the design choices and where certain methods were seemed to be not very well thought out.

- **How did you use these measurements to guide your refactoring?**

The team used the original metrics to determine which areas were most in need of refactoring or reimplementation. Given some classes with long sections of hard-coded methods, the team decided it would be a good idea to simplify the code and reduce the number of method lines of code in some files, such as CheckerGUI. Some of the metrics also pointed to files that were very complex and the team decided on which to alter large blocks of logic to determine rules, or determine which not to refactor, given some had high coupling to many different areas of the code base and would take much to long to alter, as combining some classes would touch a majority of the code base, see piece, etc.

- **What are the metrics for the refactored code base? This only has relevance for the areas of the refactored design that you implemented.**

(see above table, with CheckerGUI being the main file that was reduced by the team, others have small changes and refactorings done, CheckerGUI.initComponents() was the largest metric change from the original to refactored design seen.

- **How did your refactoring affect the metrics? Did your refactoring improve the metrics? In all areas? In some areas What contributed to these results?**

The main changes were to CheckerGUI.initComponents(). This method had hard-coded sections for the board class and the pieces were individually placed on the board in the gui with x and y positions. This was easy to change, with the addition of a simple for loop, the code was reduced by over half for the class. The team collaborated on a method of adding the color to the piece to denote which was a proper place to move a piece, as at first that was the hardest part of the refactoring of the class. There was a positive impact on the metrics to the application, as the number of method lines of code was reduced and some coupling was reduced given teh combination of some of the classes in the new design.

## 8. Reflection

The team began the refactoring process by looking at the old documents provided by the previous team. The team determined visually what sections looked to be most in need of a new approach and decided to alter a few sections of the original design to lead to a better design after completion.  The team next ran metrics on the original design and determined which coded sections of the original design were most in need of complete/partial re-implementations. The CheckerGUI class was one such class.

The CheckerGUI was filled with hard-coded sections to accommodate the board of hardcoded buttons, 64 of them, for the game board being used throughout the application, plus a few buttons in the CheckerGUI for various other game options. The Board class was filled with references to data structures that were generic, along with various other locations in the application; these were typechecked and the added security helps the overall program as a whole. The team decided on new features to add to the project, decided on a timer to be implemented as most of the code seemed to be in place, as well as a feature that would automatically restart the game at its' completion.

The main feature added was that of the automatic game restart, whenever the game ended originally the application would close, now the new implementation of the application restarts the game on the settings screen upon completion of the game via any of the win/ending conditions. The team had planned on implementing a timer feature, but the original code base and highly coupled design prevented this feature from being implemented.

## 9. Aspect of our Refactoring

- **Automatic Restart Functionality -** At the conclusion of the game (draw, resign, quit, win) the game will reset and allow the user to either change settings, or cancel out of the application.
    - Facade
    - Driver
    - CheckerGUI
    - LocalPlayer
    - Secondscreen
    - PlayCheckers