Drewhannay/Chess - Milestone 3

Tina Ali - 6299008
Kevin Silva - 6324703
Arash Zare - 1942190
Gaspard Diallo - 9088016
Oleksandr Dymov - 5987555

SOEN 343
Software Architecture
Professor: Peter C. Rigby
Due: Wed, October 30th

Summary of Project

Project Description

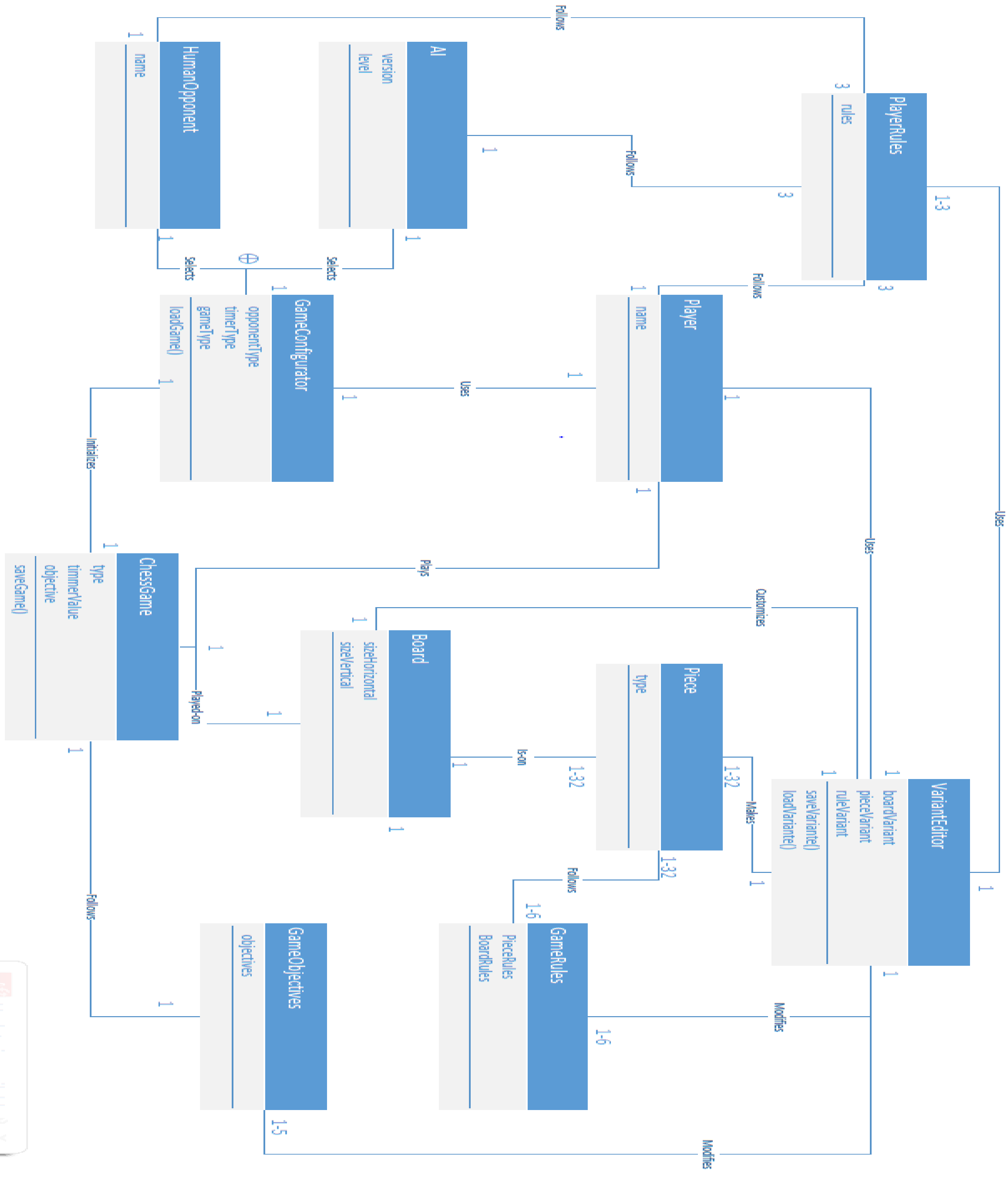
This project is concerned with the development of a Java-Based Chess game that enables its users to play against other users, or a smart AI player on the same computer. Moreover, the player is given the choice of playing either a regular chess game with its conventional rules, or a variant of the game. The game variant can be created by customizing a wide range of rules and game parameters, and represents this application's main use case. As we have a team of passionate chess players, the project will allow us to study the mechanics of the game, as well as improve our knowledge on proper design patterns and refactoring in medium-sized projects.

The project is at a stable maturity level, since it is possible to compile the program, play the game and easily interact with variant customizations, without any alarming or noticeable bugs. The project was started about two years ago, but it was last updated in the current week by the author. This last update was concerned with some programming tuning, which suggests that the project is still active and of interest to be creator. We hope to provide an improvement through the refactoring process, which the creator will want to accept.

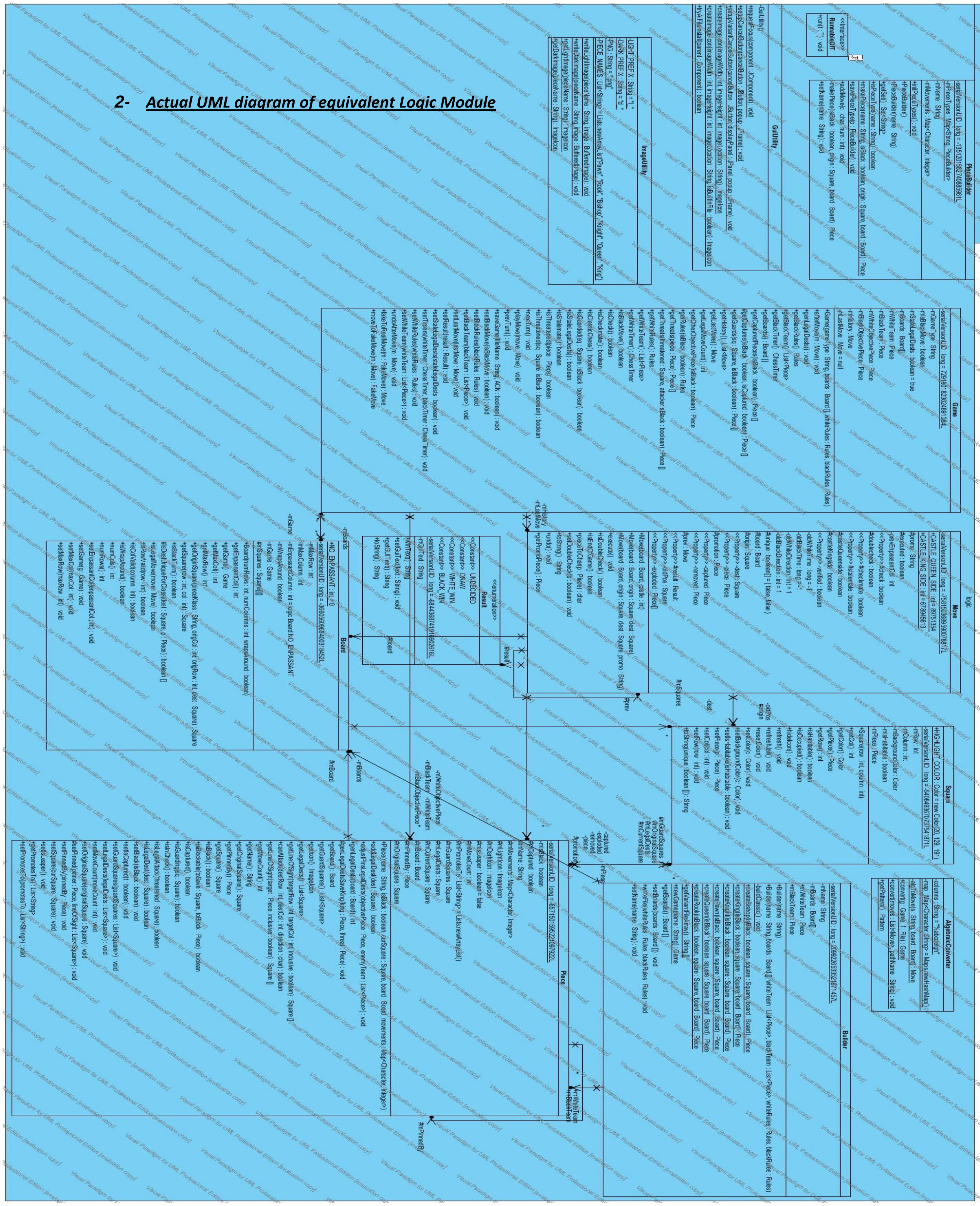
Currently, there is a total of 9 contributors listed to be working on this project, with 4 of them providing at least 25 source code commits. This project is coded in Java, and we will be using Eclipse to refactor, compile and run the project. Since it falls under the gaming category, the program cannot be considered a dangerous or harmful application for its users. Nonetheless, good design and architecture are key factors when determining the functionality and enjoyment of any game. Games are a part of any healthy lifestyle and as developers who may end up working in gaming development, we believe it is a necessity to understand the complexities of a gaming architecture on a simpler level, before coding with the standards of today's industry. This project allows us to do just that.

Class Diagrams of the Actual system

1- *Conceptual UML Diagram of D/Chess (from previous milestone)*



2- *Actual UML diagram of equivalent Logic Module*



3- ***<u>Diagram analysis and Comparison</u>***

Although hard to read because of non-removable watermarks, the actual class diagram provided by the reverse engineering tool consists of a selection of relevant classes and the interactions between them. The diagram itself appears very convoluted, but it is simply due to the fact that the system implementation suffers from notorious code smells that are described in further detail later in the report. Nonetheless, we have attempted to omit some filler elements by showing the classes which are of interest in the purpose of this refactoring process. Additionally, many of these classes, such as *PieceBuilder* Class and *Builder* Class for example, refer to the customizable nature of the game, a key feature of the application.

Just by looking at both the conception and actual diagram versions of the system, it is clear that the team's implementation decisions with respect to design differ from those chosen by the author of the program. Clearly, the actual diagram includes a lot more details which the conceptual one does not. However, this diagram also shows that the number of dependencies among classes is very high, therefore indicating that there is probability a high degree of coupling among objects.

By analyzing the class diagram of the implemented system, it is very apparent that the *Game* class is considered to be beyond "fat", and can be characterized as an extreme case of feature envy, seeing as it has over 40 public methods. This architecture shows a clear lack of cohesion, as this one class assumes far too many responsibilities. In fact, many different aspects of the game are encapsulated within this class, making it an extremely hard to maintain piece of code which will need to be refactored. Although this refactoring might require a lot of work, the program will benefit greatly from a leaner architecture which provide more modularity. For example, in our conceptual version of the diagram, this class is represented by the following classes: *GameGenerator, GameRules* and *GameObjectives.* Similarly, the *Move* class has dozens of public properties and methods and is shared by all kinds of chess pieces, which we assumed to be a feature which could be taken care of using an interface. This way, by using polymorphism, we determined that the smartest way of achieving this was to provide an interface, which decides the implementation of how each piece is supposed to move, depending on its type.

Nonetheless, there are multiple classes from the conceptual class diagram which map to those of the actual class diagram. For example, both *Board* and *Piece* classes are present in both versions of the diagrams, and by looking at their methods and attributes, it is clear that they assume similar responsibilities. However, the conceptual class diagram focuses a lot more on the customization aspect of the game, while in the actual class diagram is geared towards providing the main logic of the game. Additionally, in the conceptual classes, we deal with the *AI* aspect of the game by encapsulating it all within its own class which will take care of the algorithms, whereas the actual diagram does not take this into consideration, as it represents a module of its own. Finally, another noticeable difference between diagrams is the fact that the actual implementation considers a *Square* class, in order to represent a physical square on the chess board, which would mean that each and every square is a separate object which are then used by the Board object using its own data structure. This is an interesting idea because it allows individual squares to adopt special behaviors according to its assigned properties; this can make the game much more customizable.

Reverse engineering tools:

We used the **Virtual Paradigm** for UML modelling Tool to help us generate our UML class diagram straight from the java code. It does not show a perfect UML, with all the necessary relationships but it does allow us to see the classes, their attributes and their methods. By putting that into a visual, we were able to better determine the classes needing the most refactoring.

Two class' linking methods and attribute **declarations:**

Board Class:

1. private Game mGame;

2.
```
public Square getOriginSquare(String pieceKlass, int origCol, int origRow, Square dest)
  {
        // This method should never be called for anything but Classic chess
        if (!getGame().isClassicChess())
                return null;
… }
```

3.
```
List<Piece> movingTeam = (getGame().isBlackMove()) ? getGame().getBlackTeam() : getGame().getWhiteTeam();
```

Game Class:

1.
```
public Game(String gameType, Board[] boards, Rules whiteRules, Rules blackRules)
  {
        mGameType = gameType;
        mBoards = boards;

        …



        for (Board b : mBoards)
                b.setGame(this);
        …
}
```

2.
```
public boolean isClassicChess()
  {
        return mGameType.equals("Classic");
  }
```

3. public boolean isBlackMove()

```
  {
              return mIsBlackMove;
  }
```

3. public List<Piece> getBlackTeam()

```
  {
              return mBlackTeam;
  }
```

3. public List<Piece> getWhiteTeam()

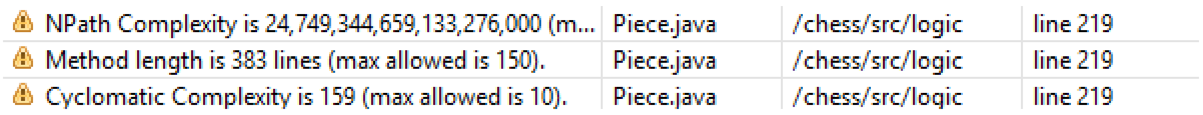
```
  {
              return mWhiteTeam;
}
```

Code Smell and Possible Refactoring

In order to initiate and simplify the code smell detection process, the “**Checkstyle**” project was used to explore possible areas of improvement within the game. It is a useful source code analyzer which can be used in the form of an Eclipse plug-in. This plug-in enabled to detect more than 6000 problems, according to the plug-in’s predefined coding standards. Four of these problems were selected for the initial refactoring plan, one of which was described using additional UML diagrams, and was selected for analysis with respect to its class, method, and attribute declarations directly from the source-code.

The four Refactorings:

The first three detected code smells are found in the **Piece** class at the genLegalDests(Board board) method:



⚠ NPath Complexity is 24,749,344,659,133,276,000 (m...	Piece.java	/chess/src/logic	line 219
⚠ Method length is 383 lines (max allowed is 150).	Piece.java	/chess/src/logic	line 219
⚠ Cyclomatic Complexity is 159 (max allowed is 10).	Piece.java	/chess/src/logic	line 219

Refactoring 1: The first objective of this refactoring would be to reduce the high NPath Complexity number of 24,749,344,659,133,276,000, given that the standard threshold allowed is 200.

This problem will be first approached by reducing as much as possible the number of possible independent paths, or branches. This can be achieved by reducing the number of conditional statements visible within the source code. For example, we can replace the various conditional logics like “**if (mMovements.containsKey('N'))** {...}”, by strategies which can assign things in a dynamic manner, such as polymorphic behaviours.

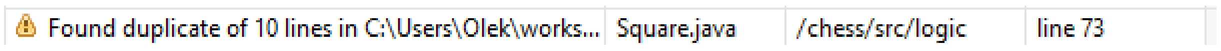
Refactoring 2: The second objective for this class method would be to deal with the Excessive Method Length of 383 lines (max allowed of 150 lines). This is normally attributed to a lack of cohesion in the class’ design. In other words, this class wants to accomplish too many things and suffers from feature envy.

The refactoring of this method can be achieved by reducing the number of method tasks. In fact, a great contributor of the excessive length is the repetition of code which occurs by the lack of smart conditional logic. Therefore, by creating and implementing helper methods that could to decompose these conditionals by a smart design using interfaces, it will reduce the method complexity and efficiency with respect to reusability. Removing the accumulation to the visitor **Board** class might also contribute to reducing the number of lines of code.

Refactoring 3: Then the last objective for this class is to decrease the Cyclomatic Complexity evaluated at 159 (max allowed is 10).

To do this, the main refactoring task will be to reduce the number of if statements and implementing a more object oriented approach to accomplish this. It would be necessary to create an interface which could define legal behaviors depending on types of Pieces being used. Such an interface could impose a method for this, such as "**addLegalDest()"**, which could then generate the proper behavior using polymorphism.

Refactoring 4: The fourth detected code smell of "code duplication" found in the Square class for the methods refresh() and **refreshJail()**:

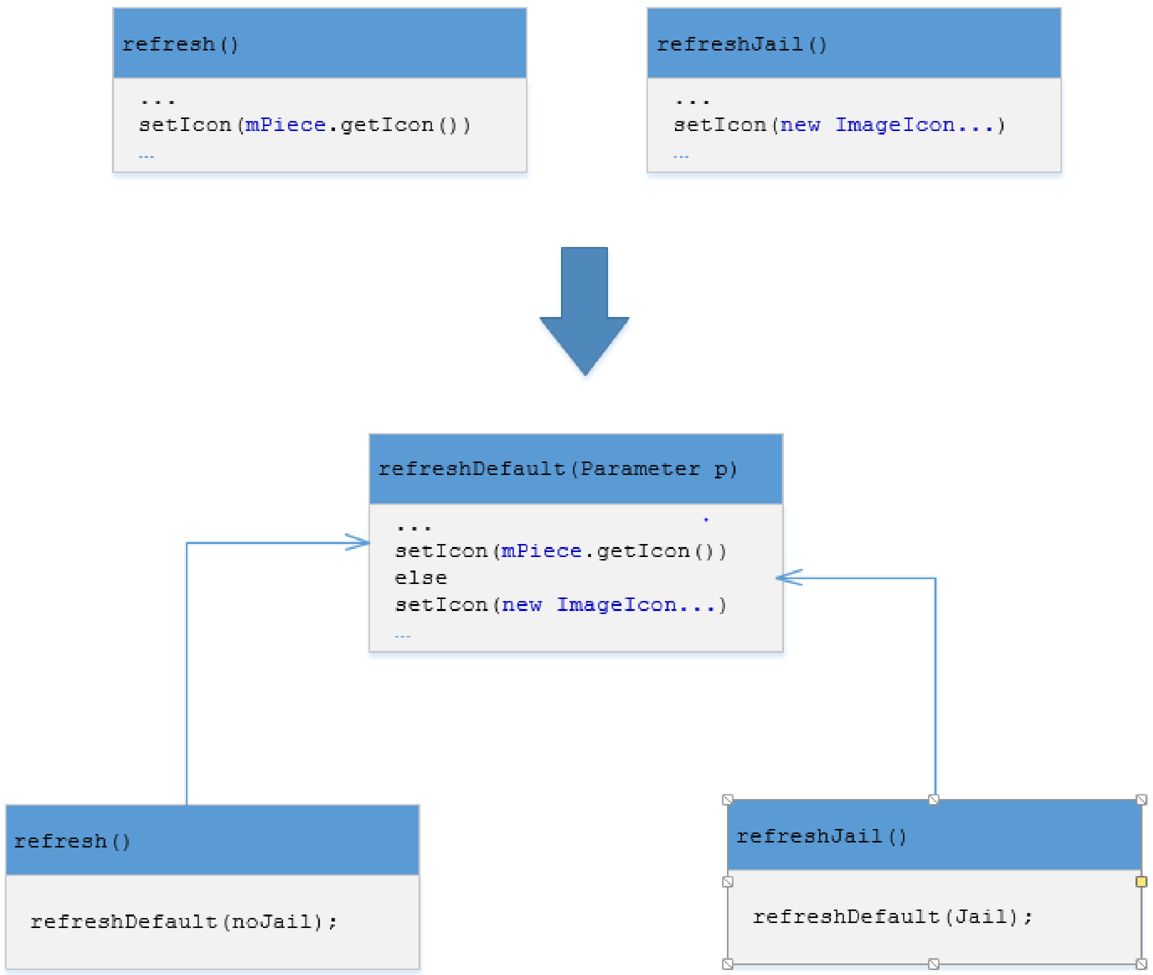


Found duplicate of 10 lines in C:\Users\Olek\works...	Square.java	/chess/src/logic	line 73

The fourth refactoring objective is to simplify this class by replacing the code duplication using either the **Form template method** or **Extract method**. Due to the lack of superclass, the second refactoring is preferable as it would use a passed parameter to decide on one single non-repeated line execution and therefore will be replacing the 10 other duplicated code lines in each method.

- This could be accomplished by creating a method called refreshDefault(parameter p) and the end execution of the different else statement "**setIcon**" will depend on the passed parameter.

*UML Diagram of the Extract Method refactoring of the detected duplicated methods in the **Square** class:*



refresh()
...
setIcon(mPiece.getIcon())
...
refreshJail()
...
setIcon(new ImageIcon...)
...
refreshDefault(Parameter p)
...
setIcon(mPiece.getIcon())
else
setIcon(new ImageIcon...)
...
refresh()
refreshDefault(noJail);
refreshJail()
refreshDefault(Jail);

*Code provided for the 4th Refactoring: Duplicate methods detected in the **Square** class:*

```
/**
* Refresh the GUI's view of this Square with the current accurate
* information.
*/
public void refresh()
{
		setOpaque(true);
		// if there's a Piece here...
		if (mPiece != null)
		{
				// ...and it has no Icon
				if (mPiece.getIcon() == null)
						setText(mPiece.getName());// Use it's name
				else
						setIcon(mPiece.getIcon());// Otherwise, use it's Icon
		}
		else
		{// If there's no Piece, clear the Icon and Text of the Square.
			setIcon(null);
			setText("");
		}
		resetColor();// Then reset the color too.
}

/**
* Refresh the GUI's view of this Square with the current accurate
* information, but only for Jails
*/
public void refreshJail()
{
		setOpaque(true);
		// if there's a Piece here...
		if (mPiece != null)
		{
			// ...and it has no Icon
			if (mPiece.getIcon() == null)
				setText(mPiece.getName());// Use it's name
			else
				setIcon(new   ImageIcon(mPiece.getIcon().getImage().getScaledInstance(25,   25,   Image.SCALE_SMOOTH)));//
Otherwise, use it's Icon
		}
		else
		{// If there's no Piece, clear the Icon and Text of the Square.
			setIcon(null);
			setText("");
		}
		resetColor();//             Then             reset             the             color             too.
}
```

As we can see from the code above, both method implementations do basically the same thing. These methods are used depending on a conditional statement elsewhere in the program. We can improve the behavior of the program by extracting these into one method that can dynamically chose the correct implementation, thereby reducing the maintaining efforts, and increasing reusability.