Chess'N'Chat

Team members:

Adigun Jide Idris 7128525

Chenglong Zhang 6842666

Kun Wang 6713084

Pavithra Keshavamurthy 6827527

Sun liu 6758878

The refactoring we suggested in Milestone3 are:

* Duplicated Code
* God Class
* Long Method
* Type Checking

**Pull Request**: https://github.com/edijman/SOEN/pulls?direction=desc&page=1&sort=created&state=closed

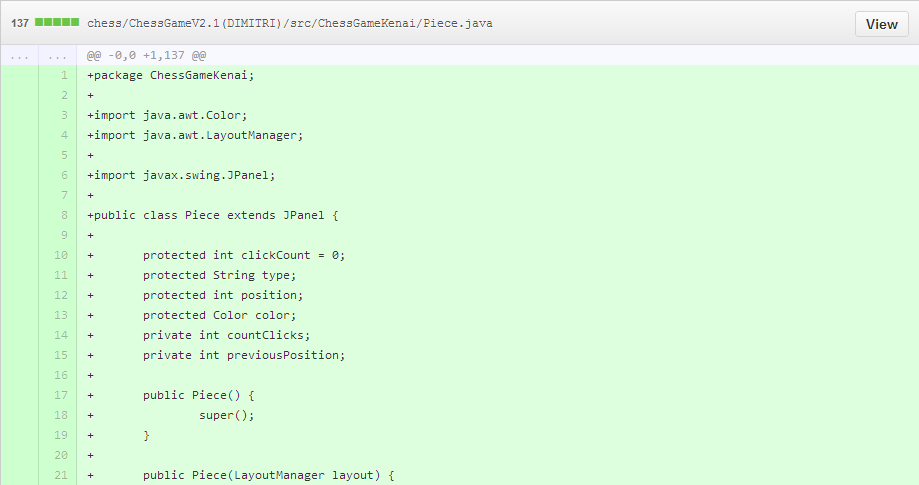
We have emailed the developer for his email address for sending the patch sets. Until now we are waiting for his reply.

**Change (0/3): Create a super class from VisualPiece and NonVisualPiece**:

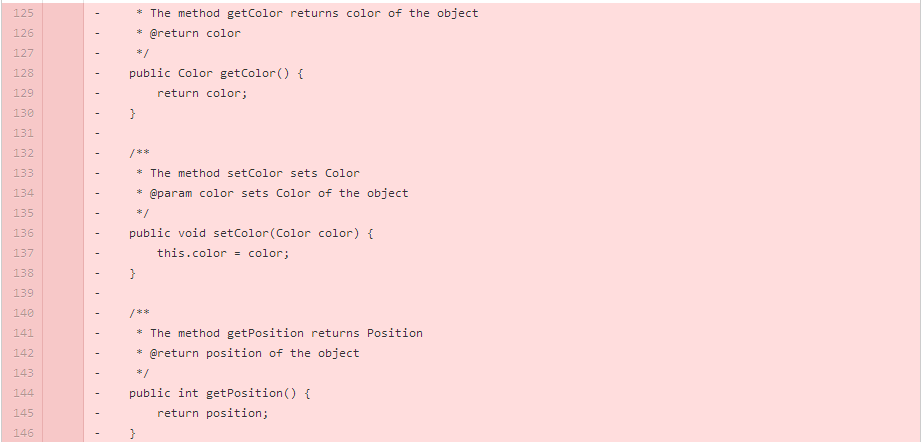
The original class Visual Piece and NonVisualPiece have a lot of attributes and functionality in common, that does exact same thing. This depict that there are unnecessary duplicated code when there should be none, this would create complication when trying to make changes to the system.

Moreover, based on Object Oriented Programming, we would expect a class to represent a chess piece object and should be intuitively name ‘Piece’ and any type of chess piece (Visual Piece and Non-visual Piece) should inherent attributes and functionalities from this class, thus making modifiability easy if it had to be made to all type of chess piece.

Hence, we created a super class with the common attributes of class VisualPiece and NonVisualPiece, making both classes (VisualPiece and NonVisualPiece) a subclass of class ‘Piece’ and deleting common and duplicated attributes and methods from class VisualPiece and NonVisualPiece.



Created a new class (Piece.java) containing attributes and methods that are common to classes VisualPiece and NonVisualPiece



Deleted common attributes and functionality that are common to both class (VisualPiece and NonVisualPiece).





Then, we make both classes (VisualPiece and NonVisualPiece) a subclass of class ‘Piece’





Updated reference to Piece class on classes that uses visualPiece and NonVisualPiece. Classes such as Chess\_data.java and CapturedPieces.java

1. **(Change 1/3)** : **Extract unrelated method to new created class**

The class ConnectionBridge.java has many unrelated methods and fields. This presents the class to have more than one responsibility other than connection process. When the class is doing more than what is supposed to do, it is God Class. Thus, I use the Extract Class Refactoring to extract and isolate unrelated methods and attributes from the large class.

Extract class creates a new class and move methods from the original class which follows the object oriented design principle “Single Responsibility Principle”. Create an instance of the new class in original class ConnectionBridge.java. Update references to ConnectionBridge.java where these methods are called. Move unrelated methods setGuestName() and setPlayerIconPath() into the new class.

Creation of an instance of the new Class created using Extract Class and updating the references in original class.





Unrelated methods will be removed from original class

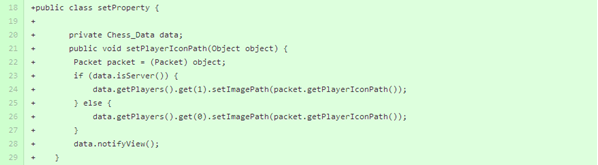


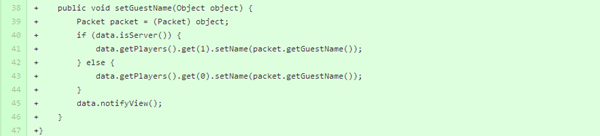






Methods moved in new Class



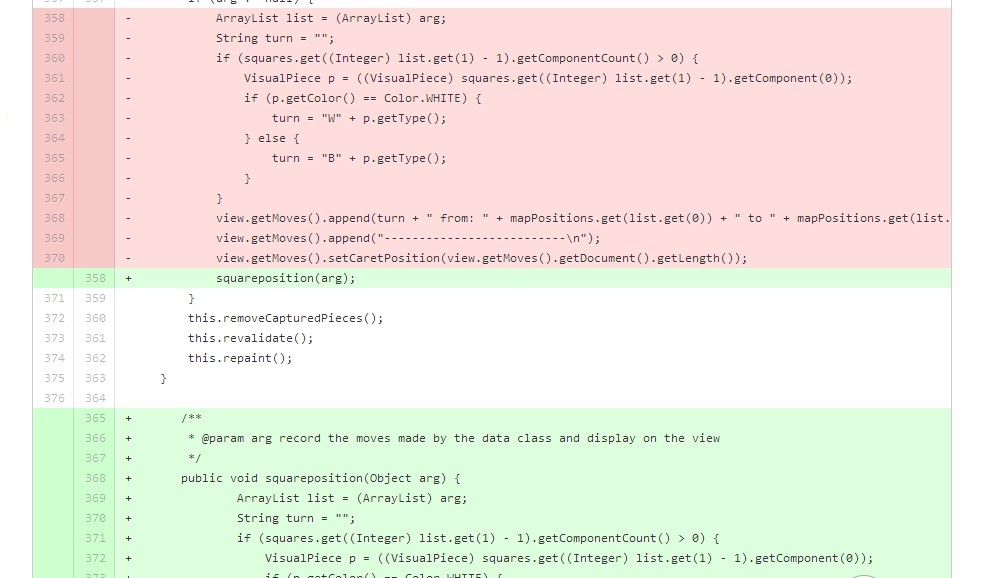


1. **Change 2/3: Extract method and delete unrelated code**

In the Board class the update method (related to the observer pattern) has too many things to do with the ChessData observable methods and contains too many attributes and parameters in it. So I create a new method named squarePosition to interacts with these instances’ method, hide the detail to another method

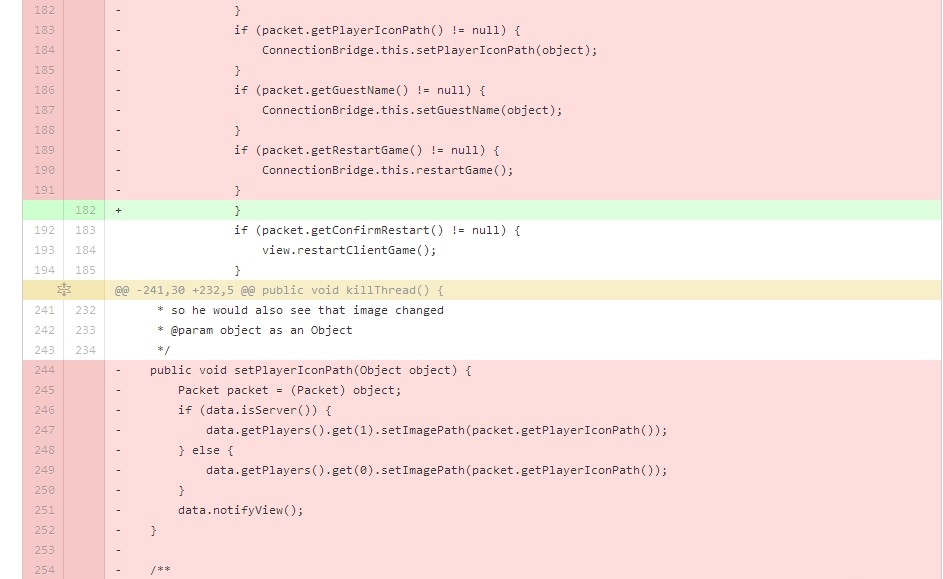
Deleted the unrelated code in ConnectionBridge class which has been proven fail to fulfill the original function desired by the developers. So I deleted the code due to the efficiency issue.

Chess/ChessGameV2.1(DIMITRI)/src/ChessGameKenai/Board.java

Extract the method in Board class from the conditional statement to the new method named squareposition. hod only pass the parameter to the new method. 

Chess/ChessGameV2.1(DIMITRI)/src/ChessGameKenai/ConnectionBridge.java

Deleted the unrelated code in the ConnectionBridge Class



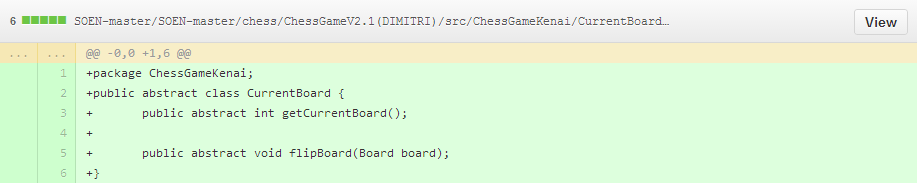
1. **Change 3/3: Replace Conditional with Polymorphism**

This type of code smell happens when to choose a variation of an algorithm (eg, NormalBoard or FlippedBoard) that should be executed, depending on the value of attribute (currentBoard). Mainly, it manifests itself as complicated conditional statements (switch statement) that make the code very difficult to understand and maintain.

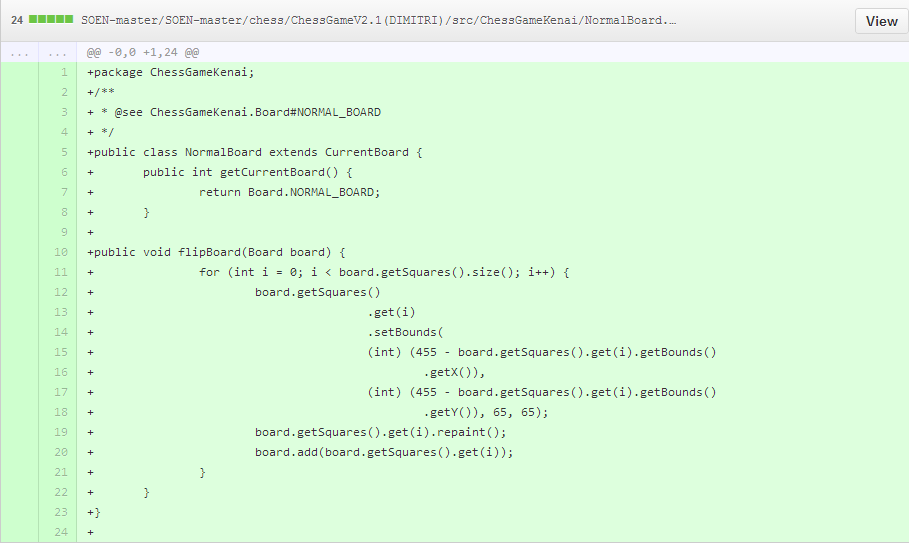
In general, I make the attribute as an abstract class, and then create sub classes class which include the variation of algorithm Inherit that abstract class.

The diffs are showed below:

Create new a class named “CurrentBoard.java”



Create a new class named”NormalBoard.java”



Create a new class named”FlippedBoard.java”;



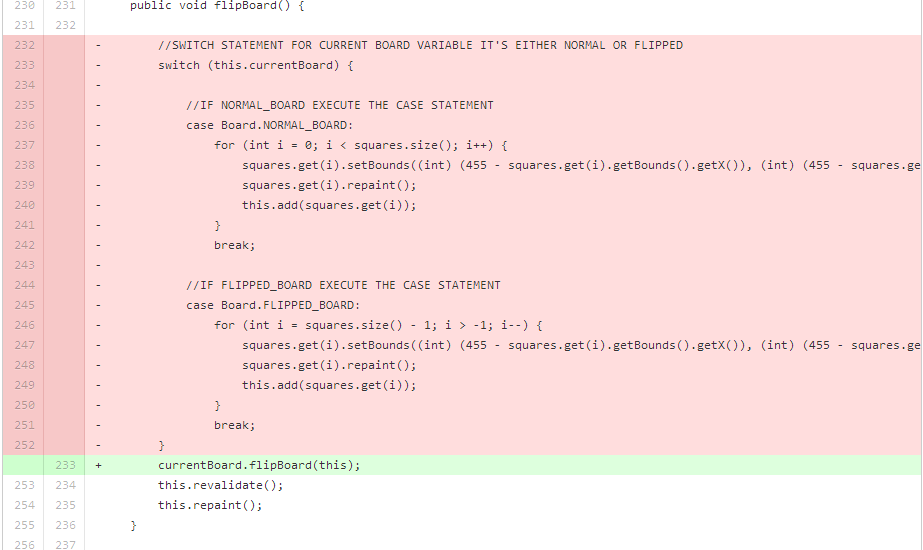
Modify the class “board.java”: for “currentBoard” Replace primitive type with State type:



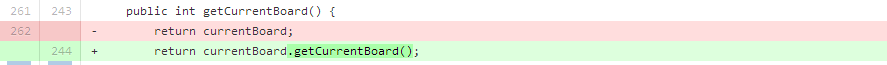
For “setBoard(int)”Replace field assignment with invocation of setter method



For flipBoard() Replace conditional structure with polymorphic method invocation



For getCurrentBoard() Modify getter method for the field holding the current state



For setCurrentBoard(int) Modify setter method for the field holding the current state

