Project Name

Team members:

Adigun Jide Idris 7128525

Chenglong Zhang 6842666

Kun Wang 6713084

Pavithra Keshavamurthy 6827527

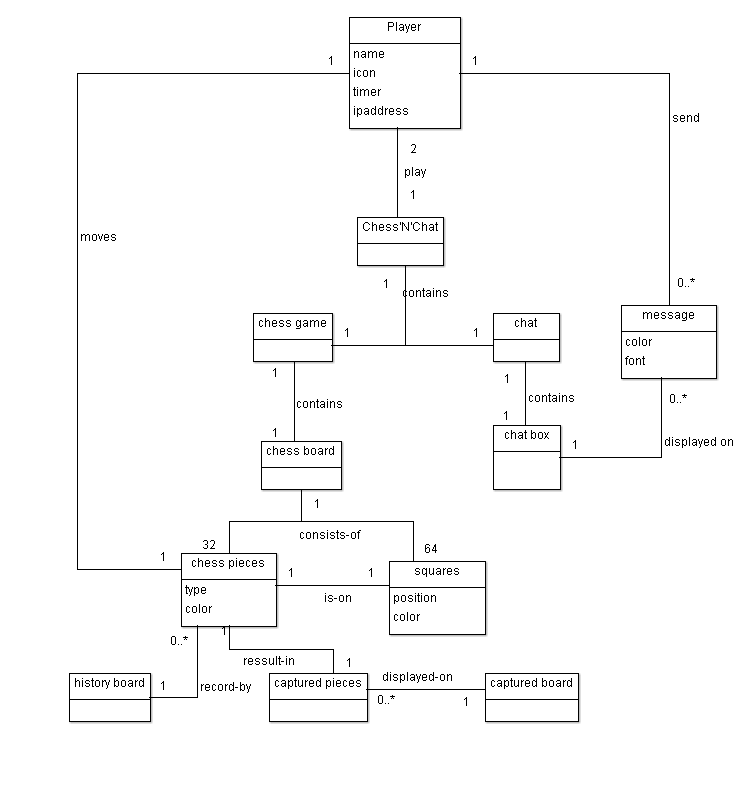
Sun liu 6758878

Due: March 2nd, 11 marks

# Summary of Project

Chess ‘N’ Chat project is an open source project, a game which is written in java and it implements multiple functionalities of a basic chess online game, it can be compiled and tested on Eclipse. It is a stable project. The current release of the project is in 2011. There were two developers working on this project. Chess game is responsible for creation of graphical entities like icons, characters, squares, board etc. and involves the population of the characters/pieces on the chess board. Players can set their names and their icons to their accounts. This project contains two players as primary actors. The system could be divided into two main subsets: chess and chat. The chess part contains different functionalities for users. For example, help and record of the time, save, load, record moves, and record captured pieces, etc. The chat part successfully fulfills the connection requirement during the chess game. The combination of Chess’N’Chat application will make the system more interactive and challenging.

# Class Diagram of Actual System



**Figure 1: Domain diagram of the system.**

Class **Square**

The attributes of class **Square**:

**private** **int** position;

**private** Color currentColor;

**private** **static** Chess\_Data *data*;

**private** Color previousColor;

OverLoaded Constructor for creating square objects:

**public** Square(Color color, **int** position) {

**this**.currentColor = color;

**this**.position = position;

**this**.setBackground(color);

**this**.setLayout(**null**);

**this**.addMouseListener(**new** SendData(*data*));

}

The Methods of class **Square:**

**public** Color getColor() {**return** currentColor;}

**public** **void** setColor(Color color) {**this**.currentColor = color;}

**public** **int** getPosition() {**return** position; }

**public** **void** setPreviousColor(Color previousColor) {**this**.previousColor = previousColor;}

**public** Color getPreviousColor() {**return** previousColor;}

Class **Board**:

The attributes of class board:

**public** **static** **final** **int** *NORMAL\_BOARD* = 2;

**private** **int** currentBoard = *NORMAL\_BOARD*;

**private** **boolean** isWhite = **true**;

**private** ArrayList<Square> squares;

**private** HashMap<String, String> imageMap = **new** HashMap<String, String>();

**private** Chess\_Data data;

**private** ArrayList<VisualPiece> pieces = **new** ArrayList<VisualPiece>();

**private** **boolean** isFirstTime = **true**;

**private** ChessBoardView view;

**private** HashMap<Integer, String> mapPositions = **new** HashMap<Integer, String>();

**private** ArrayList<String> positions = **new** ArrayList<String>();

The methods of class **Board:**

**public** ArrayList<Square> getSquares() {

**return** squares;}

The method getSquares returns the squares to the caller

return squares as an ArrayList.

**public** **void** setSquares() {

**int** y = 0;

**int** p = 0;

//CONSTRUCT SQUARE OBJECTS

**for** (**int** i = 0; i < 64; i++) {

p = i;

**if** ((i + 1) > 8 && (i + 1) % 8 != 0) {

p = ((i + 1) % 8) - 1;

}

**if** ((i + 1) % 8 == 0) {

p = 7;

}

**if** ((i) % 8 == 0) {

p = 0;

}

**if** (isWhite) {

squares.add(**new** Square(Color.*WHITE*, (i + 1)));

squares.get(i).se0000tBackground(Color.*WHITE*);

squares.get(i).setBounds(p \* (65), y, 65, 65);

squares.get(i).repaint();

isWhite = !isWhite;

} **else** {

squares.add(**new** Square(Color.*BLACK*, (i + 1)));

squares.get(i).setBackground(Color.*BLACK*);

squares.get(i).setBounds(p \* (65), y, 65, 65);

squares.get(i).repaint();

isWhite = !isWhite;

}

**if** ((i + 1) % 8 == 0) {

isWhite = !isWhite;

y += 65;

}

**this**.add(squares.get(i));

}

**for** (**int** i = 0; i < pieces.size(); i++) {

squares.get(pieces.get(i).getPosition() - 1).add(pieces.get(i));

}

}

**The board class creats the object of square class**

private ArrayList<Square> squares;

The board uses method setSquares to set the squares and creates the squares and adds them to the board within its own method.

squares.get(i).se0000tBackground(Color.WHITE);

squares.get(i).setBackground(Color.BLACK);

**1. Introduction**

We use Enterprise Architect - Sparx Systems 8.0 as a reverse engineering tool to generate the class diagrams based on the source code. (The original class diagram can be viewed in the attachment files) The class diagram is very big and hard to follow, so we decided to divide it in to two branches of interests, and only keep the key attributes and methods of the classes to make it more readable and understandable.

**2. Chat System (sub-system of Chess’N’Chat)**



**Figure 2: Class diagram of chat system**

According to the actual class diagram generated from tool ‘Enterprise Architect - Sparx Systems 8.0’, five classes (Chat.java, Chess\_Data.java, ConnectionBridge.java, ChessBoardView.java and Emoticons.java) are involved in the Chat System.

**2.1 Purpose of Actual Class**

* **Chess\_Data.java Class**: This class is data class that provides related data to most of the classes to the Chess’N’Chat system. This by itself does not obey the Object oriented design principles.
* **Chat.java:** This class focuses on GUI interface for the chat application. However, it has some functionalities such as saveChat () and sendMsg() which gives the class additional responsibility.
* **ConnectionBridge.java**: This class is responsible for establishing connection between client and server. However, the class has no link with the player.java class is not instinctual, because we expect player to be involve in a chat connection.
* **ChessBoardView.java Class:** This is also GUI class which is responsible for creation of chess board interface. But, it has functionalities like flipClientBoard, startTimer, stopTimer, loadCapturedPieces, getMoves whose responsiblility belongs to another class.
* **Emoticons.java:** This class contains all the emoticons required for chat application.
* **Player.java**: contains information about a player object but has no connection with any other class in the Chess’N’Chat system.

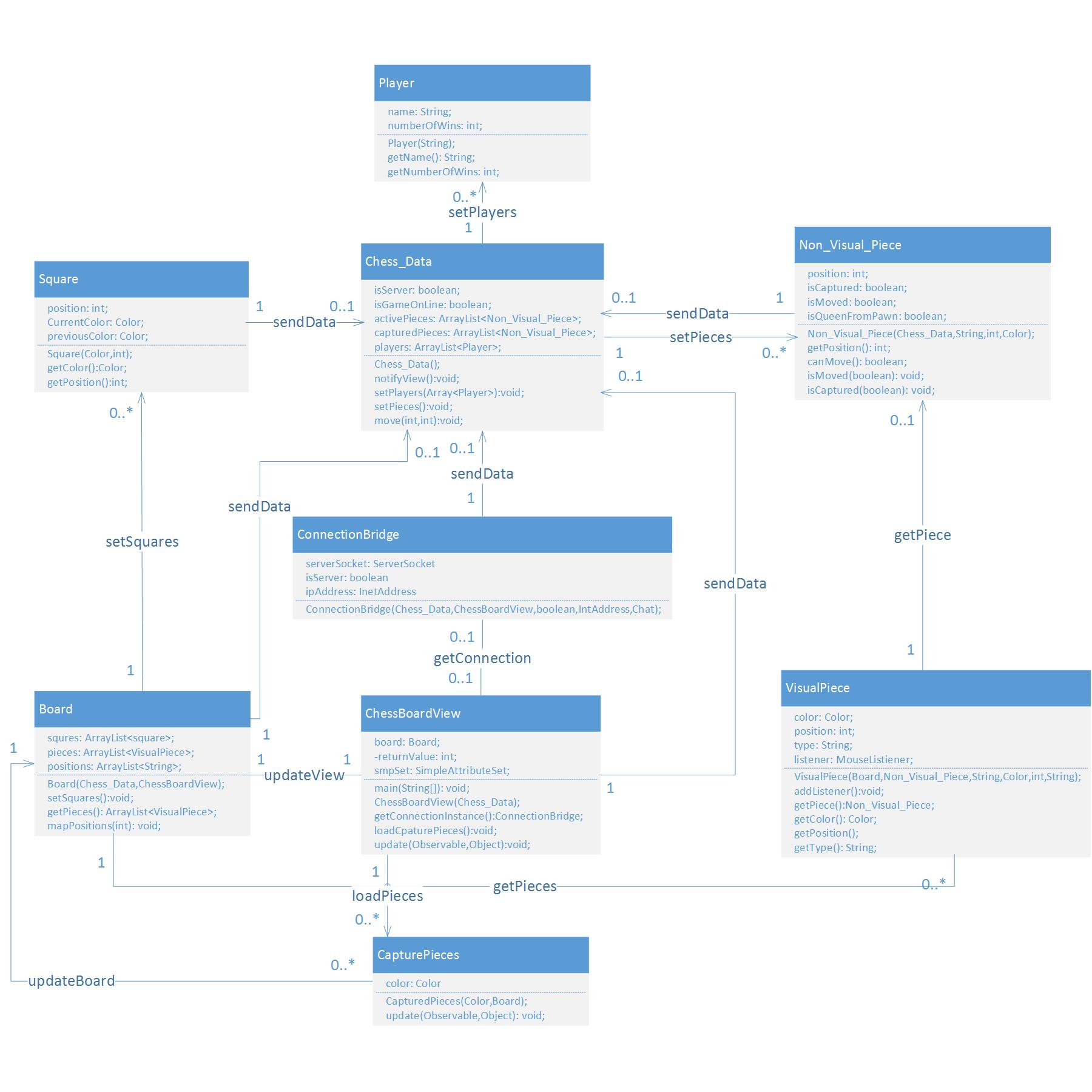
|  |  |  |  |
| --- | --- | --- | --- |
| Conceptual class | Map to | | Actual class |
| Chat | | Chat | |
| Player | | Player | |
| Message | | Emoticons | |

Mapping table of conceptual classes and actual classes

**2.2 Mapping Actual Class Diagram to Conceptual Class Diagram**

|  |  |  |
| --- | --- | --- |
| **Class** | **Conceptual Class** | **Actual Class** |
| **Player-Player** | This class contains all the informations and function associated to a player, such as player\_name, setName(), getName() | This class exists in the actual class diagram but isolated from any other class in the system, which is not appropriate given that it contains all the data and functionalities of a Player object. |
| **Chat-Chat** | This class is composed of chatbox (a class whose responsibility is to display the chat interface). But the main functionality is itself for creating connection bridge between players. | 1. The functionality of **Chat.java** in conceptual class diagram is the same has the functionality of **connectBridge.java** in the actual class diagram. However,  **connectionBridge.java** has no association with **player.java.** 2. Chat.java of the actual class provide GUI representation of the whole Chess’N’Chat system. |
| **Message-Emoticons** | This class contains data and functionalities of all the emoticons which will be used during chat. | This map to emoticon.java in the actual class diagram. Whose main purpose is also related to **Message.java.** |
| **Chatbox-None** | A GUI representation of only the chat application | There exist no class in the actual class with similar functionality as that of the Chatbox class of the conceptual class diagram. |

**3 Chess system (sub-system of Chess’N’Chat)**

****

**Figure 3: Class diagram of chess system**

According to the actual class diagram generated from tool ‘Enterprise Architect - Sparx Systems 8.0’, nine classes (Chess\_Data.java, ConnectionBridge.java, Board.java,Square.java, Player.java, CapturedPieces.java, VisualPiece.java, NonVisualPieve.java ChessBoardView.) are involved in the chess sub-System.

* 1. **Purpose of Actual Class**
* **Chess\_Data.java Class**: This class is data class that provides related data to most of the classes to the Chess’N’Chat system. This by itself does not obey the Object oriented design principles. The Chess\_Data Class are the model for the chess game. This model is an Observable object which when changed notifies the observers in order to update observer views. This model contains all possible moves for the game
* **ChessBoardView.java Clas**s: This is the GUI class which is responsible for creation of chess board interface. And when change occurs in Chess\_Data, its view will change accordingly.
* **Board.java Class**: This is the board of the chess system. It defines the position to place squares and pieces. The Board class is the Board of our Chess Game it consists of a JPanel with the 8 by 8 layout. This class is an Observer so each time change happens to the data class it is notified by executing its update method. It represents it is happened each time the move method of the data class is called thus executing this method
* **Square.java Class**: This class creates the square objects. It also defines the color and position of these objects, which is a template for making Square objects when called upon in the ChessBoardView class they also listen for click events to register their position at the source and at destination.
* **CapturedPieces.java Class**: This class is the panel that our view uses to show the captured pieces on. The update method asks the data class if any piece was captured, if yes it add it to captured piece panel and remove it from the board.
* **VisualPiece.java Class:** This class is the piece that is visual to the user. It is only displayed in our ChessBoardView. The VisualPiece class is the piece that is visual to the user. The Visual Piece knows its color, position, imagePath type and etc....
* **NonVisualPieve.java Class**: This class is the abstract piece that is not visual to the user but it’s the piece that our data class (Chess\_Data) uses to move the piece.
* **Player.java**: contains information about a player object but has no connection with any other class in the Chess’N’Chat system.
* **ConnectionBridge.java**: This class is responsible for establishing connection between client and server. However, the class has no link with the player.java class is not instinctual, because we expect player to be involve in a chat connection.

|  |  |  |  |
| --- | --- | --- | --- |
| Conceptual class | Map to | | Actual class |
| Chess board | | Board | |
| Squares | | Square | |
| Chess pieces | | VisualPiece and NonVisualPiece | |
| Captured pieces | | CapturedPiece | |

**Mapping table of conceptual classes and actual classes**

**3.2 Mapping Actual Class Diagram to Conceptual Class Diagram**

|  |  |  |
| --- | --- | --- |
| **Class** | **Conceptual Class** | **Actual Class** |
| **Chess board -Board** | This class is the board of chess game which contains chess pieces and chess squares classes | This class exists in the actual class diagram but it has so many functionalities, some of which can be what the board class should not perform. |
| **Chess pieces - VisualPiece and NonVisualPieces** | This class has some basic attribute of pieces like colors and positions. | This class does not exist in the actual class diagram but there are visual pieces and non-visual pieces classes in the system, which implement the basic functionality of pieces in the system. |
| **Squares-Square** | This class is the square of the board which piece can move on the black square or white square. | There exist the same class in the real system which is a template for make square object when called upon some square objects will have pieces on them, some will be empty. |
| **Captured pieces –Capturedpiece** | This conceptual class consist the pieces that are captured | This class is the panel that our view uses to show the captured pieces on. |

**Interpretation of the Architecture of Actual System**

The architecture of the actual system represents that there is a deviation from the real world model because the basic and fundamental requirement is that the players should be the primary actors for the chess and chat application but on observation, the player has no connection whatsoever with any other class in the chess’N’Chat application. In addition, other classes of the actual diagram are either god class, data class or have other responsibility that has no relation with the actual purpose of the class. The ChessBoardView Class is one of the god class in the system. It connected to chess board, squares and pieces classes, etc. This view is an observer so each time the change made by the connected classes’ related actions will reflect in this class. It is notified immediately by Chess\_Data class and changes its view accordingly. This class is also a pure GUI class that contain a lot of auto generated non-domain code, which is hard to modify and refactor. Chess\_Data is the data class existing in the project. Almost all other classes require data from this class. Because it records all the moves made by the users and returns the values to other classes. There are also some violations of software development. This also does not abide to the principles of Object Oriented design.

# Code Smells and System Level Refactorings

|  |  |  |
| --- | --- | --- |
|  | **SOURCE CODE SMELLS[1]** | **REFACTORING METHOD[2]** |
| 1 | Duplicated Code | Extraction of duplicated code in two methods of the same class into new class. |
| 2 | Comments | Exact and appropriate comments depicting actual purpose of system functionality. |
| 3 | Uncommunicative Name | Rename variable to correct name such that maintenance becomes easier. |
| 4 | Sub Optimal Code and unused Fields and methods | Operation of removing or deleting unwanted string and string Buffer usage and unused local variable. |
| 5 | Too many methods and fields | Extract collection of large number of methods and variables to new class |
| 6 | God Class | God classes do too many things, are very big and overly complex. |
| 7 | Long Method | To shorten a method is Extract Method. Find parts of the method that seem to go nicely together and make a new method. |
| 8 | Inconsistence naming | The project does not have a consistence naming convention, which cause misunderstanding and confusion. |
| 9 | Feature envy | This method obviously want elsewhere, use the Move method allows you to get it. Sometimes, part of the method suffers from jealousy, envy bit extraction method and the Move method. |
| 10 | Type checking | Type-checking code mainly manifests itself as complicated conditional statements that make the code difficult to understand and maintain.  Replace Conditional with Polymorphism |

**Tables of code smells found and corresponding actions taken for refactoring**

|  |  |
| --- | --- |
| **1** | **Duplicated Code**  **Class –** Visaul\_Piece.java, Non\_Visual\_Piece.java |
|  | **public** **class** Non\_Visual\_Piece **implements** Serializable {  **private** String type;  **private** **int** position;  **private** Color color;  **private** **int** countClicks;    …………………………………………………………………………………………………………………………………………………………  **public** **class** VisualPiece **extends** JPanel {  **private** Color color;  **private** **int** position;  **private** **int** clickCount = 0;  **private** String imagePath; |
|  | Action To be Taken:  Create a new class ‘Piece.java’, extract duplicated code to Piece.java and make Visual\_Piece.java and Non\_Visual\_Piece.java a subclass of Piece.java. |

|  |  |
| --- | --- |
| **2** | **Comments**  **Class -** Chat.java  Line No:123 |
|  | //chatA is the JTextPane where the messages are seen  chatA = **new** JTextPane() { |
|  | Action To be Taken:  Certain comments are not necessary and is self-explanatory. |

|  |  |
| --- | --- |
| **3** | **Uncommunicative Name**  **Class -** Chat.java  Line No: 131 |
|  | Before Refactoring  private Container c;  public void paintComponent(Graphics g)    After Refactoring  private Container container;  public void paintComponent(Graphics graphics) |
|  | Action To be Taken:  Renamed the names of variables and methods such that name explains precisely real purpose. |

|  |  |
| --- | --- |
| **4** | **Sub Optimal Code and unused Fields and methods**  **Class -** Chat.java  **Line No**. – 502 |
|  | Before Refactoring  **public** **class** FontDialog **extends** JDialog {  **private** JCheckBox chkItalic, chkBold;  **private** JButton btnOk, btnColor;  **private** JPanel pnlChk, pnlBtn;  **private** SimpleAttributeSet smpSet1;  After Refactoring  **public** **class** FontDialog **extends** JDialog {  //INSTANCE VARIABLES  **private** JCheckBox chkItalic, chkBold;  **private** JButton btnOk, btnColor;  **private** SimpleAttributeSet smpSet1; |
|  | Action To be Taken:  Deletion of unused fields ‘**private** JPanel pnlChk, pnlBtn;’ from the source code since it is never used |

|  |  |
| --- | --- |
| **5** | **Too many methods and fields**  **Class -** Chat.java  **Line No**. – 502 |
|  | Before Refactoring  private ArrayList<VisualPiece> pieces = new ArrayList<VisualPiece>();  ...  public void removeAllPieces() {  for (int i = 0; i < squares.size(); i++) {  if (squares.get(i).getComponentCount() > 0) {  squares.get(i).remove(0);  }  ...  public ArrayList<VisualPiece> getPieces() {  return pieces;  }}  ...  **public** **void** redrawPieces() {  **for** (**int** i = 0; i < data.getActivePieces().size(); i++) {  **if** (data.getActivePieces().get(i) != **null**) {  Non\_Visual\_Piece p = data.getActivePieces().get(i);  **for** (**int** j = 0; j < pieces.size(); j++) {  VisualPiece peice = pieces.get(j);  **...**  **public** **void** removeCapturedPieces() {  **if** (!data.getCapturedPieces().isEmpty()) {  Non\_Visual\_Piece p = (Non\_Visual\_Piece) data.getCapturedPieces().get(data.getCapturedPieces().size() - 1);  **for** (**int** i = 0; i < pieces.size(); i++) {  **if** (pieces.get(i).getPiece().equals(p)) {  pieces.remove(pieces.get(i));  }  }  }  } |
|  | Action To be Taken:  Firstly, Create a new class VisualPiece.java and extract methods and variable to the newly created class |
|  |  |

|  |  |
| --- | --- |
| **6** | **God Class**  **Class -** Chat.java  **Line No**. 244 and 261 |
|  | Before Refactoring    --------------  --------------  **public** **void** setPlayerIconPath(Object object) {  Packet packet = (Packet) object;  **if** (data.isServer()) {  data.getPlayers().get(1).setImagePath(packet.getPlayerIconPath());  } **else** {  data.getPlayers().get(0).setImagePath(packet.getPlayerIconPath());  }  data.notifyView();  }    **public** **void** setGuestName(Object object) {  Packet packet = (Packet) object;  **if** (data.isServer()) {  data.getPlayers().get(1).setName(packet.getGuestName());  } **else** {  data.getPlayers().get(0).setName(packet.getGuestName());  }  data.notifyView();  }    After Refactoring  **public** **class** ConnectionBridge **implements** Observer {  **private** ServerSocket serverSocket;  **private** Socket socket;  **private** ObjectOutputStream oos;  **private** ObjectInputStream dis;  **private** **boolean** isServer = **true**;  ---------------------------  ---------------------------  **public** ConnectionBridge(Chess\_Data data, ChessBoardView view, **boolean** isServer, InetAddress ipAddress, Chat chat) {  setProperty set=**new** setProperty(); // instance of new class  **if** (packet.getPlayerIconPath() != **null**) {  set.setPlayerIconPath(object);  }  **if** (packet.getGuestName() != **null**) {  set.setGuestName(object);  }  ---------------------------  --------------------------- |
|  | Action To be Taken:  Firstly, We created a new class named setProperty.java. Then, move the related attributes to the new class. Then, move the method setPlayerIconPath and setGuestName to the new class. Finally, let the old class call the new class instance to fulfill the function by doing this refactoring the source class gain the clarity and split responsibility to another class. |

|  |  |
| --- | --- |
| **7** | **Long Method**  **Class -** Chess\_Data.java  **Line No**. 329-359 |
|  | **public** **boolean** isMoveable(**int** x, **int** y) {  Non\_Visual\_Piece p = activePieces.get(x - 1);  //IF PIECE IS KING GO INSIDE THE IF BLOCK  **if** (p.getPieceType().equals("King")) {  **if** (((y - 1) == (x) && (y - 1) % 8 != 0) || ((y - 1) == (x - 2) && (y) % 8 != 0) || (y - 1) == (x - 9) || (y - 1) == (x - 10) || (y - 1) == (x - 8)) {  **if** (activePieces.get(y - 1) != **null**) {  **if** (activePieces.get(y - 1).getColor() != p.getColor()) {  **return** **true**;  }  } **else** {  **return** **true**;  }  } **else** **if** ((y - 1) == (x + 6) || (y - 1) == (x + 7) || (y - 1) == (x + 8)) {  **if** (activePieces.get(y - 1) != **null**) {  **if** (activePieces.get(y - 1).getColor() != p.getColor()) {  **return** **true**;  }  } **else** {  **return** **true**;  }  } //CASTLE THE KING WITH THE ROOK ON THE RIGHT AND WHITE COLOR PIECE IS USED  **else** **if** (activePieces.get(y - 1) != **null**) { |
|  | Action To be Taken:  Decompose into method, change if statement to switch case and comment appropriately. |

|  |  |
| --- | --- |
| **8** | **Inconsistent Naming**  **Class -** Chat.java  **Line No**. – 188 and 221 |
|  | Before Refactoring  All the classes are contained inside the package ‘ChessGameKenai’. But this package is not according to the naming standards. |
|  | Action To be Taken:  Rename Package and Class Names according to naming convention and coding standards. |

|  |  |
| --- | --- |
| **9** | **Feature Envy**  **Class -** ConnectionBridge.java  **Line No**. 248-254 |
|  | Before Refactoring  public void setPlayerIconPath(Object object) {  Packet packet = (Packet) object;  if (data.isServer()) {  data.getPlayers().get(1).setImagePath(packet.getPlayerIconPath());  } else {  data.getPlayers().get(0).setImagePath(packet.getPlayerIconPath());  }  data.notifyView();  } |
|  | Action To be Taken:  Firstly, we extract the method from the ConnectionBridge to the ChessData class. Secondly, the method uses the attribute “data” that is an instance of the ChessData class so it will not create the coupling. Finally, make sure the method is correctly executed. |

|  |  |
| --- | --- |
| **10** | **Type checking**  **Class -** Chess\_Data.java and Choose\_Icon.java  **Line No**. line 233-252 |
|  | Before Refactoring  **switch** (**this**.currentBoard) {  //IF NORMAL\_BOARD EXECUTE THE CASE STATEMENT  **case** Board.*NORMAL\_BOARD*:  **for** (**int** i = 0; i < squares.size(); i++) {  squares.get(i).setBounds((**int**) (455 - squares.get(i).getBounds().getX()), (**int**) (455 - squares.get(i).getBounds().getY()), 65, 65);  squares.get(i).repaint();  **this**.add(squares.get(i));  }  **break**;  //IF FLIPPED\_BOARD EXECUTE THE CASE STATEMENT  **case** Board.*FLIPPED\_BOARD*:  **for** (**int** i = squares.size() - 1; i > -1; i--) {  squares.get(i).setBounds((**int**) (455 - squares.get(i).getBounds().getX()), (**int**) (455 - squares.get(i).getBounds().getY()), 65, 65);  squares.get(i).repaint();  **this**.add(squares.get(i));  }  **break**;  } |
|  | Action To be Taken:  a. Create new a class named “CurrentBoard.java”,  b. Create a new class named ”NormalBoard.java”,  c. Create a new class named ”FlippedBoard.java”;  d. Modify the class “board.java”  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 |

**Code smells Number 1: Duplicated Code**

|  |  |  |  |
| --- | --- | --- | --- |
| **Class name** | **VisualPiece.java class** | **Non\_Visual\_Piece.java** | **Description** |
| Attributes | private Color color;  private int position;  private int clickCount = 0;  private String imagePath;  private String type;  private Non\_Visual\_Piece piece;  private MouseListener listener;  private Board board; | private String type;  private int position;  private Color color;  private int countClicks;  private int previousPosition; | These two classes have some identical methods and attributes existing to fulfill the same functions retrieved from the requirement. The left of the table shows the related attributes and methods are duplicated. We simply just create a super class to contain this identical variables and methods and delete the duplicates. |
| Methods | public String toString()  public void setPosition(int position)  public void setClickCount(int countClicks)  public String getType()  public int getPosition()  public Color getColor() | public String toString()  public void setPosition(int position)  public void setClickCount(int countClicks)  public String getType()  public int getPosition()  public Color getColor() |

# Specific Refactorings that you will implement in Milestone 4

The refactorings we are going to be implemented in Milestone4 are

|  |  |
| --- | --- |
| **Refactoring Number (Reference to the Code Smells Mentioned in above tables)** | **Name** |
| 1 | Duplicated Code |
| 6 | God Class |
| 7 | Long Method |
| 10 | Type checking |

**Reference:**

1. [website] <http://sourcemaking.com/refactoring/bad-smells-in-code>
2. [website] <http://sourcemaking.com/refactoring>
3. [website]http://java.uom.gr/~jdeodorant/index.php?option=com\_content&view=article&id=69&Itemid=53