Homework 4
ECE 590: Towards More Reliable Software

Jeff Fan || zf70@duke.edu

February 6, 2024

Question 1

Code:

```java
package edu.duke.ece651.team16.controller;

import java.io.IOException;
import java.io.PrintWriter;
import java.io.BufferedReader;
import java.io.EOFException;
import java.io.InputStreamReader;
import java.net.Socket;
import java.io.PrintStream;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.core.JsonProcessingException;
import java.util.HashMap;
import java.util.ArrayList;
import java.util.Map;
import java.util.Collections;

public class Client {
  private Socket clientSocket;
  private BufferedReader socketReceive;
  private PrintWriter socketSend;
  private Views view;
  private String color;
  private String room;

  // system input and output
  final PrintStream out;
  final BufferedReader inputReader;

  private boolean ifExit;

  /**
  * Constructor for Client
  *
  * @param clientSocket: client socket
  * @param inputSource:  system input
  * @param out:          system output
  * @throws IOException
  */
  public Client(BufferedReader inputSource, PrintStream out,
  BufferedReader socketReceive, PrintWriter socketSend) throws
   IOException {
    this.out = out;
    this.inputReader = inputSource;
    this.socketReceive = socketReceive;
    this.socketSend = socketSend;
    this.view = new Views(out);
    this.ifExit = false;
    this.clientSocket = null;
    this.color = null;
    this.room = null;
  }

  /**
  * Get the if player exit
  *
  * @return boolean ifExit
  */
  public boolean ifExit() {
    return ifExit;
  }

  /**
  *
  * Receive message
  * from server**@return message*
  *
  * @throws IOException
  */

  public String recvMsg() throws IOException {
    boolean received = false;
    String msg = null;
    while (!received) {
      try {
        msg = socketReceive.readLine();
        if (msg != null) {
          received = true;
        }
      } catch (IOException e) {
        System.out.println("Failed to receive message.");
      }
    }
    return msg;
  }

  /**
  * Send response to server
  *
  * @param response: response message
  * @throws IOException
  */
  public void sendResponse(String response) throws IOException
   {
    this.socketSend.println(response);
  }

  /**
  * Wait for everyone to finish
  *
  * @throws IOException
  */
  public void waitEveryoneDone() throws IOException {
    String prompt = recvMsg();
    boolean done = false;
    while (!done) {
      if (prompt.equals("stage Complete")) {
        done = true;
      } else {
        prompt = recvMsg();
      }
    }
  }

  /**
  * Player choose color and send to server
  *
  * @throws IOException
  */
  public boolean playerChooseColor(String color) throws
 IOException {
    try {
      sendResponse(color);
      // see if selection is valid
      String prompt = recvMsg();
      if (prompt.equals("Valid")) {
        // successful choose color
        out.println("Successfully set color: " + color);
        this.color = color.toLowerCase();
        return true;
      }
    } catch (EOFException e) {
      out.println(e.getMessage());
    }
    return false;
  }

  // set color
  public void setColor(String color) {
    this.color = color;
  }

  public void setRoomID(String room) {
    this.room = room;
  }

  /**
  * Get player color
  *
  * @return color
  */
  public String getColor() {
    return this.color;
  }

  public String getRoom() {
    return this.room;
  }

  /**
  * Player choose playerNum and send to server
  *
  * @param num
  */
  public void playerChooseNum(String num) throws IOException {
    // useless in gui: follow
    String prompt = recvMsg();
    if (prompt.equals("finished stage")) {
      out.println(prompt);
      return;
    }
    // useless in gui: above
    try {
      sendResponse(num);
      prompt = recvMsg();
      if (prompt.equals("Valid")) {
        out.println("Successfully choose number of players: "
  + num);
      }
    } catch (EOFException e) {
      out.println(e.getMessage());
    }
  }

  /**
  * Player choose territory and set unit number
  *
  * @throws IOException
  * @return boolean if player finished placement
  */
  public void playerAssignUnit(String input) throws
  IOException {
    String prompt = recvMsg();
    if (prompt.equals("finished stage")) {
      out.println("Finished Placement. Please wait for other
  players to place their students.");
      return;
    }
    try {
      sendResponse(input);
      prompt = recvMsg();
    } catch (EOFException e) {
      out.println(e.getMessage());
    }
  }

  /**
  * Player send action input to server
  * clientInput = ["order", "T1, T2, numbers"]
  *
  * @throws IOException
  */
```

```java
  public String playerOneAction(ArrayList<String> clientInput)
   throws IOException {
    recvMsg();// String choices = view.displayEntry(recvMsg())
; old version
    System.out.println("after recvMsg");
    sendResponse(clientInput.get(0)); // send order (a or m)
    System.out.println("after sendResponse");
    System.out.println(clientInput);
    if (clientInput.get(0).equals("d")) {
      String msg = recvMsg();
      System.out.println(msg);
      if (msg.equals("finished stage")) {
        out.println("Finished 1 Turn of orders. Please wait
for other players to issue orders.");
      }
      return "Valid";
    }
    String prompt = recvMsg(); // "Please enter <Territor
......"
    try {
      if (clientInput.get(0).equals("a") || clientInput.get(0)
.equals("m") || clientInput.get(0).equals("u")
      || clientInput.get(0).equals("l") || clientInput.get(0).
equals("s")) {
        sendResponse(clientInput.get(1));
      }
      // research
      prompt = recvMsg(); // Valid, or respective error
message, from serverside "tryAction" result
      return prompt;
    } catch (EOFException e) {
      out.println(e.getMessage());
    }

    return "Invalid";
  }

  /**
  * Check if the move/attack input format is correct
  *
  * @param clientInput the input from client
  * @return if move input format is valid
  */
  public boolean checkMoveInputFormat(String clientInput) {
    String[] input = clientInput.split(", ");
    if (input.length != 3) {
      return false;
    }
    String unitNum = input[2];
    if (!unitNum.matches("[0-9]+")) {
      return false;
    }
    return true;
  }

  /**
  * Client receives message from server about if the watch
option is valid or
  * not
  *
  * @throws IOException
  */
  public void playerChooseWatch(String clientInput) throws
IOException {
    // receive color choosing prompt from server
    recvMsg(); // prompt
    try {
      sendResponse(clientInput);
      // see if selection is valid
      recvMsg(); // valid
      if (clientInput.toLowerCase().equals("e")) {
        // System.exit(0);
        this.ifExit = true;
        return;
      } else {
        view.setWatch();
        return;
      }
    } catch (EOFException e) {
      out.println(e.getMessage());
    }
  }

  /**
  * Get client socket
  *
  * @return clientSocket
  */

  public Socket getClientSocket() {
    return this.clientSocket;
  }

  /**
  * Set client socket
  *
  * @param socket
  */
  public void setClientSocket(Socket socket) {
    this.clientSocket = socket;
  }

  /**
  * Player choose room and send to server
  *
  * @param roomID
  * @return ifEnter
  * @throws IOException
  */
  public String playerChooseRoom(String roomID) throws IOException {
    try {
      sendResponse(roomID); // send roomId
      String ifEnter = recvMsg(); // Room created, or "Room joined." or "Room exceeded player number, game already
      // started"
      out.println(ifEnter);
      return ifEnter;
    } catch (EOFException e) {
      out.println(e.getMessage());
    }
    return "";
  }
}
```

Question 2

Code Smell: Long Method.

Lines 206-234: Long methods violate the Single Responsibility Principle (SRP) by handling multiple tasks within a single method. They are harder to understand, maintain, and test. Long methods often contain duplicated code and tend to be less modular and flexible. Long methods typically accumulate functionality over time, as developers add new features or modify existing ones without refactoring. This accumulation leads to a bloated method that performs multiple tasks, making it harder to comprehend and modify. Additionally, long methods often contain duplicated code segments, as developers may copy and paste code rather than creating reusable components. Overall, long methods hinder code readability, maintainability, and extensibility.

Code Smell: Duplicate Code.

Lines 161-198: Duplicate code violates the Don't Repeat Yourself (DRY) principle, leading to maintenance issues, increased complexity, and decreased code quality. Duplicate code arises when identical or similar code fragments appear in multiple places within the codebase. This duplication can occur due to copy-pasting code segments, lack of awareness of existing functionality, or failure to abstract common functionality into reusable components. Duplicate code increases the likelihood of inconsistencies, as developers may update one instance of the code but forget to update others. It also makes code maintenance more challenging, as changes need to be applied in multiple places, increasing the risk of introducing errors.

Code Smell: Feature Envy.

Lines 117-132, 186-198: Feature envy occurs when a method accesses or manipulates data of another class more than its own data, violating the principle of encapsulation and leading to tight coupling between classes. Feature envy suggests that the responsibilities of a method might be better placed within the class that owns the data being manipulated. When a method in one class extensively interacts with data from another class, it indicates that the functionality might be better suited to the class owning that data. This violation of encapsulation can lead to increased coupling between classes, making the code harder to understand, maintain, and refactor. Feature envy also reduces the cohesion of the class, as it suggests that the method's responsibilities are spread across multiple classes instead of being concentrated within a single class.

Code Smell: Data Clumps.

Lines 22, 23: Data clumps occur when groups of variables frequently appear together, suggesting that they may belong together in a separate class or structure. This violates the principle of high cohesion and low coupling. Data clumps typically arise when related data elements are scattered throughout the codebase and are frequently manipulated together. This indicates a lack of abstraction and encapsulation, as the related data elements should ideally be grouped together into a single entity. Without encapsulating related data into a separate class or structure, the code becomes less modular and flexible, making it harder to maintain and extend. Data clumps also hinder code readability, as developers must mentally track the relationships between different data elements scattered throughout the codebase.

Question 3

Code Smell: Long Method.

Long methods make the codebase harder to understand, maintain, and extend. For instance, imagine a scenario where a developer needs to debug an issue related to a specific player action in the playerOneAction method. With the method spanning multiple pages or containing hundreds of lines of code, finding the relevant code segment becomes akin to finding a needle in a haystack. This increases the likelihood of introducing errors during debugging or modifying the method, as developers may inadvertently overlook certain parts or fail to understand the entire flow of execution.

Code Smell: Duplicate Code.

Duplicate code leads to maintenance nightmares and inconsistencies. Consider a situation where a bug is discovered in the duplicated code block within playerAssignUnit and playerChooseNum methods. If a fix is applied to one method but forgotten in the other, it introduces inconsistency and increases the risk of regression. This inconsistency can lead to unexpected behavior during runtime and erode user trust. Moreover, the presence of duplicate code increases the cognitive load on developers, as they must remember to update all occurrences when making changes, which is error-prone and time-consuming.

Code Smell: Feature Envy.

Feature envy violates encapsulation and leads to tight coupling between classes. For instance, in the playerChooseColor method, direct access to the color field of the Client class suggests that the method is overly concerned with the internal details of managing player information. If the internal representation of player information changes, such as adding validation logic or additional attributes, every method accessing these fields directly would need to be updated. This tight coupling increases the risk of unintended consequences and makes the system more fragile to changes.

Code Smell: Data Clumps.

Data clumps hinder maintainability and extensibility by scattering related data elements throughout the codebase. For example, imagine a scenario where color and room are frequently manipulated together in various parts of the system. Without encapsulating them into a separate entity like PlayerInfo, modifying or extending the representation of player data becomes cumbersome. This increases the likelihood of introducing errors during development or maintenance, as developers must track and update related data elements across multiple places. Moreover, data clumps hinder code readability, as developers must mentally connect scattered data elements to understand their relationships and dependencies, leading to confusion and potential errors.

Question 4

Code Smell: Long Method.

Principle to Avoid: Single Responsibility Principle (SRP) states that a class should have only one reason to change, meaning it should only have one responsibility. By breaking down the playerOneAction method into smaller, more focused methods, each responsible for a single task, we can improve readability, maintainability, and testability. For example, we can have separate methods for receiving messages, sending responses, and processing player actions. This ensures that each method is focused on a specific task and adheres to the SRP.

```
//refactored code
public class Client {
  public String playerOneAction(ArrayList<String> clientInput)
   {
    recvMsg(); // Receive message
    sendOrder(clientInput); // Send player order
    if (clientInput.get(0).equals("d")) {
      handleDoneMessage(); // Handle message indicating end of
   stage
      return "Valid";
    }
    return processPlayerAction(clientInput); // Process player
   action and return result
   }

```

```java
  // Helper methods for refactoring

  private void handleDoneMessage() {
    out.println("Finished 1 Turn of orders. Please wait for other players to issue orders.");
  }

  private String processPlayerAction(ArrayList<String> clientInput) {
    String prompt = recvMsg(); // Receive prompt message
    try {
      if (isValidAction(clientInput)) {
        sendActionResponse(clientInput); // Send action response to server
      }
      prompt = recvMsg(); // Receive response from server
      return prompt;
    } catch (IOException e) {
      out.println(e.getMessage());
    }
    return "Invalid";
  }

  private boolean isValidAction(ArrayList<String> clientInput) {
    return clientInput.get(0).equals("a") || clientInput.get(0).equals("m") || clientInput.get(0).equals("u")
    || clientInput.get(0).equals("l") || clientInput.get(0).equals("s");
  }

  private void sendActionResponse(ArrayList<String> clientInput) throws IOException {
    if (clientInput.get(0).equals("a") || clientInput.get(0).equals("m") || clientInput.get(0).equals("u")
    || clientInput.get(0).equals("l") || clientInput.get(0).equals("s")) {
      sendResponse(clientInput.get(1));
    }
  }
}

```

Refactored code: The original long method playerOneAction has been broken down into smaller, more focused methods. Here's how the refactoring was done:

Extracted Helper Methods: The code responsible for receiving messages from the server, sending player orders, handling end of stage messages, and processing player actions was extracted into separate helper methods. This breaks down the original long method into smaller, more manageable parts, each with a single responsibility.

Improvement in Readability and Maintainability: By breaking down the original long method into smaller, focused methods, the code becomes easier to understand, maintain, and test. Each method now has a clear purpose and responsibility, making it easier to reason about and modify in the future.

Encapsulation of Functionality: The functionality related to sending and receiving messages, processing player actions, and handling responses from the server is encapsulated within the Client class. This promotes better encapsulation and adheres to the Single Responsibility Principle, ensuring that each method is responsible for a single task.

Clear Flow of Execution: The refactored code follows a clear flow of execution, starting with receiving a message, sending an order, handling special cases like the end of stage message, and finally processing player actions. This clear flow makes it easier to follow the logic of the program and understand its behavior.

Code Smell: Duplicate Code.

Principle to Avoid: Don't Repeat Yourself (DRY) principle advocates for eliminating duplicated code by abstracting common functionality into reusable components. In this case, we can refactor the duplicated code into a separate method, which can be called from both playerAssignUnit and playerChooseNum methods. By doing so, we reduce redundancy, improve maintainability, and decrease the likelihood of introducing bugs due to inconsistencies.

```java
//refactored code
public class Client {
  public void playerAssignUnit(String input) {
    handleStageCompletion(); // Check for stage completion
    try {
      sendResponse(input); // Send response to server
      recvMsg(); // Receive message from server
    } catch (IOException e) {
      out.println(e.getMessage());
    }
  }

  public void playerChooseNum(String num) {
    handleStageCompletion(); // Check for stage completion
    try {
      sendResponse(num); // Send response to server
      recvMsg(); // Receive message from server
    } catch (IOException e) {
      out.println(e.getMessage());
    }
  }

  // Helper methods for refactoring
  private void handleStageCompletion() {
    String prompt = recvMsg(); // Receive prompt message
    if (prompt.equals("finished stage")) {
      out.println("Finished Placement. Please wait for other players to place their students.");
    }
  }
}
```

Refactored code: In the refactored code, the duplicate code for checking stage completion and sending/receiving messages from the server has been refactored into a single helper method handleStageCompletion. Here's how the refactoring was done:

Extracted Helper Method: The code for checking stage completion and handling the subsequent actions was duplicated in both playerAssignUnit and playerChooseNum methods. This duplicated code has been extracted into a separate helper method handleStageCompletion.

Reuse of Code: Both playerAssignUnit and playerChooseNum methods now call the handleStageCompletion method to check for stage completion. This promotes code reuse and eliminates redundancy.

Improvement in Maintainability: By eliminating duplicated code and centralizing the logic for handling stage completion in a single method, the codebase becomes easier to maintain. Any changes or updates to the stage completion logic can now be made in one place, reducing the risk of inconsistencies and errors.

Code Smell: Feature Envy.

Principle to Avoid: Low Coupling suggests that classes should have minimal dependencies on other classes, promoting independence and modularity. High Cohesion advocates for ensuring that the responsibilities of a class are closely related and focused. To address feature envy, we can refactor the methods to delegate operations to the Client class itself, ensuring that the logic related to color and room stays within the Client class. This promotes higher cohesion and reduces coupling between classes, leading to a more maintainable and flexible codebase.

```
//refactored code
public class Client {
  private PlayerInfo playerInfo; // New class to encapsulate
 color and room

  // Methods after refactoring

  public void playerChooseColor(String color) {
    playerInfo.setColor(color); // Setting color through
 encapsulated object
  }

  public String playerChooseRoom(String roomID) {
    playerInfo.setRoom(roomID); // Setting room through
 encapsulated object
    String ifEnter = recvMsg(); // Receive response from
 server
    out.println(ifEnter); // Print response to console
    return ifEnter; // Return response
  }

  // Other methods...
}
```

Refactored code: In the refactored code, the feature envy smell is addressed by encapsulating the related data (color and room) within a separate PlayerInfo class. Here's how the refactoring was done:

Encapsulation of Data: The color and room fields, which were previously part of the Client class, are now encapsulated within a separate PlayerInfo class.

Delegation of Responsibility: The responsibility for setting the color and room values is delegated to methods within the PlayerInfo class (setColor and setRoom). This ensures that the Client class is not overly concerned with the internal details of managing player information, reducing feature envy.

Improved Cohesion and Reduced Coupling: By encapsulating related data and functionality within the PlayerInfo class, the cohesion of the Client class is improved. Additionally, the coupling between the Client class and the specific details of player information is reduced, promoting a more modular and maintainable design.

Code Smell: Data Clumps.

```java
public class Client {
  private String color;
  private String room;
}
```

Principle to Avoid: High Cohesion - Low Coupling. High Cohesion suggests that related functionality should be grouped together within a class or module. In this case, we can create a new class, let's say PlayerInfo, that encapsulates both color and room fields. By doing so, we improve cohesion by grouping related data and functionality together. Additionally, Low Coupling suggests that classes should have minimal dependencies on each other. By having a separate PlayerInfo class, we reduce the coupling between the Client class and the specific details of player information, leading to a more modular and maintainable design.

```java
//refactored code
public class Client {
  private PlayerInfo playerInfo;
}

public class PlayerInfo {
  private String color;
  private String room;

  // Getters and setters for color and room...

  // Other methods...
}
```

Refactored code: color and room were encapsulated in a separate PlayerInfo class, grouping related data together and improving maintainability by reducing redundancy.