SENG201 Project Report

Jack Miller (24170316) and Robert Ivill (46012819)

For this project, we have decided to create a basketball simulator game where users can create a team and play through a season with the goal of earning as many points as possible. Users who play are first prompted to enter a team name and choose the difficulty. After this, they get a starting budget and 10 picks in the draft to build their initial team. Once the user is happy with their team, they can enter the main game where they can see their players and their stats, pick which players are on the bench and are on the starting lineup and go to the player market to buy and sell players. They can also see the items they own and go to the item shop to buy items which increase a player's stats or make a player healthy. When the user chooses to play a game, they have a choice of whether they want to play an easy, medium, or hard matchup. The difficulty decides how easy it is for the team to win, but also increases the possible money and points to be earned from the matchup. When playing a game, the player can make final changes to the lineup and apply items to players. Then the first half of the match is simulated, and the user can make more changes to their team. After the game is finished, the user is notified if they won or lost and shown how much money and points were earned. At the end of the season, the user is shown their final team and their ratings, the total points and how much money they had leftover.

The first class we created was the Athlete class, which held information about a player within the game. These athletes have their fitness, offense and defence stats that are randomly implemented and determine the players overall rating. They have a stamina stat that affects their performance and can be injured if it reaches 0. From this class comes the Team class, that holds an ArrayList of Athletes to track the users team. It modifies which players are in the starting five and which are reserves, holds the inventory of the user’s items, which week of the season they are in and how many points they have. The Purchasable class is the superclass for both the Athlete and Item classes. It holds the price and the methods related to the price of both athletes and items. The item class is also a superclass for the five different items in the game (Dumbbells, Energy Drinks, Jump Rope, Panadol, and Treadmill). It holds the description of each item and has an abstract method that allows these items to be consumed by athletes. The RandomEvent class is the superclass for the various random events that can happen from week to week. These include a player's stats being changed, a player becoming injured and in exceedingly rare cases a player leaving the club. These classes and their methods are all accessed from the various GUI (graphical user interfaces) windows that form the main game. The WindowManager class controls the flow of the game and has all functions which open each window. It initializes the start-up window and then flows from the draft to the locker and all the other various windows. Apon later reflection of our approach to planning the structure and design, it was decided that we did not allocate enough time to this, which had an adverse effect on the final product. It has been a big learning experience in terms of the importance of planning.

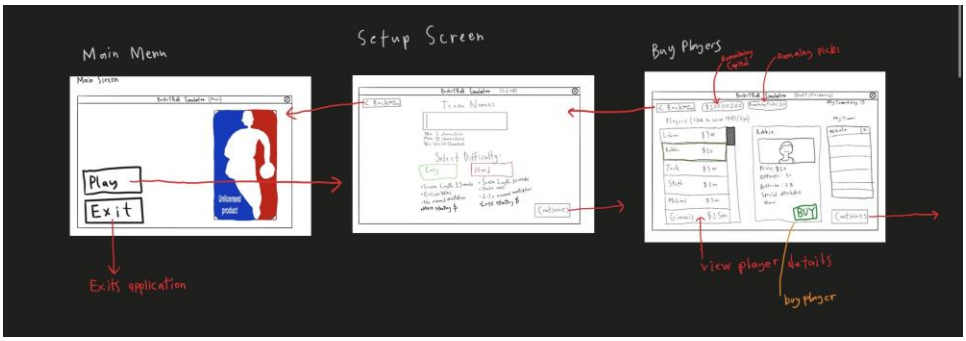
We got a Unit Test coverage of 15.3%. This is less than what we were expecting to achieve. This is because we failed to do Unit Tests for the window application classes as we found it quite

difficult to do. These classes held a large percentage of the overall number of instructions in our project. So, instead we followed the instructions of the project specifications and just did them for the smaller, basic classes. The tests made sure that most of the methods within these classes were covered, especially the less simple methods like getters and setters but larger ones that used logic and other methods within them. These tests were helpful in verifying that our code was working as envisioned.

This project was a tough one. As time creeped toward the due date, the idea of being able to submit a fully functional game was increasingly harder to picture. However, we managed to get there and are happy to be able to deliver our game. At times, it felt like we were not prepared enough to be able to create something like this, which resulted in quite a fair amount of time and effort spent on finding work arounds and solutions to problems that appeared. This project ended up becoming the main focus of our university studies, despite all the other requirements from our other courses. Time management and planning have been two major weaknesses of ours that we have identified throughout the duration of this project, and it would be fair to say that both members of this team could and have benefitted from learning this the hard way.

There were some issues with our approach to this project. For starters, we should have spent some of our time over the mid-term break to get our project started and got some sort of understanding of what we were going to be doing over the next term. We also should have created a more robust plan of what was going to be done each week and set more specific goals for us to complete week by week. Due to this, we ended up not realizing that we needed to add more requirements to our project and the distribution of time spent was skewed towards the due date. Our communication could have been improved as well. It was decided that Jack would handle more of the front-end part of the project, including GUI design and making sure that the game handled the user's interactions properly, and Robbie handled more of the back-end part of the project including class creation and handling the structure, data, and logic of our game. We should have been more collaborative and had a better understanding of what we were doing and what we needed from each other to make both of our lives easier. We did have some problems with our git repository at the beginning, but we did find a solution and were able to use it effectively and efficiently to our advantage once we had figured out a system of using it. All these factors from this project have given us a great learning experience when it comes to software development and working within a team and we will be able to use this in future projects, software development or not.

Jack and Robbie both agreed that we believe we had an even split of the contribution, 50% each. We would estimate that we both ended up spending about 50 hours each on this project.



Chat GPT usage:

how would I check for special characters in an input in java

ChatGPT

You can use a regular expression to check for special characters in a Java string. Here's an example code snippet that uses the **`matches()`** method of the **`String`** class to check if the input contains any special characters:

javaCopy code

```
String input = "Hello, World!"; // replace this with your actual input

if (input.matches(".*[^a-zA-Z0-9 ].*")) {
    System.out.println("Input contains special characters.");
} else {
    System.out.println("Input does not contain special characters.");
}
```

In this example, the regular expression **`.*[^a-zA-Z0-9 ].*`** matches any string that contains one or more characters that are not letters, digits, or spaces. The **`^`** character inside the square brackets means "not", so **`[^a-zA-Z0-9 ]`** matches any character that is not a letter, digit, or space.

The **`matches()`** method returns **`true`** if the input matches the regular expression, and **`false`** otherwise. In the example above, if the input contains any special characters, the output will be "Input contains special characters.", otherwise it will be "Input does not contain special characters."

how would I add a scroll bar to a jlist in window builder java?

ChatGPT

To add a scroll bar to a **JList** in Window Builder for Java, you can follow these steps:

1. Drag and drop a **JScrollPane** from the Swing Controls palette onto your GUI designer.
2. Resize the **JScrollPane** to the desired size and position it so that it surrounds your **JList**.
3. Drag and drop your **JList** component onto the **JScrollPane**.
4. Make sure that the **JList** is selected, then open the Properties view.
5. In the Properties view, locate the **visibleRowCount** property and set it to the number of rows you want to be visible in the **JList**.
6. In the Properties view, locate the **model** property and set it to the **ListModel** you want to use for the **JList**.
7. Run your application to see the **JList** with the scroll bar in action.

Note: If your **JList** already has data in it, you may need to reset the **model** property after dragging and dropping the **JList** onto the **JScrollPane**. You can do this by selecting the **JList**, opening the Properties view, and setting the **model** property to the current **ListModel** used by the **JList**.

how would I do it if I am using an arraylist?

ChatGPT

If you are using an **ArrayList** to populate your **JList**, you can create a custom **ListModel** that is backed by your **ArrayList** and then use that **ListModel** to set the model of your **JList**. Here is an example:

javaCopy code

```
import java.util.ArrayList;
import javax.swing.AbstractListModel;

public class ArrayListModel<E> extends AbstractListModel<E> {
    private ArrayList<E> data;

    public ArrayListModel(ArrayList<E> data) {
        this.data = data;
    }

    @Override
    public int getSize() {
        return data.size();
    }

    @Override
    public E getElementAt(int index) {
        return data.get(index);
    }

    public void add(E element) {
        int index = data.size();
```

```
        data.add(element);
        fireIntervalAdded(this, index, index);
    }

    public void remove(int index) {
        data.remove(index);
        fireIntervalRemoved(this, index, index);
    }
}
```

In this example, the **`ArrayListModel`** class extends **`AbstractListModel`** and implements the **`getSize()`** and **`getElementAt()`** methods required by the **`ListModel`** interface. It also provides **`add()`** and **`remove()`** methods for adding and removing elements from the **`ArrayList`**, which trigger the appropriate events to notify the **`JList`** of the change.

To use this custom **`ListModel`** with your **`JList`**, you can create an instance of **`ArrayListModel`** and pass in your **`ArrayList`** as the data source, like this:

javaCopy code

```
ArrayList<String> myData = new ArrayList<String>();
myData.add("Item 1");
myData.add("Item 2");
myData.add("Item 3");

ArrayListModel<String> listModel = new ArrayListModel<String>(myData);
JList<String> myList = new JList<String>(listModel);

JScrollPane scrollPane = new JScrollPane(myList);
```

In this example, the **`ArrayListModel`** is created with an **`ArrayList<String>`** as the data source, and then used to create a **`JList<String>`** component. The **`JList`** is then wrapped in a **`JScrollPane`**, which will provide the scroll bar functionality. Finally, the **`JScrollPane`** can be added to your GUI designer as needed.

I want to be able to select players from a Jlist, which will then be added to an arraylist, and removed from the jlilst. The players that are selected should then be placed in another identical scrollable jlist on the other side of the screen. How would I implement this?

ChatGPT

To implement the functionality you described, you can add event listeners to the JList for selection events, and then handle the selection by adding the selected player to an ArrayList and removing it from the original JList. Additionally, you can create a second JList to display the selected players on the other side of the screen.

Here's an example implementation:

javaCopy code

```
import java.util.ArrayList;
import java.util.List;
import javax.swing.DefaultListModel;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.ListSelectionModel;
import javax.swing.JScrollPane;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

public class PlayerSelectionExample {
    public static void main(String[] args) {
        // Create a JFrame to hold the components
        JFrame frame = new JFrame("Player Selection");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(500, 400);

        // Create a DefaultListModel to store the players
        DefaultListModel<String> playerListModel = new DefaultListModel<>();
        playerListModel.addElement("Player 1");
        playerListModel.addElement("Player 2");
        playerListModel.addElement("Player 3");
        playerListModel.addElement("Player 4");
        playerListModel.addElement("Player 5");

        // Create the original JList with the playerListModel
        JList<String> playerList = new JList<>(playerListModel);
        playerList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        // Create a JScrollPane and add the playerList to it
        JScrollPane scrollPane = new JScrollPane(playerList);
        scrollPane.setBounds(22, 132, 225, 308);
        frame.getContentPane().add(scrollPane);

        // Create a DefaultListModel to store the selected players
        DefaultListModel<String> selectedPlayerListModel = new
DefaultListModel<>();

        // Create the second JList for selected players
        JList<String> selectedPlayerList = new JList<>(selectedPlayerListModel);
        selectedPlayerList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        // Create a JScrollPane for the selectedPlayerList
        JScrollPane selectedScrollPane = new JScrollPane(selectedPlayerList);
        selectedScrollPane.setBounds(253, 132, 225, 308);
        frame.getContentPane().add(selectedScrollPane);
```

```
        // Add a ListSelectionListener to the playerList
        playerList.addListSelectionListener(new ListSelectionListener() {
            @Override
            public void valueChanged(ListSelectionEvent e) {
                if (!e.getValueIsAdjusting()) {
                    // Get the selected player
                    String selectedPlayer = playerList.getSelectedValue();

                    // Add the selected player to the selectedPlayerListModel
                    selectedPlayerListModel.addElement(selectedPlayer);

                    // Remove the selected player from the playerListModel
                    playerListModel.removeElement(selectedPlayer);
                }
            }
        });

        frame.setVisible(true);
    }
}
```

In this example, we create two **`DefaultListModel`** objects, **`playerListModel`** and **`selectedPlayerListModel`**, to store the players in the original and selected player lists, respectively. We create two **`JList`** objects, **`playerList`** and **`selectedPlayerList`**, with their respective list models.

We add a **`ListSelectionListener`** to the **`playerList`**, which listens for selection events. When a player is selected in the **`playerList`**, the listener's **`valueChanged`** method is called. Inside this method, we retrieve the selected player, add it to the **`selectedPlayerListModel`**, and remove it from the **`playerListModel`**.

Finally, the two **`JList`** objects are added to separate **`JScrollPane`** objects, which are then added to the **`JFrame`** for display.

---

public static void teamInjuries() {

for (Athlete player : Team.playerTeam.getPlayers()) {

player.updateInjured();

if (player.getInjured()) {

Team.playerTeam.getPlayers().remove(player);

injuredPlayers.add(player);

}

} for (Athlete player : injuredPlayers) { player.updateInjured();

if (!player.getInjured()) {

Team.playerTeam.getPlayers().add(player);

injuredPlayers.remove(player);

}

}

}

ChatGPT

The code you provided seems to have a logical issue. When you iterate over the **`Team.playerTeam.getPlayers()`** list and remove an injured player from it, the iterator becomes invalid, and it may lead to unexpected behavior or throw a **`ConcurrentModificationException`**.

To fix this issue, you can use an iterator and its **`remove()`** method instead. Here's an updated version of your code:

javaCopy code

```
public static void teamInjuries() {
    Iterator<Athlete> iterator = Team.playerTeam.getPlayers().iterator();
    while (iterator.hasNext()) {
        Athlete player = iterator.next();
        player.updateInjured();
        if (player.getInjured()) {
            iterator.remove();
            injuredPlayers.add(player);
        }
    }

    iterator = injuredPlayers.iterator();
    while (iterator.hasNext()) {
        Athlete player = iterator.next();
        player.updateInjured();
        if (!player.getInjured()) {
            iterator.remove();
            Team.playerTeam.getPlayers().add(player);
        }
    }
}
```

By using the iterator's **`remove()`** method, you can safely remove elements from the list without causing any issues.

Note: Make sure to initialize the **`injuredPlayers`** list somewhere before using it in the **`teamInjuries()`** method.

I hope this resolves the issue you were facing. Let me know if you have any further questions!