SENG201 Project Report - Football Manager:
Callum Whitehead (cwh74: 63125122)
Harry Ellis (hel46: 81186884)

Structure of 'Football Manager'

Game Logic Classes (Refer to the UML chart for connections)***:***

- **Game** class handles the main game loop, where it sets game parameters like user chosen difficulty, season length and is used to initialise the GUI
- **Athlete** class sets parameters for an individual athlete. An 'Athlete' is used in the **Team, Market** and **PotentialPlayers** classes. It has methods for the usual getters' and setters' for its' attributes and has a method to apply an **Item**
- **Item** class is used to define an item that a **Player** can apply to an **Athlete** object and serves as a generalisation of potential items that can be generated
- **Market** class is the class that holds the 'purchasables' that a **Player** can buy to add to their **Team**. It has methods for generating an **Athlete** and **Item** waiver list (through **PotentialPlayers**) and rotating through a list of **Item** objects and a randomly generated list of **Athlete** objects that a **Player** can buy. A **Player** can also sell athletes and items at the market. Also has methods to update **Player** attributes.
- **Player** class represents the user playing the game. A player has a money balance and an inventory which holds **Item**s. The Player class includes methods to add, use and sell **Items**
- **PotentialPlayers** class is responsible for generating athletes. It generates random values for an athlete's skill attributes as well as picking a random name for the athlete. The athletes generated here are sent to the market for the player to buy and also used to form an opposing team for the player to play against.
- **Match** class handles the main gameplay logic.
- **Team** class represents the users team which consists of two lists of athletes, a starting athlete and a reserve athlete list. The starting athlete list is used for gameplay and the reserve list is used for substitutions. Methods included for substitutions
- **RunApp** game executable (holds main function which launches the game)

Game GUI Classes (Refer to the UML chart for connections)**:**
AthleteSelectionGUI (initial player team selection), GameEndGUI (end of the game), InventoryGUI (players' inventory), MarketGUI (market view where the player buys and sells), MatchSelectionGUI (Stadium GUI where the player chooses 3 opposing teams with different difficulties), MatchSubGUI (Substitutions for a players' team menu during match), SetupGUI(initial player selected variables: season length, team name and difficulty), SubstituionGUI(Same as MatchSubGUI but out of match), trainGUI (special GUI that pops up when a user selects to take a bye)

Design decisions
The game does not include any inheritance or interface classes. This is because of a lack of understanding of interfaces/inheritance and the way we initially coded our game where we believed every class/object was too unique to justify using any inheritance/interface. It was not until later into project development where we saw a way to implement inheritance by combining an Athlete and an Item object into a Purchasable inheritance class but found that there was not enough time to implement it due to time constraints.

Testing

Our JUnit testing on the game logic classes had 64% coverage, according to EclEmma. Although, we believe that these JUnit tests did not test the games' logic at depth and were testing only boundary instead of testing how certain functions interact with each other. This can be attributed to the fact that we built the code first, and then added the tests after, so our project was not test driven. However, when it came to testing our GUI, we did extensive tests each day to see if the GUIs worked accordingly like for example a label outputting the right text when an action is performed. Our GUI testing shows in a solid final product for the user.

Thoughts and Feedback

A takeaway that we have discovered from doing this project is that everything never goes exactly as planned. For example, we had planned to have the command line version of the game done before getting started on the GUI but we found that due to time constraints we had to shift full focus to the GUI. We were also having to shift priorities constantly and would have to skip on 'adding the bells and whistles' in order to only just meet the specifications provided due to time constraints. All in all, we both believed that this project was an excellent learning experience in both Java OO programming and project management.

Retrospective

What went well:

- **Game GUI**
  We believe that the final GUI of the game is solid as there are little to no bugs.
- **Specifications Met**
  Our final game met most, if not, all specifications the project set for us.

What did not go well:

- **Game Logic Testing**
  The JUnit tests that we developed were not enough for a solid depth.
- **Lack of Expertise**
  A problem that occurred often was with our Git repository and how we would constantly have merge conflicts and did not know how to work it until a few weeks before the due date. This was attributed to our lack of understanding and skill. It also showed in our code through the lack of inheritance and interface classes.

What to do for next time:

- **Improve on our flaws**
  We felt as we understood the need for an inheritance/interface class near the end of the project but due to time constraints we felt that it was safer to not implement as it would mean a complete redesign of our code.

Effort and Statement of Agreed Contribution

We tended to do work only when we were together which would be an estimated 180 hours combined (90 each). We, Callum Whitehead and Harry Ellis, both agree to have done an equal 50/50 split in project contribution as we spent the same amount of time on each stage of the project.