Reuben Schoonbee - 13357179
Isaac Steele - 45913792

Our application ‘Sports Tournament’ consists of four packages. The main package contains the classes which constitute the core game logic, the GUI package contains the Swing Window classes which make up the front end or ‘view’ of the application, the test package contains the JUnit testing for the classes within the main package, and the UI package contains the code which allows the application to function as a command line application.

Within the main package, are the individual classes for each aspect of the game which together function as the ‘model’ or the ‘backend’ of our application, as well as the Game Environment class which acts as the controller of the application. The Game Environment class effectively runs the whole application, it accesses and manipulates the data from the other classes in the main package. It is also the medium through which the user interface interacts with the backend of the application, each GUI class has reference to the Game Environment class, and simply calls its methods and variables so that it does not have to interact with the backend directly. This use of encapsulation helped when implementing the GUI, and also made testing much more efficient.

To improve the cohesion of our application we used inheritance and interfacing. For example, the club class which represents the players own team extends the team class which represents a generic team (including opposing teams). By enabling the club class to inherit all the methods and variables of the parent team class, it allowed the club object to act as a generalised team object with all the required functionality when facing other teams, while also possessing the specialised functionality required to act as a club. For interfacing, we designed an abstract class called screen, which every other screen in the GUI implemented. This screen class contains concrete variables and methods for opening and closing windows, as well as a constructor of which some screen classes had their own implementation. This use of abstraction with methods and variables allowed for a much more efficient design, with less unnecessary code. Additionally the use of polymorphism with the constructor methods allowed us to guarantee a consistent performance within each screen while also implementing specific functionality for each.

Unit test coverage reflects the amount of code that is checked by your unit tests. It is best to aim for 100% coverage as that means all possible methods and branches have been accounted for

in your unit tests. Our overall coverage was 38%, as unit tests were only written for the classes in the sportstournament.main package (excluding GameEnvironment). These classes all had over 90% coverage, indicating that the unit tests covered the majority of the attributes and methods contained within each class. Where the coverage was lower would have been due to missed getters/setters and possible error cases. For GameEnvironment and the classes in the sportstournament.gui package a thorough plan was made to ensure all key functionalities were working as expected and that no bugs occurred. This involved running and playing the game multiple times, each time testing different scenarios and possible cases where bugs were more likely to occur.

Isaac Steele : The whole process of the project has developed my skills as a software engineer immensely. It has been my first real experience working with a large amount of code and integrating it all together. It really helped my understanding of Java and all the different techniques and designs used in software projects. Although it was hard and frustrating at times, I enjoyed it and am looking forward to learning even more in the next project.

Reuben Schoonbee: Somewhere during this project everything clicked for me, having the opportunity to use and understand how all the different techniques we had learned worked together gave me a much greater understanding of software development. I would feel much more comfortable approaching a large project in the future. Next time however, I would definitely commit more time to implementing the application step by step in a more structured and well planned manner, as the time this saves is much greater than the time it takes.

All in all, the two of us have worked extremely well together with the both of us contributing approximately 70 hours each into the project. We spent most of the time working on the project with one another as that allowed for clear communication and effective teamwork. Due to this, the implementation of our project went smoothly and it slowly built up to the working application it is now. However, a downside was our poor planning and time management to start off with. This led to us making many changes to our initial design and having to work more hours in the latter weeks. Next time, we both agree that we should put a lot more time into planning and thinking about how the project will work, rather than diving in head first to the implementation
.

Agreed Contributions: Isaac Steele - 50%, Reuben Schoonbee - 50%