# SENG201 Sports Game Project Report

David Williamson (#32580309) & Jackson Reilly (#91682484)

During the course of working on our SENG201 project, we have learned a great deal about the software development process, particularly the importance of thorough planning and making good design decisions early.

We began development by creating basic classes within a few distinct categories. The various types of athletes, items, and random events are all implemented as extensions of their respective abstract superclasses; for example, the abstract class “Athlete” defines the basic properties of all athletes – their name, attack, defense, health, and so on. All individual Athlete objects are then generated as one of three specialised subclasses, Attacker, Defender, and AllRounder, which each provide unique bonuses to specific stats when constructed, and may also receive extra bonuses from stat boosting items depending on their type.

The classes RandomEvent and Item are constructed similarly, with abstract methods (notably “activate” and “use” respectively) that are then implemented by their subclasses, greatly simplifying their integration into the overall game environment, as most code can then be generalised without concern for the specific type of item or event. Some types of items require additional input from the player, ie. selecting which of an athlete’s stats to increase. These are implemented as extensions of the abstract class InteractiveItem, which itself extends Item; InteractiveItem implements the “use” method and prompts the player for input, which is then passed to methods implemented differently by each subclass depending on the item’s intended effects.

The actual “environment” that the player navigates is implemented through the Shop, Club, and Stadium classes, which each handle methods and variables related to that location’s features; for instance, the Shop class stores arrays of athletes and items for sale, and implements methods for processing the buying and selling of these. Athlete and Item objects both implement the Purchasable interface, simplifying the process of displaying and selling them.

Perhaps the most key class in our project is Team and its extensions, PlayerTeam and OpposingTeam. A single instance of PlayerTeam is generated during initial setup to represent the player, and this object keeps a record of everything they own and collect throughout the game. It inherits from the abstract superclass a fixed-length array of Athlete objects that represent the team’s active members, and extends this functionality with a second array containing the player’s reserve Athletes, as well as an array list of Items representing the player’s inventory, and other key variables such as their money and points. It is also responsible for storing the Random object that is used for most random number generation in the game, a seed for which may optionally be entered by the player during initial setup.

OpposingTeam has limited functionality by comparison, as it simply handles the opponents that the player competes against, which have no need of reserve members or player-specific variables. It includes a method to generate Athletes with higher stats based on the game difficulty, as well as money and points variables representing the rewards if they are defeated by the player.

When the player chooses to compete in a match via the Stadium, a new Match object is created, and a PlayerTeam and OpposingTeam are passed to it. The Match runs comparisons between the statistics of the Athletes in the same-indexed slots of both Team’s “inTeam” arrays, and generates strings describing the outcome which are then passed to the UI for display. If the PlayerTeam’s Athletes win enough comparisons, the player wins the match, and a method of OpposingTeam is called which adds money and points to the PlayerTeam.

We have achieved a unit test coverage rate of 85% for the main package containing the game code, excluding the GUI classes. The code that is not covered by unit tests primarily consists of interactive elements such as calls to the user interface, which was tested manually to ensure it functioned as expected. Also not covered were additional minor elements such as methods that were unable to be fully implemented due to time constraints.

The most significant problem faced during development, besides time limitations, was the lack of modularity present in our initial code. We had begun developing the program purely as a command line application, and as a result, printing to and scanning from the command line interface was baked into our game logic rather than being kept separate. This caused issues and delays when the time came to implement a GUI, as methods that had previously worked as intended on the command line needed to be entirely rewritten to return their intended output instead of printing it directly, and any code that relied on reading user input via the Scanner also needed to be thoroughly reworked. The time dedicated to making such extensive modifications to the game logic subsequently ate into the time available for actual GUI programming, exacerbated further by the need to write new unit tests for the majority of the affected code. As a result, the game’s final GUI lacks polish and may not be user-friendly in some areas even if critical functionality does work as expected.

Many elements of the code also suffer from a lack of efficiency which, while it may have minimal effects on such a small program, would be debilitating on a larger scale. Due to the previously mentioned difficulties and the limited time available to work on the project as a whole, refactoring and optimisation were considered lower-priority tasks and did not receive the attention that they may have needed.

While these issues were demoralising and affected the overall quality of our final project, they also demonstrated the importance of proper planning and forethought during the software development process. This has taught us key lessons for future projects, particularly the value of code modularity with regards to streamlining development and making modifications later in the process, as well as the need for regular and repeated testing to catch bugs as early as possible.

The hours of work put into the project by each team member, as well as our agreed-upon contribution percentages, are as follows:

| **Team Member** | **Total Work Hours** | **Contribution Percentage** |
|---|---|---|
| David Williamson | 90 | 50% |
| Jackson Reilly | 90 | 50% |