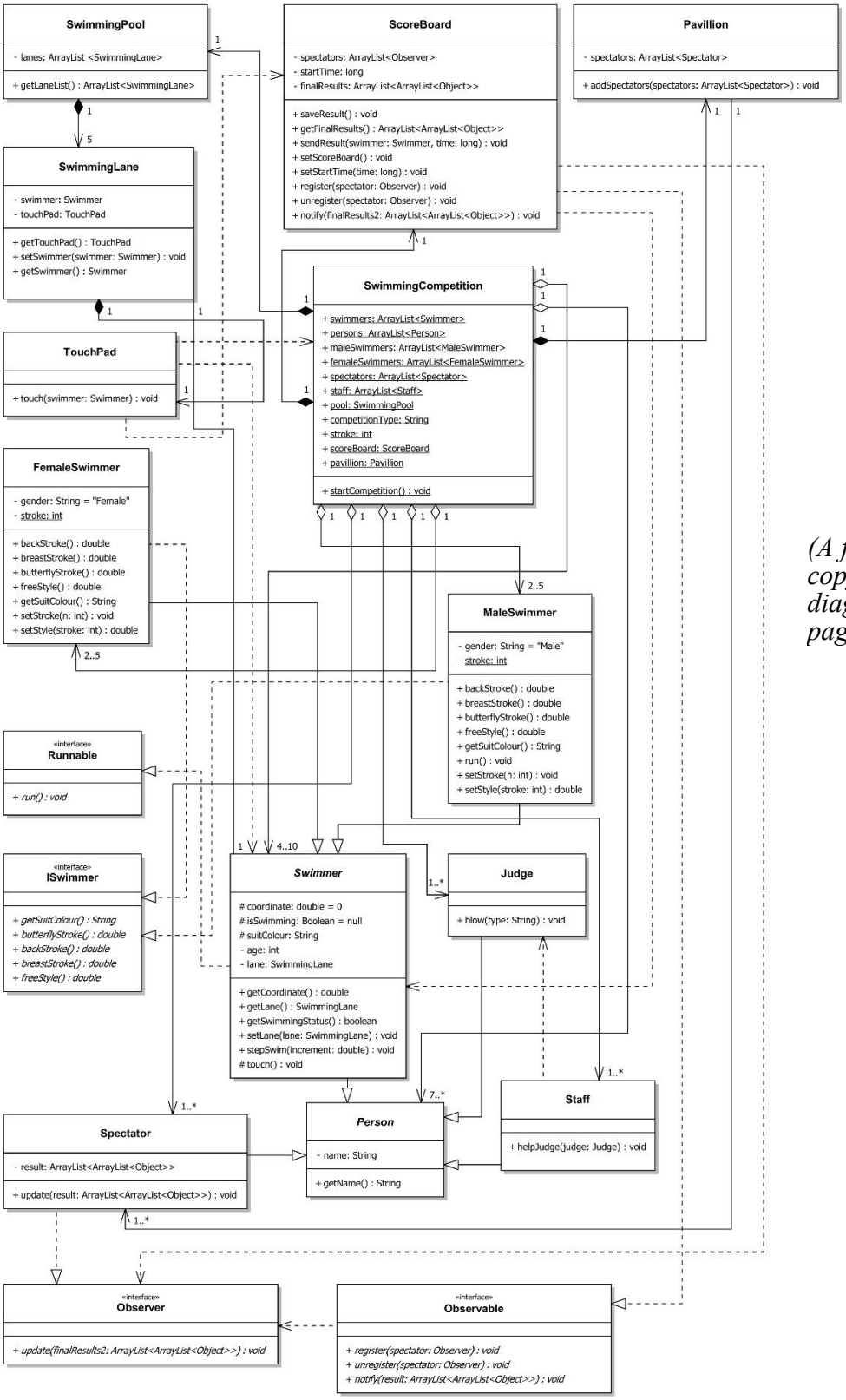
CS2012 Principles of Object Oriented Programming – Semester Project Report

Name – Shane Wolff
Index No. – 140701M

Class Diagram of the Model Part of the MVC Architecture



(A fresh and more clear copy of the class diagram is on the last page.)

Implementation of Fundamental OOP Principles

(Examples code lines are given in brackets with bold italics)

1. **Inheritance** *(Allows a child class to inherit characteristics of the extended parent classes and implemented interfaces.)*

   Simulation has various types of people. Those people have a set of attributes and methods which are common across the simulation. Those common attributes and methods have been inherited from super classes hierarchically. And also multiple inheritance is achieved by implementing interfaces. Following are some examples of them.

   - All the people are extended by the *Person* class. ***(Judge.java – 6, Spectator.java – 7, Staff.java – 4, Swimmer.java – 4)***
   - Swimmers are extended by *Swimmer* class. ***(MaleSwimmer.java – 4, FemaleSwimmer.java – 4)***
   - *Swimmer* class implements *Iswimmer* interface. ***(MaleSwimmer.java – 4, FemaleSwimmer.java – 4)***
   - *Spectator* and *ScoreBoard* classes implement *Observer* and *Observable* interfaces respectively. ***(Spectator.java – 7, ScoreBoard.java – 10)***

2. **Abstraction** *(Abstract out relevant data by hiding what is irrelevant. Can be implemented by using Abstract Classes)*

   Person is the super class of all the people in the simulation but it does not instantiate person objects, rather it contains abstract data relevant for people. Therefore *Person* class is declared abstract. *Swimmer* class is also made abstract since male swimmers and female swimmers are instantiated and not swimmers.

   - Abstract class – *Person* ***(Person.java – 4)***
   - Abstract class – *Swimmer* ***(Swimmer.java – 4)***

3. **Polymorphism** *(Ability of an object to take many forms.)*

   In *SwimmingCompetition* class, ArrayLists are defined to keep a collection of objects created.

   - All the people in the simulation are extended from the *Person* class and therefore all types of people can be stored in a Person type ArrayList. ***(SwimmingCompetition.java – 7, Person.java – 10)***
   - All the swimmers are extended from the *Swimmer* class, hence all types of male and female swimmers can be stored in an ArrayList of type Swimmer. ***(SwimmingCompetition.java – 8, Swimmer.java – 16)***

   A specific object can be passed as a parameter to a method which requires a more generic type.

   - Swimmer invokes touch method ***(MaleSwimmer.java – 31 and FemaleSwimmer.java – 28)*** which invokes a method in respective *TouchPad* classes ***(Swimmer.java – 26)***. Touch method in the *TouchPad* class requires a generic Swimmer type object ***(TouchPad.java – 6)***. Then ScoreBoard object receives data of Swimmer objects ***(ScoreBoard.java – 52)***. But actually there are no swimmer objects instead the methods are activated by the touch of male or female swimmers which is a more specific type of swimmers.

   Benefit of casting

   - Starting and finishing times are abstracted from system time which is of long type. At the time of presenting elapsed time, they are casted into float types. ***(ScoreBoard.java – 55)***

4. **Encapsulation** *(Conceal data from external of the classes)*

   Most of the class attributes are declared private. Hence they are not visible to other classes. Getters and setters are provided to access them. Actual implementation to the data is hidden from the user.

   - Most of the classes in the model have private attributes with relevant getters and setters. ***E.g. (ScoreBoard.java – 11, 12, 13, 47, 66)***

Some fields are set final to avoid further changes being made and to declare constant fields.

- Gender of the male and female swimmers are set final since they don't change after instantiation. ***(MaleSwimmer.java – 6, FemaleSwimme.java – 6)***

Hiding method implementations

- When user clicks on the "Whistle" button, the button invokes a method of *SwimmingCompetition* called startCompetition ***(GUI_2 – 278, SwimmingCompetiton.java – 35***). *GUI* and *SwimmingCompetition* classes have no notion of how swimmers are swimming. They just invokes methods. Then inside the startCompetiton method, *Judge* class blow method is again invoked ***(SwimmingCompetition – 37)***. *Judge* class is responsible to invoke further implementation of the swimming methods ***(Judge.java – 18)***.

Implementations of Concurrency Controls using Multiple Threads

(Examples code lines are given in brackets with bold italics)

Each swimmers must be swum concurrently. Therefore to achieve that, swimming methods are declared with threads. In order to create threads, *Swimmer* class is implemented with *Runnable* interface ***(Swimmer.java – 4)***. Now all male and female swimmers can declare threads in their own respective classes. (Note that *Swimmer* class cannot be extended from Thread class since java does not allow multiple inheritance.)

Movement is achieved by incrementing the *x* coordinate attribute ***(Swimmer.java – 41)***. It is incremented randomly defined by the selected stroke and gender type of the competition. Thus male and female swimmers swim differently in gender wise and stroke wise ***(MaleSwimmer.java – 37 to 76, FemaleSwimmer.java – 34 to 73)***.

Threads are declared in run methods of Male and Female Swimmers. They update the coordinate as specified by the random value ***(MaleSwimmer.java – 17 to 33, FemaleSwimmer.java – 17 to 30)***.

When *SwimmingCompetition* invoke the startCompetition method, it call the blow method of *Judge* which actually starts the threads belongs to each swimmer ***(SwimmingCompetition.java – 35, 37 and Judge.java – 18 to 40)***.

When blow method is invoked, it iteratively starts all the threads ***(Judge.java – 26 and 35)*** and the exhaustion of threads are defined inside respective thread declarations ***(MaleSwimmer.java – 19 and FemaleSwimmer.java – 19)***.

When swimmers are swimming in the model, then *GUI* class grabs the swimmers' coordinates and update in the interface. To accomplish that task, *GUI* class starts a new thread until all other swimmer threads are dead ***(GUI_2.java – 281 to 347)***. This clearly separates the function of the model and the user interface and thus preserves the MVC architecture.

When the Score board is updated with swimmer name and time after finishing the game, each data is stored in an array list for further implementation. During that invocation, it should be guaranteed that the thread should not interleave the process in the middle to avoid any mismatches between swimmer data. This is handled by declaring the method by synchronized key word ***(ScoreBoard.java – 52)***.

Shane Wolff (140701M)

Class Diagram of the Swimming Competition