SWEN20003 - Project 2 Reflection

On the whole, the code created for project 2B is mostly the same to the design in 2A, though there are some key changes.

A fairly simple but useful change from the UML diagram submitted in 2A is that Racer and Item have a RenderableObject superclass. This just defines the basic characteristics of an object that is rendered to the screen, namely its position and angle, as well as providing the standard getters, setters, a constructor and the standard render method. Besides this however, no additional classes were defined than those that were on the UML diagram. The way these classes interact, and as such the methods through which these interactions are defined, were subject to some changes however.

The way active Items, denoted as hazards in the sample solution, are handled is a bit different to how the sample solution does it. It seemed sensible to me that a class representing the Tomato, for example, would handle all interactions and states of that object. As such both the inactive form the Tomato when it's just sitting on the map waiting to be picked up as well as the active form in which it is a hazardous projectile are handled within the one class. I can however understand abstracting these state changes into the further, smaller classes, but felt that this solution encapsulated the entity together nicely. The way these interactions are handled specifically is given in greater depth in the documentation for the Item class and its subclasses, but essentially the Items that have active forms of existence on the map, Tomato and Oil, are updated differentially depending on the Boolean variable active, which is set to true upon activation by the player. Each of these items also has a method which invokes its effect. A Racer has a list of all Items whose effects it is currently suffering, and goes through each one applying the effect to itself (until the duration of the effect is reached, at which point the Item is removed from this list).

Something which became standard practice throughout this project was the use of abstract superclasses as centralised control mechanisms for their concrete subclasses. This involved the use of static methods such as updateItems() which was given a list of all the Items (which were also created statically with a method in Item) and called the update methods of each of the subclasses. This has its obvious benefits, such as single centralised method call to update all the items, but also its drawbacks, such as needing to define a common abstract method in the superclass which may pass arguments that are not needed by all versions of the subclass method. For example in the updateEnemies() method of Enemy, Player is passed through to each determineBehaviour() call, even though Elephant doesn't need it to determine its behaviour. I'm not sure about the efficiency implications of this, but I can imagine they'd be fairly light, and a reasonable trade off for the strong object oriented principles that such a paradigm allows. Despite all the implications and potential inefficiencies, I found this a very handy way to deal with potentially very large numbers of objects which all fall under the same superclass and delegate the necessary work to them.

The GlobalConstants interface was renamed to GlobalHelper and many of the constants it held were reallocated to the appropriate class, as many didn't really need to be global. These were then passed to those that needed them. Additionally, a few helper methods were added, such as getHypotenuse(), which calculates the distance between two points, and a method for reading in text files and returning them in an array of Strings, one line for each element.

The given code from project one had calls to static public final constants, such as Game.SCREENWIDTH from other classes. Apparently this is a breaking of encapsulation, and other ways were used to move this data between classes. Either the constant was passed down in the constructor to the new instance of the class being created, or a getter was used. This hopefully helps preserves encapsulation.

In terms of the actual design and creation process behind the project, the thing that surprised me most was how much I found myself looking to my UML diagram for guidance whn I was looking for the next step in the implementation. Just having an idea of which classes should be doing what and how they interconnect was very helpful to the overarching design process. The coding part was mostly easy, it was mainly succinctly structuring all the classes and subclasses that proved most difficult, in which the design steps taken previously proved really useful. It seems to me for future projects that any time put into the design stage most certainly pays off in the implementation stage, and experience with following this design has somewhat "enlightened" me on better design practices, which I could implement in the design stage next time instead of learning them the hard way during implementation as was done for this project.