Part III

Observer pattern:

Observer is a design pattern with the intent of defining a dependency between objects so that when object changes its state other objects are notified and given the opportunity to change their own state to correspond to the initial change. The motivation behind the observer design pattern is that many objects have relations between one another that require actions to be performed when the status of an object changes. An example of this is updating a driver’s status when they process through an order. In the case where after the order’s status is changed it is required to change the driver’s status to reflect it, it is possible to change the driver’s status in each block of code that may change the status. However, it is difficult to maintain such a system and ensure that the event is handled appropriately in every case. The solution to an issue like this is to use an observer. Whenever a change to the order’s status is made a signal is sent out which calls a receiver to check the driver’s status and update it appropriately. Examples of when an observer should be used include “When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently”, “When a change to one object requires changing others, and youdon't know how many objects need to be changed”, and “When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled”.

We used the observer pattern in implementing our button listeners. The listeners would wait until the linked button was pressed and would then run the corresponding function.

Singleton

Singleton is a design pattern with the intent of ensuring a class has one global instance that can be accessed anywhere as opposed to created multiple instances of a class. A common problem in software development is ensuring that data is created and processed appropriately. In cases where there are limited resources it is important that there is some type of management system that can prevent concurrency issues from occurring. One of the best ways to solve this issue is to use a singleton. A singleton will ensure that when a class needs a resource it does not cause issues with the resource’s allocation. There is only one resource manager (the singleton) which can prevent the creation of new managers in its constructor, further it can appropriately allocated resources since it has access to all of the requests for the resource. Examples of when a singleton should be used includes when “there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point” and “when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code”

Façade

Façade is a design pattern with the intent of creating an interface between operations and the systems that wish to perform them. An example of a façade in practice is creating a database class to handle database operations in other classes. There are many different types of databases with their own syntax, operations, and restrictions. In the event that the project changes its database engine, any code that performs low level queries must be changed and tested. Performing these changes is difficult and prone to error. A solution to this is to use a façade, by providing a unified interface to database operations it is much simple to switch database engines as there is only one place to change the low-level queries. Facades are a great way to reduce code rigidity and fragility. The application of facades in practice is when “you want to provide a simple interface to a complex subsystem.”, “there are many dependencies between clients and the implementation classes of an abstraction.”, and “you want to layer your subsystems. Use a facade to define an entry point to each subsystem level”.