Design Patterns: The Observer and Façade Pattern

1. What are Software Design Patterns?
   1. Reusable solution to a commonly occurring problem within a given context in software design
2. Behavioural Patterns
   1. Popularized in the book, design patterns by the Gang of Four
   2. The Observer pattern is an exhibit of behavioural software design patterns
3. Structural Pattern
   1. Popularized in the book
   2. The Façade pattern is an exhibit of structural software design patterns
4. The Observer Pattern
   1. Problems
      1. Many-to-one dependency relationships forces objects to be tightly together
      2. Low coupling and less dependency is desired
      3. If software change, how do we alter the behavior of objects without it having to affect others
      4. We want to edit and use classes efficiently and easily
   2. Solution
      1. Allow modularity
      2. Decouple many to one dependency relationships between a subject class, and other subclasses
      3. A main “subject” object will have many observer objects that are attached subscribed to the main object
      4. A single state change of the subject will automatically notify all observers subscribed to the subject
   3. Allows
      1. Automatic notification of the subject’s state change sent to individual observer objects allow the objects to update their state auto
      2. This is because observers are subscribed to a subject of interest
      3. Easier to modify and integrate with other components… all an effort to reduce coupling
   4. Push Model or Pull model
5. Observer Limitation
   1. If not properly implemented, will be very complexity
   2. Too many notification
   3. The pull model isolate communication in separate steps
6. The Façade Pattern
   1. Problems
      1. How do we simplify and overcome using a complicated system without leading to improper use or testing?
      2. Can we minimize coupling and dependencies between subsystems?
      3. How do we access certain parts of the software to users without being restricting all of the extra details?
   2. Solution
      1. Provides a unified, higher level simplistic interface to interact with the larger and complex subsystem interface
      2. Creates a façade between the user utilizing a system and the actual complex system of interest
      3. Defined to avoid accessing large, complex subsystem
      4. Avoid having the user know of all details prior to using the system
      5. Façade acts as a wrapper class that you would call prior to accessing additional classes
   3. Allows
      1. Promotes low coupling; objects don’t need to be aware of one another
      2. Shield clients from additional classes that are not of concern; no need to worry about low level details
   4. Drawback
      1. If the structure of the subsystem changes, the façade class may also need to be updated in respect to the new change made
      2. Can return a system into an oversimplified class