Design Patterns

March 29, 2015 12:38 PM

- Procedural code gets information and then makes decisions
- Object-Oriented code tells objects to do things

The Decorator Pattern:

- Allows us to dynamically extend the behavior of a particular object at runtime
  - Without needing to resort to unnecessary inheritance
- Use an interface for the common function
- Then use a decorator which uses a constructor injection
- Then wrap the core service with any number of decorators
- e.g. echo (new TireRotation(new OilChange(new BasicInspection())))->getCost()
  - Where BasicInspection is the interface with the method getCost()
  - Remember to use protected variables
- Be very carefully with inheritance, don't break SOLID principles

The Observer Pattern:

- Want to create a one-to-many relationship so that when one object changes its dependencies/observers are immediately notified
  - Then they can respond to the event that just occurred
- Don't want to just dependency injection; don't want objects to have awareness of one another
  - This makes it more flexible and follows the single responsibility principle
- A change in one object needs to have a nice flexible decoupled way of notifying other objects of this change
  - They in turn can respond however they need to
- Gives use a way for objects to notify one another without being intrinsically linked

The Adapter Pattern:

- An adapter allows you to translate one interface for use with another
- Wraps one interface for use with another
- Think creating wrappers to add some functionality or transitively apply a different contract/interface

The Template Method Pattern:

- A behavioural design pattern that defines the program skeleton of an algorithm in a method called template method, which defers some steps to subclasses
- You can tell when you need a template method when you're copying and pasting multiple classes that are heavily related and only changing very minute details
- Use this design principle when worried about code duplication
  - Use an abstract class for commonalities
- Also be wary of duplicated algorithms
- Make a parent abstract class this lets all subclasses call the exactly same functions
  - For unique methods use an abstract function so that subclasses must implement their own version of it
- Abstract methods let subclasses fill in the missing information

The Strategy Pattern:

- Defines the family of algorithms, encapsulates each one and makes them interchangeable
  - By leveraging polymorphism we're able to develop loosely coupled applications
    - Lets us swap various components at runtime
- Do this by coding to an interface
  - Create a set of algorithms (multiple ways to execute a specific strategy)
  - Then force them to be interchangeable by making them adhere to an interface

    - Then in the context class specify that you need to use the interface
        - This makes any set of algorithms that use the interface usable for the app

The Chain of Responsibility Pattern:

- Allows us to chain any number of object calls while giving each of these objects to end the execution and handle the request
    - If it can't handle the request it can send your request up the chain to give the next object to have the chance to do the exact same thing
- Client can make a request without knowing specifically how the request is going to be handled
- I.e. set up a linked list of all the classes whose functions need to be called and iterate through and call them
    - But do this with a parent abstract class
    - The function that needs to be called should be an abstract function in the parent class to enforce the subclasses to have the function
    - Have a function in the parent class to set the successor
        - Set the successor at runtime manually

Tell, Don’t Ask:

- It is best to tell your objects what to do rather than asking them for their state
- The client shouldn't be doing any checking, it just creates an object and 'uses' it
- Move the manual algorithm (from the client side) into a class as a function which the client calls
    - Keep moving this method from class to class until it belongs in the appropriate class (the class who is responsible for it)
        - Called intermediary functions
        - Just make a 'wrapper' function
        - e.g. elevator->checkAlarms() is actually elevator->monitor->checkAlarms()