Design Patterns

Strategy Design Pattern

Closed for modification, open for extension

Design Principle #1

- Identify the aspects of your application that vary and separate them from what stays the same
- Take the parts that vary and encapsulate them in a class
  - You can later alter or extend the parts that vary without affecting those that don't.

Design Principle #2

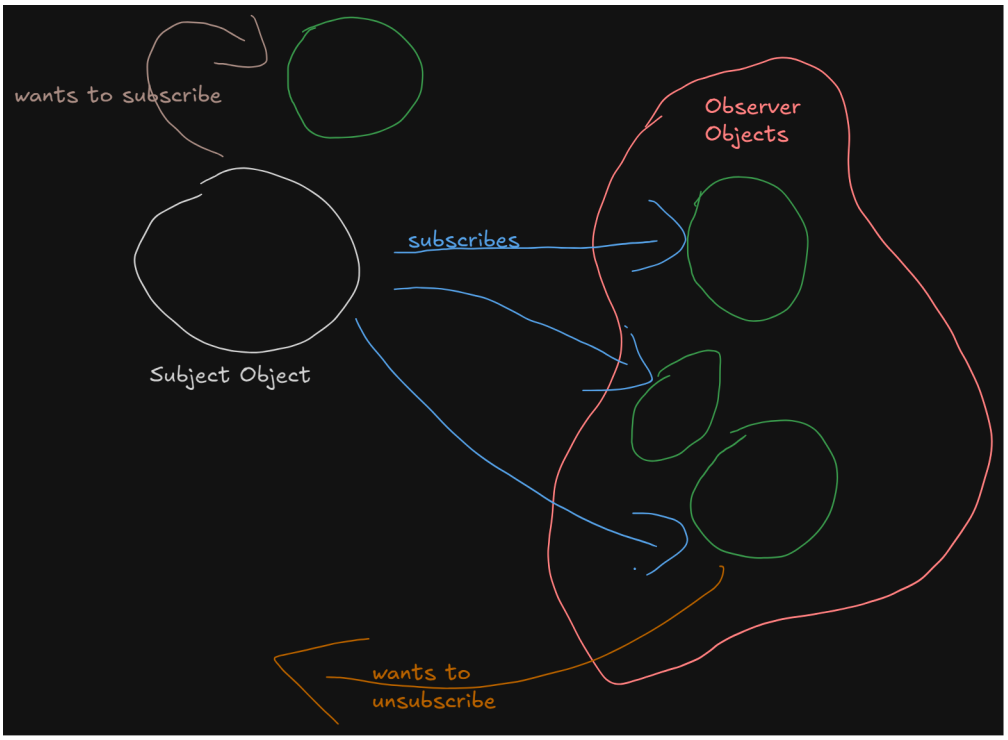
- …

Design Principle #3

- Favor composition over inheritance
- The **has a** relationship is interesting
- Creating systems using composition …

Observer Pattern

- The Subject object manages some important data
- When data in the Subject changes, the observers are notified
- The observers (Observer Objects) have subscribed to (registered with) the Subject to receive updates when the Subject's data changes
- New data values are communicated to the observers in some form when they change



- The observer can subscribe or unsubscribe to the subject
- A one to many dependency between objects
  - When one object changes state
  - all of its dependents are notified/updated automatically

Design Principle #4

- Strive for loosely coupled designs between objects that interaction

Singleton Design Pattern

- How to prevent more than one object from being instantiated?
- If we make the constructor private, an instantiation of an object can only be made inside the class
- This also allows the class to have a global point of access for the program to access
- `static` is global

```
static uniqueInstance

public static synchronized Singleton getInstance {
    if (uniqueInstance == null) {
    uniqueInstance = new Singleton();
    }
    return uniqueInstance;
}
```

Threading

- Two instances could get made if there are executions on multiple threads
- Make a method `synchronized` to make every thread wait its turn so no two threads cannot enter the method at the same time (`syncronized` disallows concurrency)
- Below code improves performance - not relevant in such a small program though

```
private volatile static Singleton uniqueInstance
private Singleton() {}

public static Singleton getInstance() {
    if (uniqueInstance == null) {
        synchronized (Singleton.class) {
            if (uniqueInstance == null) {
                uniqueInstance = new Singleton();
            }
        }
    }
    return uniqueInstance
}
```

Enums

- Using an enum to create a Singleton can help with synchronization issues

```
public enum Singleton {
    UNIQUE_INSTANCE;
    // more useful fields here
}

public class SingletonClient {
    public static void main(String[] args) {
        Singleton singleton = Singleton.UNIQUE_INSTANCE;
        // use the singleton here
    }
}
```

- constructor must be private or package private

Another Method

- make everything static in one class

Decorator

- "Decorating" a class
- "Decorates" an object
    - Like a "wrapper" for an object

Adapter

- The adapter converts one interface into another

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;
    public TUrkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    @Override
    public void quack() {
        turkey.gobble();
    }

    @Override
    public void fly() {
        for (int i = 0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

Abstract Design Pattern

- Provides an interface for creating families of related or dependent objects without specifying their concrete classes

Factory Design Pattern

- Abstract top level class
- Concrete subclasses
- Concrete subclasses for the actual thing
- Subclasses instantiated by the factory methods
- ex. Pizza store -> NY Pizza Store, California Pizza Store -> NY Cheese Pizza, NY Pepperoni Pizza, California Cheese Pizza, California Pepperoni Pizza
    - Abstract pizza store
    - You can define your own concrete stores and concrete pizzas

Abstract Factory Pattern

- The abstract factory pattern provides families of classes without concrete subclasses
- separate interfaces for each family of classes (e.g. each type of topping)