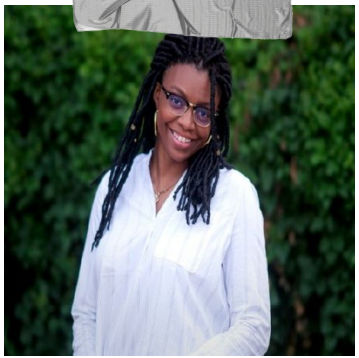
Design Patterns

Gang of Four:

By Vera Ekhator

IMG source:https://avelonpang.medium.com/gang-of-four-design-patterns-intro-e884af24b85f

Brief Intro:
My name is Vera Ekhator. I am currently a software developer in training at Zip Code Wilmington. I am in the Java 8.0 Cohort.
In this presentation, I will be covering several programming design patterns.
Stay tuned!



Creational:

- Singleton
- Builder
- Factory

1. Singleton

➔ **Appeal**

This pattern ensures that only one instance of a class can exist. It does so by means of a private constructor than can only be called if no other instance of the class currently exists.

➔ **Utility**

The singleton pattern is great for standardizing access to a resource. If you need access to the singleton instance, then you can gain it by a static call.

2. Builder

➔ **Appeal**

The builder pattern allows you to build out your object in steps. Once you're done adding the necessary components you launch the build method and create the object.

➔ **Utility**

The builder pattern allows for added flexibility when it comes to object creation.

3. Factory

➔ **Appeal**

It's a helper class with the purpose of creating objects of another class.

➔ **Utility**

Allows you to encapsulate object creation for better code maintenance.

Structural:

- Decorator
- Adaptor
- Facade

1. Decorator

➔ **Appeal**

Provides a way to reduce code redundancy by utilizing the superclass inheritance for shared attributes and methods..

➔ **Utility**

Maintenance. Since the values of attributes are shared code only has to be changed in one place to carry over to the other.

2. Adapter

➔ **Appeal**

Interface converter. Allows for broader application for code with less modifications.

➔ **Utility**

You can add functionality without having to tamper code that has already been written and tested. Rather than refactor code, you can just extend it.

3. Facade

➔ **Appeal**

Simplifies client interaction with class' library. Shows client only what they need to see.

➔ **Utility**

Helps reduce errors caused by code complexity. The underlying methods still exist and are accessible but they are not exposed.

Behavioral:

- Observer
- Strategy
- Template
- Command

1. Observer

➔ **Appeal**

Observers are connected to a “subject” and receive updates whenever the state of the subject changes.

➔ **Utility**

Since the connection is at-will, an observer can “tune-in: and “tune-out” to a subject’s state updates with relative ease. This is because the subject registers and deregisters observers at request.

2. Strategy

➔ **Appeal**

Takes a body of algorithms and externalizes them while still allowing for interchangeability.

➔ **Utility**

User can choose which strategy to implement and if they change their mind they can change their strategy since the encapsulated behavior is interchangeable.

3. Template

➔ **Appeal**

Create a super method that with varying degrees of access.

➔ **Utility**

Allows for dynamic code implementation under one method. Depending on user intent, all they need to do to access the varying degrees of method utility is add more arguments to the method call.

4. Command

➔ **Appeal**

Allows you to store “commands” to be executed at a later time. Executed commands are stored in a stack which creates command execution “memory”.

➔ **Utility**

You can undo commands by utilizing the stack .pop() function. This allows you to walk back on steps your program has taken.

Recap:

Design Patterns help keep code clean and maintainable. They emphasize good coding practices such as encapsulation, abstraction, and dependency reduction. Utilizing design patterns allows you to avoid reinventing the wheel in your code. These patterns are tried and true and provide a coding standard for developers to follow as needed.

Go forth and design good software.

Thank You!