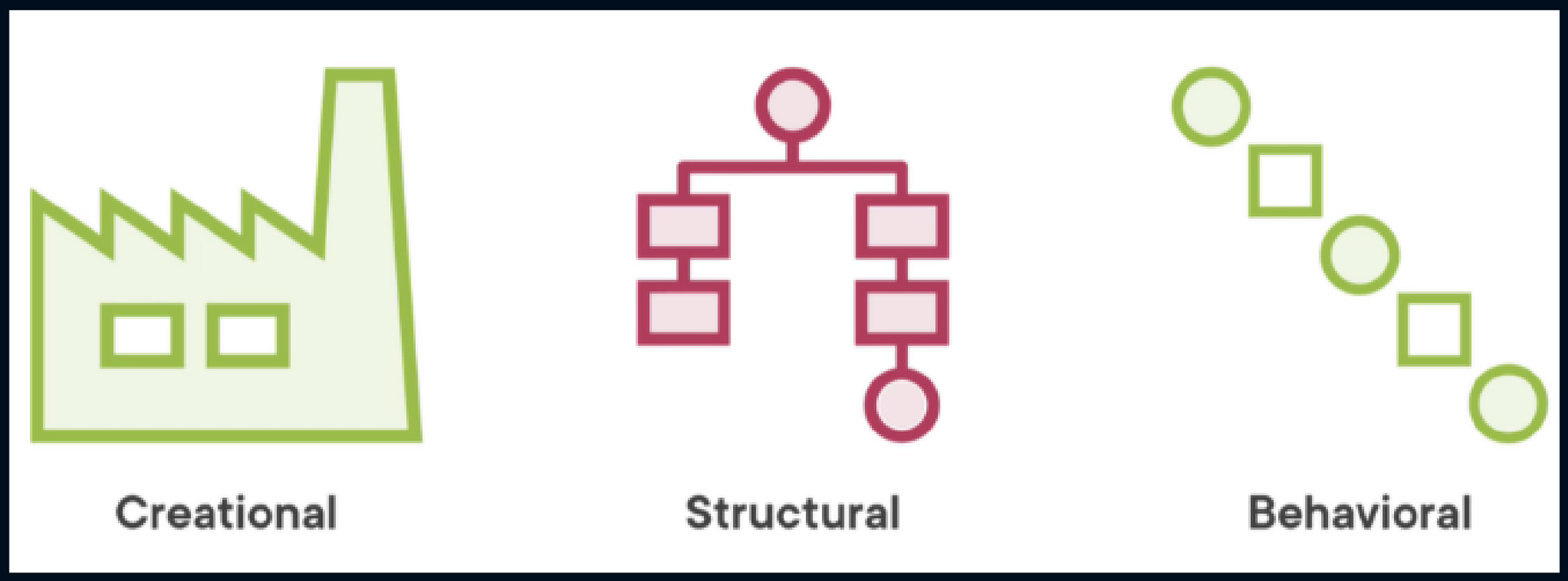
Design patterns

A pattern is a general, reusable solution to a commonly occurring problem within a given context in software design



Creational

- Abstract the object instantiation process
- Help with making your system independent of how its objects are created, composed and represented.

Structural

- Concerned with how classes and objects are composed to form larger structures

Behavioral

- Characterize complex control flow that’s difficult to follow at runtime
- Let you concentrate on the way objects are interconnected

GoG Design patterns

Most Used Design Patterns Cheat Sheet

Creational Patterns
Used to construct objects

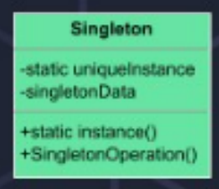
Structural Patterns
Used to form large object structures

Behavioral Patterns
Used to manage algorithms and relationships

Singleton

Use when you want to have one instances of a class.

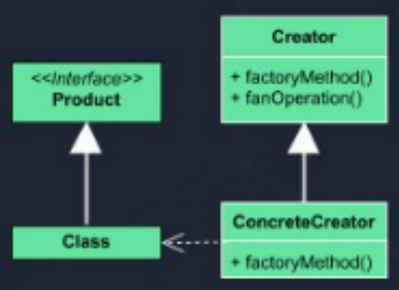
Example: logging, db connections.



Factory Method

Use when you want to delegate object creation to subclasses.

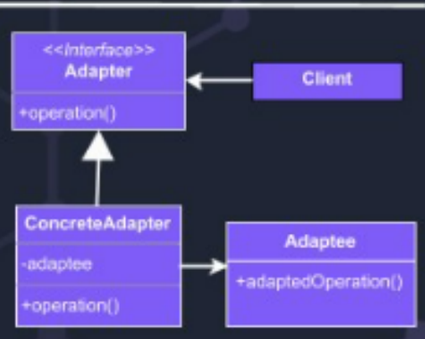
Example: create GUI component



Adapter

Use when you need to convert an interface to another interface

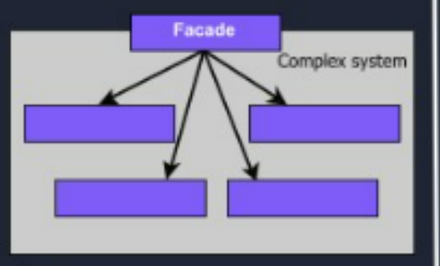
Example: make incompatible classes work toghether



Facade

Use when you want to provide a simplified interface to a complex subsystem

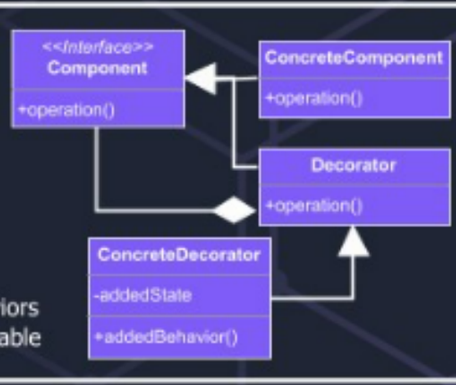
Example: Providing a simple interface to a complex subsystem



Decorator

Use when you need to wrap objects to modify their behaviors

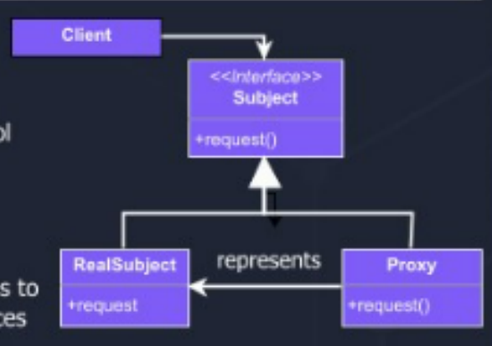
Example: make object behaviors dynamically modifiable



Proxy

Use for object access control

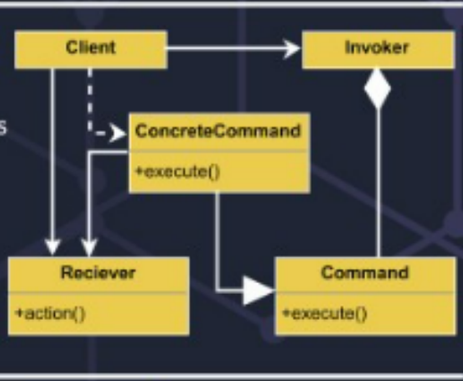
Example: Controling access to sensitive resources



Command

Use for encapsulating requests with parameters

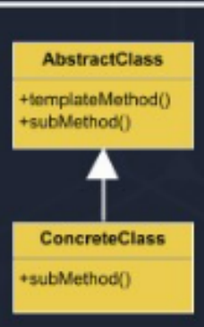
Example: Implementing operations



Template Method

Use when you want to break down an algorithm into a series of steps

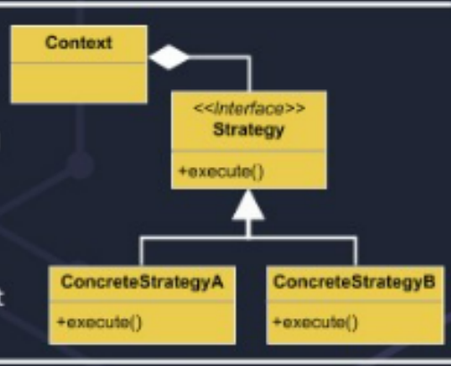
Example: Common behavior should be located in one class



Strategy

Use for interchangeable algorithms that can be swaped at runtime

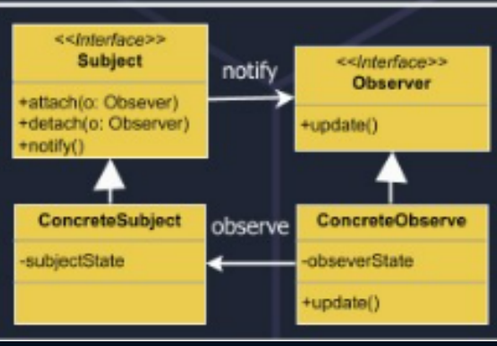
Example: Implement different sorting algorithms



Observer

Use for automatic updates of dependand objects

Example: Implement subscribers



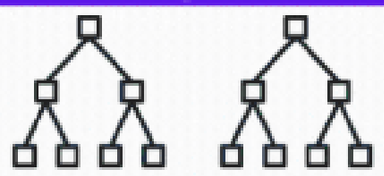
TechWorld WithMilan
simplifying complex topics

Design patterns continuation

Command

Encapsulates a request as an object, allowing parameterization and queuing.

Composite



Composes objects into tree structures to represent part-whole hierarchies.

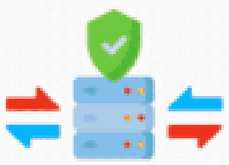
Iterator

Provides a way to access elements of an aggregate object sequentially.

Decorator

Attaches additional responsibilities to objects dynamically, enhancing flexibility.

Proxy

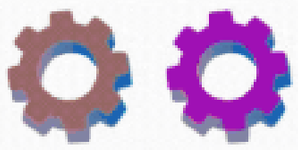


Controls access to an object by acting as an intermediary.

Facade

Provides a simplified interface to a set of interfaces in a subsystem.

Visitor



Represents an operation to be performed on elements of an object structure.

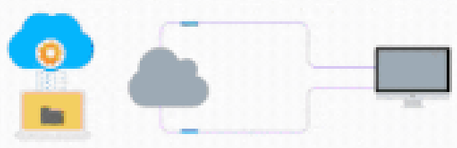
Adapter

Allows the interface of an existing class to be used with another.

Bridge

Separates abstraction from implementation, allowing them to vary independently.

Flyweight



Minimizes memory usage or computational expenses by sharing as much as possible.

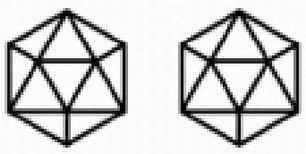
Chain of Responsibility

Passes a request along a chain of handlers, each processing or forwarding it.

Observer

Defines a one-to-many dependency between objects so that when one changes, all its dependents are notified.

Builder



Separates the construction of a complex object from its representation.

State

Allows an object to alter its behavior when its internal state changes.

Factory

Defines an interface for creating objects, but lets subclasses alter types.

Singleton

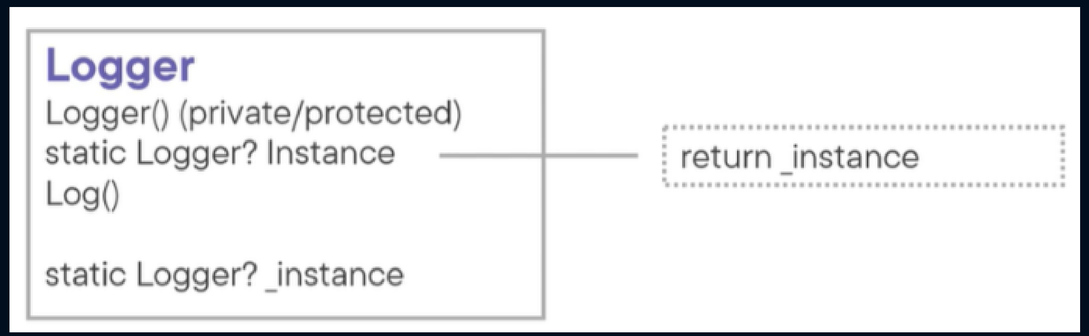
The intent of the singleton pattern is to ensure that a class only has one instance, and to provide a global point of access to it.

Use Cases

- When there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Observations

- Strict control over how and when clients access it
- Avoids polluting the namespace with global variables
- Subclassing allows configuring the application with an instance of the class you need at runtime
- Violates the single responsibility principle



Analysis

- You must have a private field which data type is the same as your class, this is going to be returned whenever a "new" instance is being created.
- The constructor must be private, so another method can be in charge of creating the object
- Instantiation only occurs one time, this is when this private field described before, is null.

Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

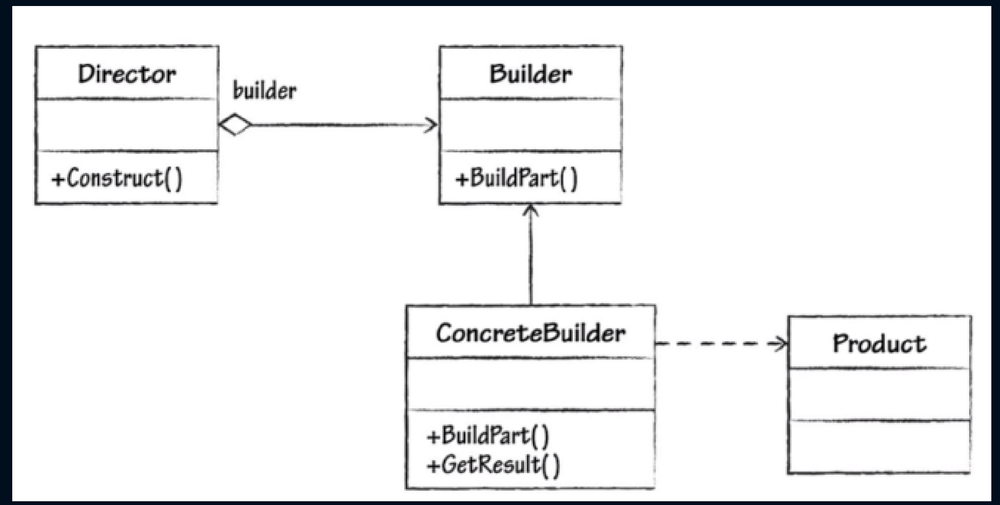
When piecewise object construction is complicated, the builder provides an API for doing it succinctly.

Use Cases

- Whn you want to make the algorithm for creating a complex object independent of the parts that make up the object and how they are assembled
- When you want the construction process to allow different representations for the object that's constructed

Observations

- It lets us vary a product's internal representation
- It isolates code for construction and representation; ti this improves modularity by encapsulating the way a complex object is constructed and represented
- It gives us a finer control over the construction process
- Complexity of your code base increases



Analysis

- Builder => interface that defines how the pieces of the complex object are created
- Concrete builder => construct the complex object according to the builder blueprint. Each concrete builder is in charge of keeping track of the representation object that it creates and retrieving that object when queried
- Director => manage the call to create the complex class by the concrete builder

Decorator

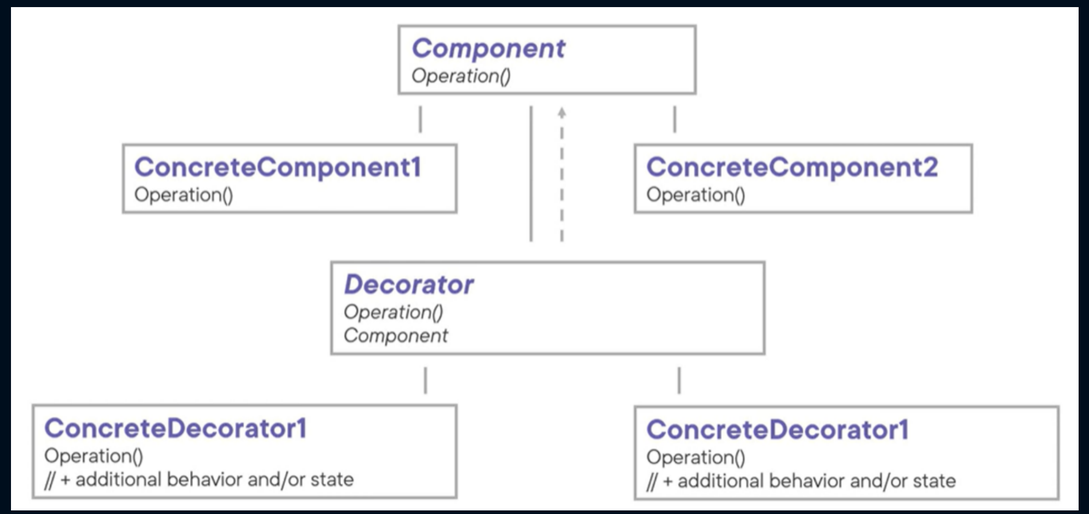
The intent of this pattern is to attach additional responsibilities to an object dynamically. A decorator thus provides a flexible alternative to subclassing for extending functionality.

Use Cases

- When you have a need to add responsibilities to individual objects dynamically (at runtime) without affecting other objects.
- When you need to be able to withdraw responsibilities you attached to an object
- When extension by subclassing is impractical or impossible

Observations

- More flexible that using static inheritance via subclassing: responsibilities can be added and removed at runtime ad hoc.
- You can use the pattern to split feature-loaded classes until there's just one responsibility left per class
- Increased effort is required to learn the system due to the amount of small, simple classes



Analysis

- Component => defines the interface for objects that can have responsibilities added to them dynamically
- Concrete Component => defines an object to which additional responsibilities can be attached
- Decorator => maintains a reference to the Component, and defines an interface that conforms the Component's interface
- Concrete decorator => adds responsibilities to the component

Observer

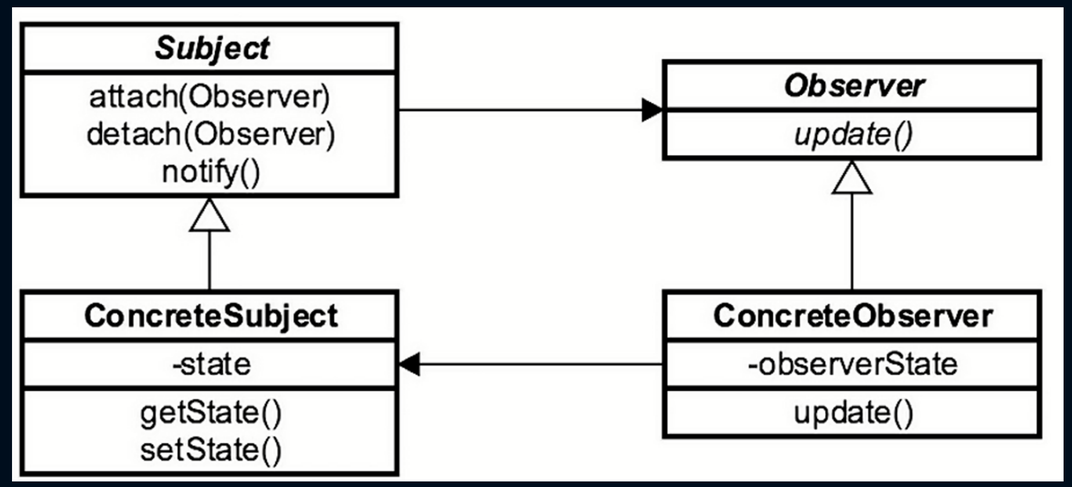
The intent of this pattern is to define a one to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Use Cases.

- When a change to one object requires changing others, and you don't know in advance how many objects need to be changed
- When objects that observe others are not necessarily doing that for the total amount of time the application runs
- When an object should be able to notify other objects without making assumptions about who those objects are

Observations

- It allows subjects and observers to vary independently: subclasses can be added and change without having to change others
- Subject and observer are loosely coupled
- It can lead to a cascade of unexpected updates



Analysis

- Observer => defines an updating interface for objects that should be notified of changes in a Subject
- Subject => knows its Observers. Provides an interface for attaching and detaching them.
- ConcreteObserver => store state that must remain consists with the Subjects' state. They implement the Observer updating interface to keep state consistent.
- ConcreteSubject => stores state of interest to ConcreteObserver objects, and sends a notification to its Observers when its state changes

Recommendations

- View design patterns as a template to start from
- Each pattern has an intent, learn which problem a pattern solves
- Don’t learn the pattern implementation from the top of your head
- Favor object composition over class inheritance
- When you need to provide some basic functionality that can be overridden you can consider using abstract classes

Demo

We will check the following:

Singleton pattern

Builder pattern

Decorator pattern

Observer pattern

You can check the following Repository for some examples:

C# fundamentals

Task

Quick Exercises

Using the implementation of the builder pattern in C# fundamentals repository (in the Design patterns project) turn that implementation into a Stepwise builder, where you need to define the Name first, then the Description and finally the Price.