Friday, 1 April 2022

Java Design Patterns & SOLID Design Principles

Structural Design Patterns

INTRODUCTION

What Are They?

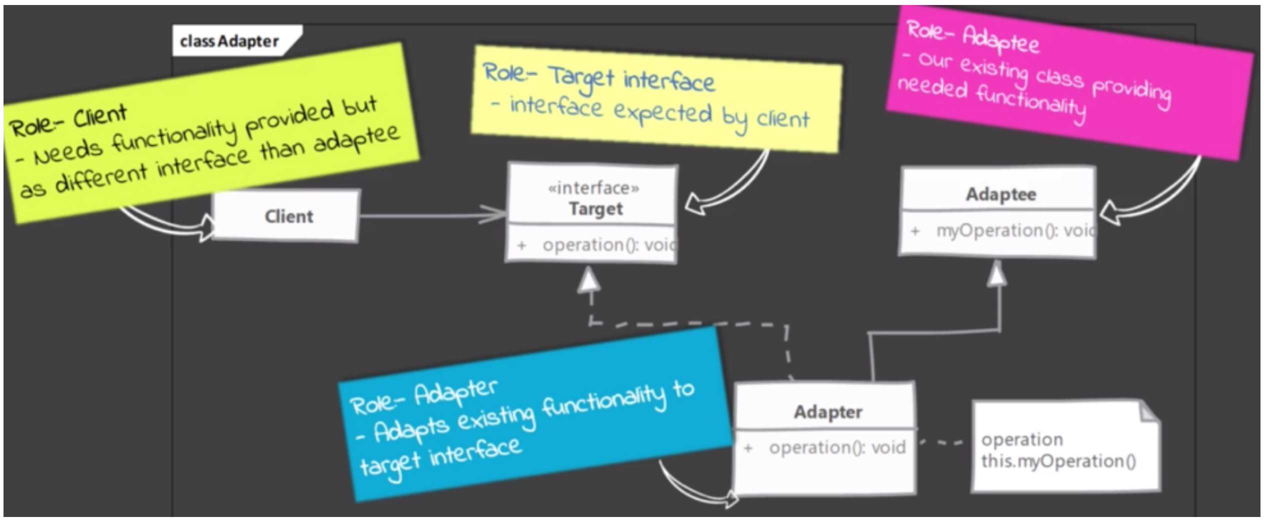
- Structural patterns deal with how classes and objects are arranged or composed (using composition and/or inheritance).
- The following patterns are considered structural:
  - Adapter
  - Bridge
  - Decorator
  - Composite
  - Facade
  - Flyweight
  - Proxy

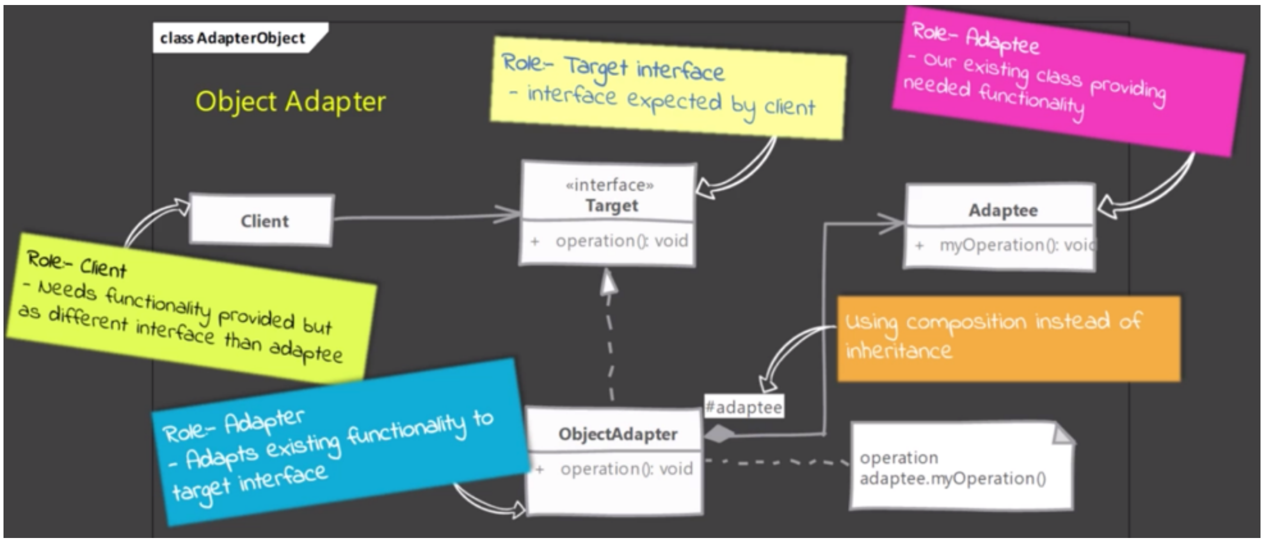
ADAPTER/WRAPPER

What Is It?

- A structural design pattern that is used when you have an existing object which provides the functionality that client needs but the client code can't use this object as it expects an object with different interface.
- Using the adapter pattern, you make the existing object work with client by adapting the object to client's expected interface.
- There are two types of adapters:
  - Class (Two way) adapter
  - Object adapter

Friday, 1 April 2022


class Adapter
Role- Client
- Needs functionality provided but as different interface than adaptee
Role- Target interface
- interface expected by client
Role- Adaptee
- our existing class providing needed functionality
Client
«interface»
Target
+ operation(): void
Adaptee
+ myOperation(): void
Role- Adapter
- Adapts existing functionality to target interface
Adapter
+ operation(): void
operation
this.myOperation()

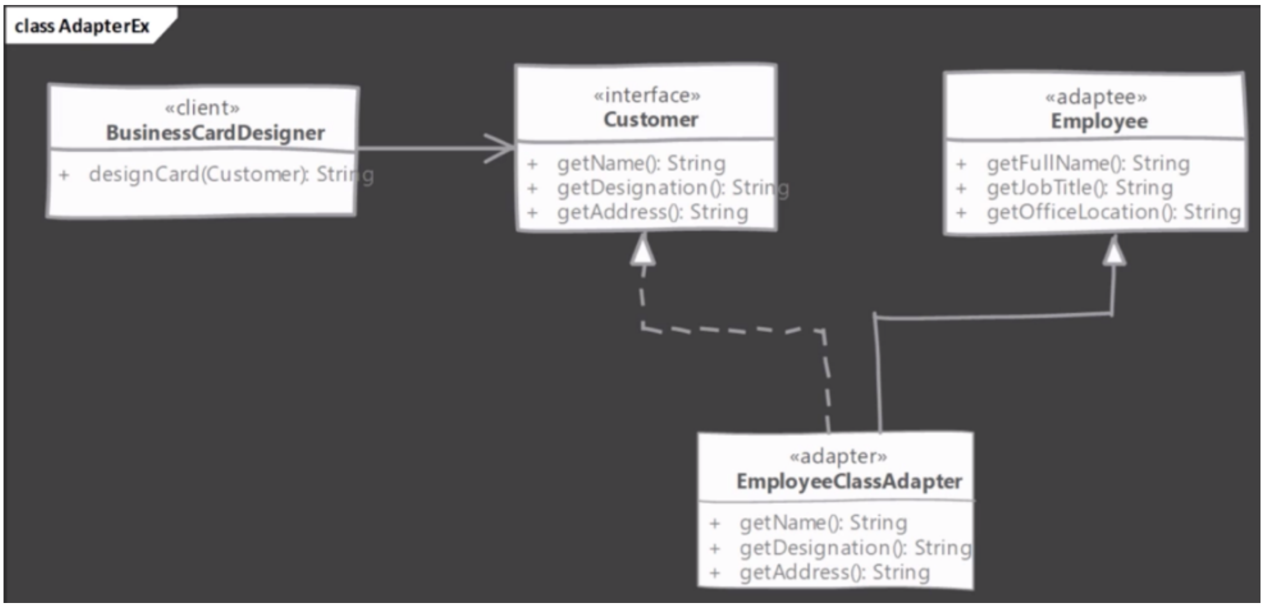

class AdapterObject
Object Adapter
Role- Target interface
- interface expected by client
Role- Adaptee
- our existing class providing needed functionality
Client
«interface»
Target
+ operation(): void
Adaptee
+ myOperation(): void
Role- Client
- Needs functionality provided but as different interface than adaptee
using composition instead of inheritance
Role- Adapter
- Adapts existing functionality to target interface
#adaptee
ObjectAdapter
+ operation(): void
operation
adaptee.myOperation()

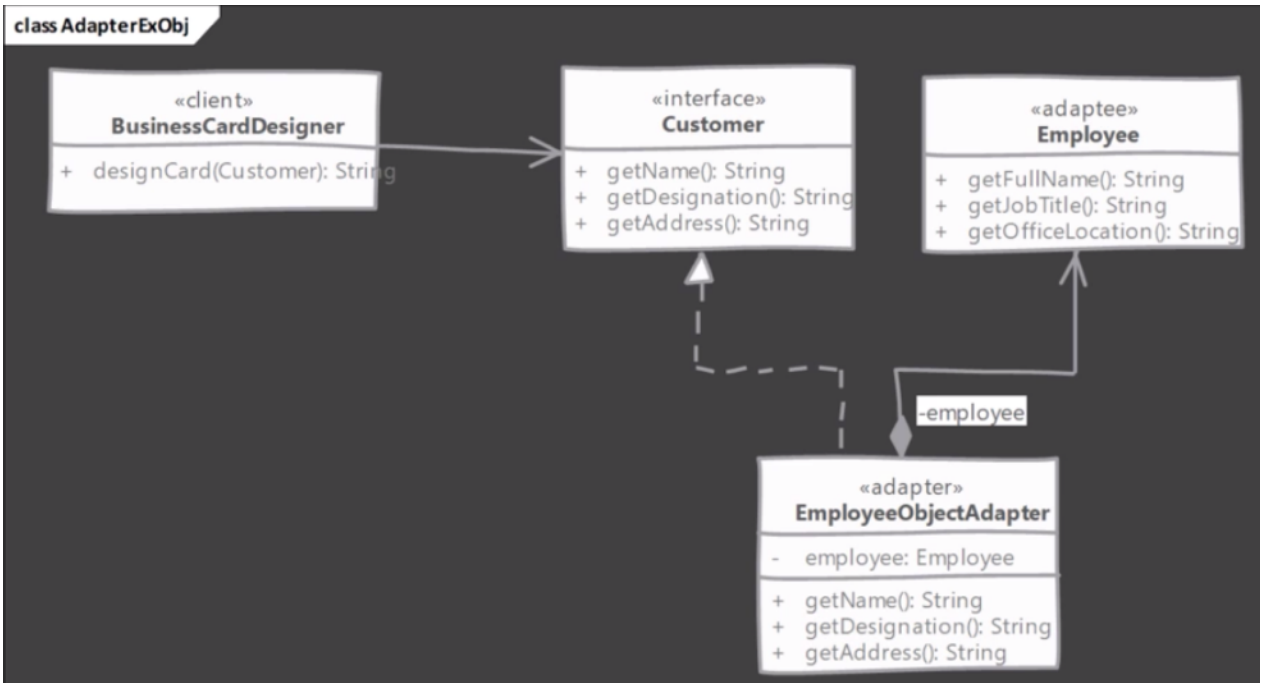
What Are Adapter Implementation Steps?

- For class adapter:
  - Start by creating a class for Adapter.
  - It must implement the target interface expected by client.
  - It must extend from the existing adaptee class (make use of **inheritance**).
  - Simply forward the method to another method inherited from adaptee class.

- For object adapter:
  - Start by creating a class for Adapter.
  - Adapter must implement the target interface expected by a client.
  - Accept adaptee as constructor argument in adapter (make use of **composition**).
  - A preferred way is to take adaptee as an argument in constructor; less preferred way is instantiating it in the constructor (tightly coupling with a specific adaptee).

Example UML Diagrams:





Adapter Implementation & Design Considerations:

- How much work the adapter does depends upon the differences between target interface and object being adapted. If method arguments are same or similar adapter has very less work to do.
- Using class adapter “allows” you to override some of the adaptee’s behaviour, but this has to be **avoided** as you end up with adapter that behaves differently than adaptee. Fixing defects is not easy anymore.
- Using object adapter allows you to potentially change the adaptee object to one of its **subclasses**.
- In Java, a “class adapter” may not be possible if both target and adaptee are **concrete classes**. In such cases, the object adapter is the only solution. Also since there is no private inheritance in Java, it’s better to stick with object adapter.
- A class adapter is also called as a two way adapter, since it can stand for both the target interface and for the adaptee - that is we can use object of adapter where either target

Adapter vs. Decorator:

- **Adapter** simply adapts an object to another interface without changing behaviour **vs**. **decorator** enhances object behaviour without changing its interface.
- With **adapter** it is not easy to use recursive composition, that is, an adapter adapting another adapter since adapters change interface **vs.** with decorators, we can do recursive composition (decorate a decorator) since decorators do not change the interface.

Adapter Pitfalls:

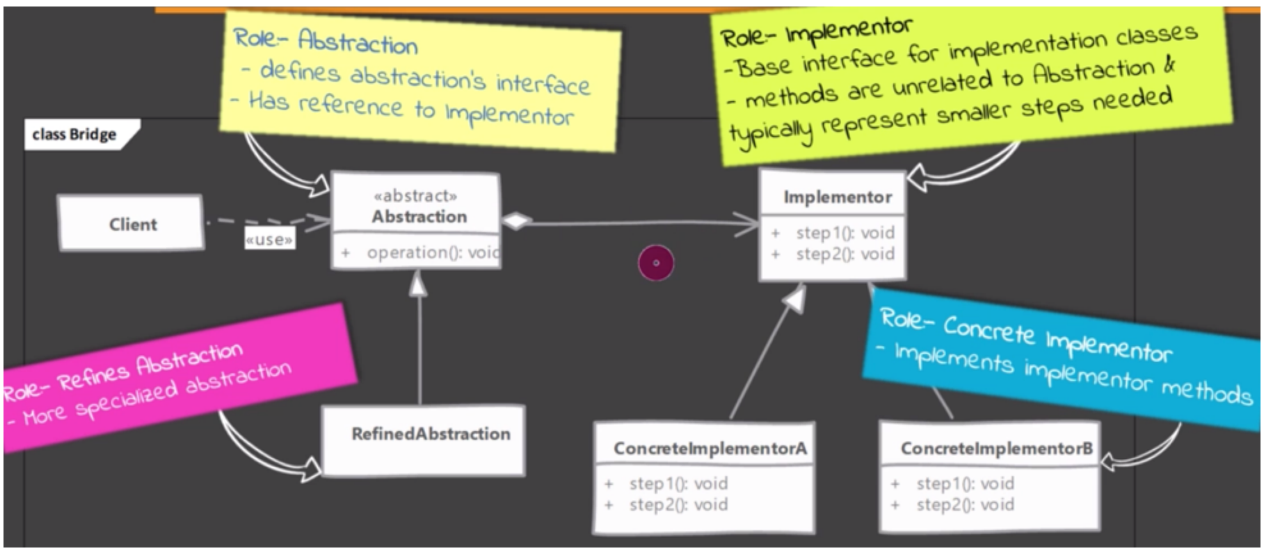
- Class adapter unnecessarily exposes unrelated methods in parts of your code through inheritance -> **avoid class adapters**.
- It is tempting to alter behaviour in your adapter besides simple interface translation but avoid doing that.

Friday, 1 April 2022

BRIDGE

What Is It?

- Implementations and abstractions are generally tightly coupled to each other in normal inheritance.
- **Bridge** is a structural design pattern used to decouple the implementations and abstractions so that they can both change without affecting each other.
- You achieve this feat by creating two separate inheritance hierarchies - one for implementation and another for abstraction - and use **composition** to bridge these two hierarchies.
- Abstraction and Implementor are connected using composition while having their own hierarchies defined.
- Note that the abstraction may not be **abstract** - here it is used in a sense of abstracting functionality from the client code.

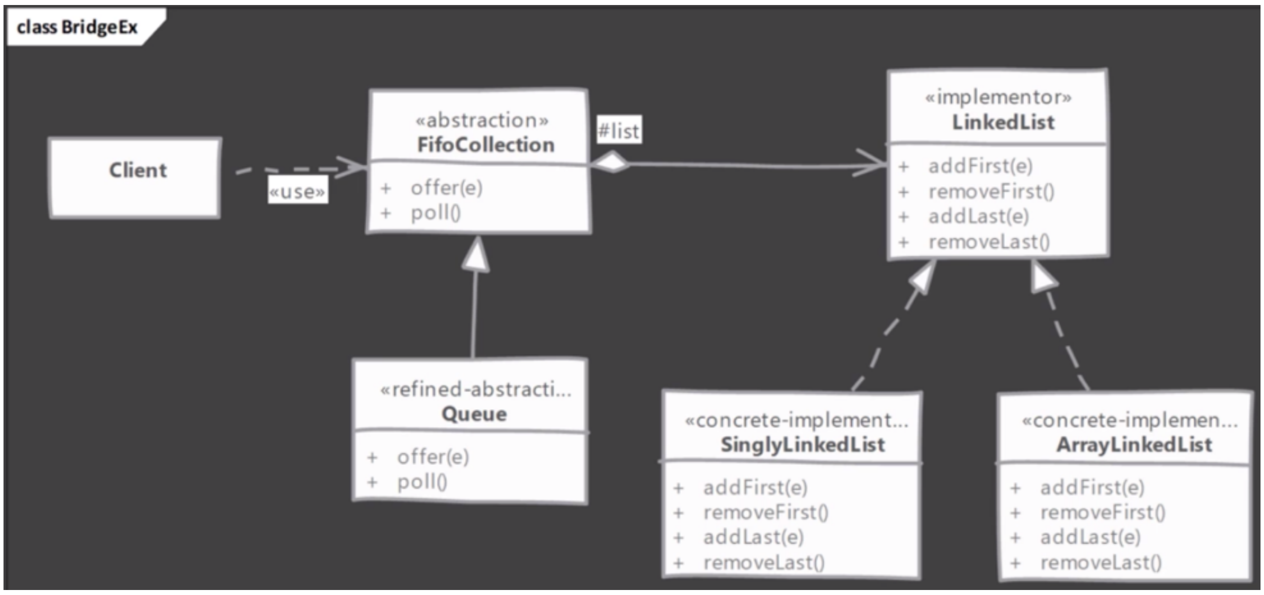


Role- Abstraction
- defines abstraction's interface
- Has reference to Implementor
Role- Implementor
-Base interface for implementation classes
- methods are unrelated to Abstraction & typically represent smaller steps needed
class Bridge
Client
«use»
«abstract» Abstraction
+ operation(): void
Implementor
+ step1(): void
+ step2(): void
Role- Refines Abstraction
- More specialized abstraction
RefinedAbstraction
Role- Concrete Implementor
- Implements implementor methods
ConcreteImplementorA
+ step1(): void
+ step2(): void
ConcreteImplementorB
+ step1(): void
+ step2(): void

What Are Bridge Implementation Steps?

- You start by defining the abstraction as needed by client:
  - Determine common base operations and define them in abstraction.
- Optionally, define a refined abstraction & provide more specialised operations.
- Define implementor next:
  - Its methods do not have to match with abstractor. However, abstraction can carry out its work by using implementor methods.
- Write one or more concrete implement providing implementation.
- Abstractions are created by composing them with an **instance of concrete implementor** which is used by methods in abstraction.

Example UML Diagram:



Bridge Implementation & Design Considerations:

- If you intend on single implementation, skip creating the abstract implementor.
- Either let the abstraction decide which concrete implementation to use in its constructor or delegate the decision to a third class. In the latter, abstraction remains unaware of concrete implementor & provides greater de-coupling.

- Bridge provides great extensibility by allowing us to **change abstraction and implementor independently** - build & package them separately to modularise overall system.
- By using the **abstract factory pattern** to create abstraction objects with correct implementation you can decouple concrete implementors from abstraction.

Bridge vs. Adapter

- **Bridge** intent is to allow abstraction and implementation to vary independently **vs. Adapter** is meant to make unrelated classes work together.
- **Bridge** has to be designed up front, only then we can have varying abstractions & implementations **vs. Adapter** finds its usage typically where a legacy system is to be integrated with new code.

Bridge Pitfalls:

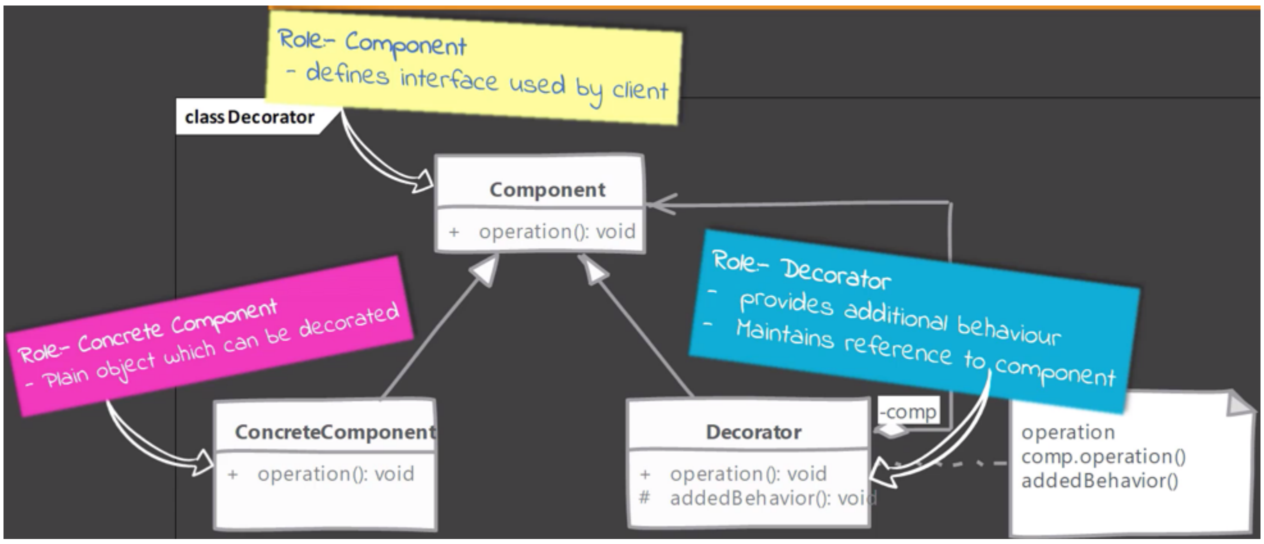
- Conceptual and implementation **complexity**.
- You need to have a well thought out & fairly comprehensive **design** in front of you before you can decide on bridge pattern.
- Needs to be **designed up front** - adding bridge to legacy code is difficult; even for ongoing projects, adding bridge at later time may require a lot of rework.

Friday, 1 April 2022

DECORATOR

What Is It?

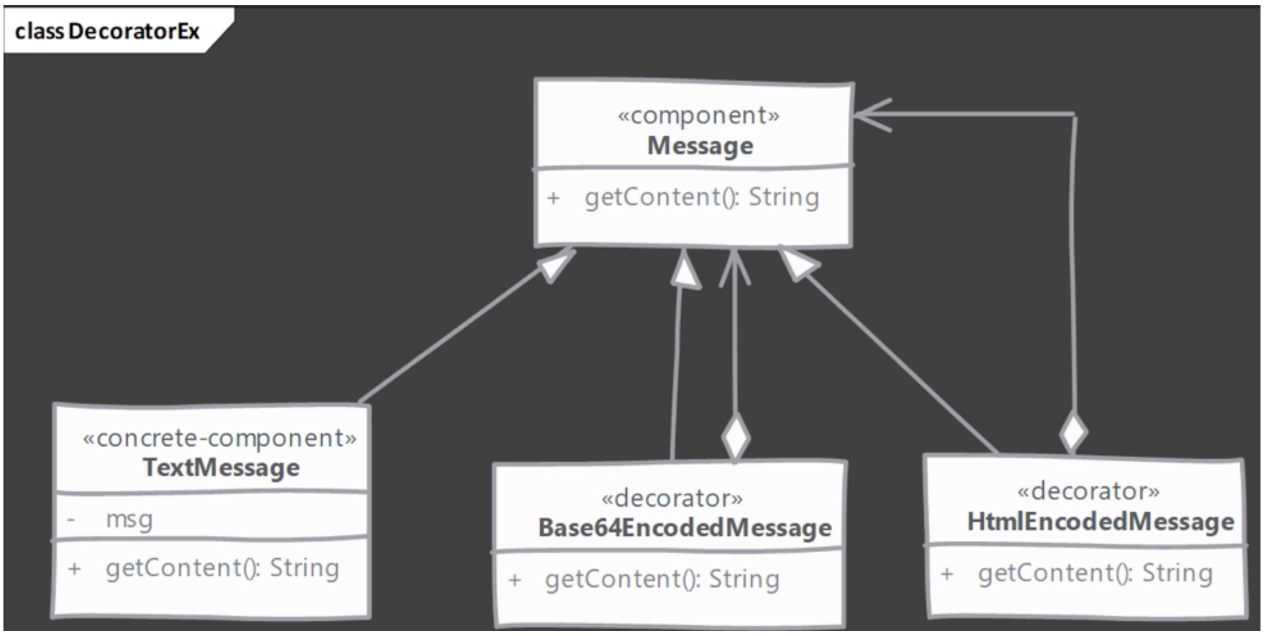
- A structural design pattern used to enhance the behaviour of the existing object dynamically (at runtime) as and when required.
- It wraps the object that you want to enhance within itself and provides same interface as the wrapped object - in a way that the client of original object doesn't need to change.
- A decorator provides the alternative to subclassing for extending functionality of existing classes.



What Are Decorator Implementation Steps?

- Start with the **component**:
    - Component defines interface needed or already used by client.
    - Concrete component implements the component.
- Define the **decorator**:
    - Decorator implements component & also needs reference to component.
    - In decorator methods, provide additional behaviour on top of that provided by concrete component instance.
    - Decorator can be abstract & depend on subclasses for providing functionality.

Example UML Diagram:



Decorator Implementation & Design Considerations:

- Since we have decorators and concrete classes extending from common component, **avoid large state in this base class** as decorators may not need all that state.
- Pay attention to **equals()** and **hashCode()** methods of decorator. When using decorators, you have to decide if decorated object is equal to same instance without decorator.
- Decorators support **recursive composition**, and so this pattern lends itself to creation of lots of small objects that add “just a little bit” functionality. Code using these objects becomes difficult to debug.
- Decorators are more flexible & powerful than inheritance. Inheritance is static by definition but decorators allow you to dynamically compose behaviour using objects at runtime.
- Decorators should act like additional skin over your object. They should add helpful small behaviours to object’s original behaviour. **Do not change meaning of operations**.

Decorator vs. Composite:

- Intent of **decorator** is to add to existing behaviour of existing object **vs.** intent of **composites** is the object aggregation only.
- **Decorator** can be though of as degenerate composite with only one component **vs. composites** support any number of components in aggregation.

Decorator Pitfalls:

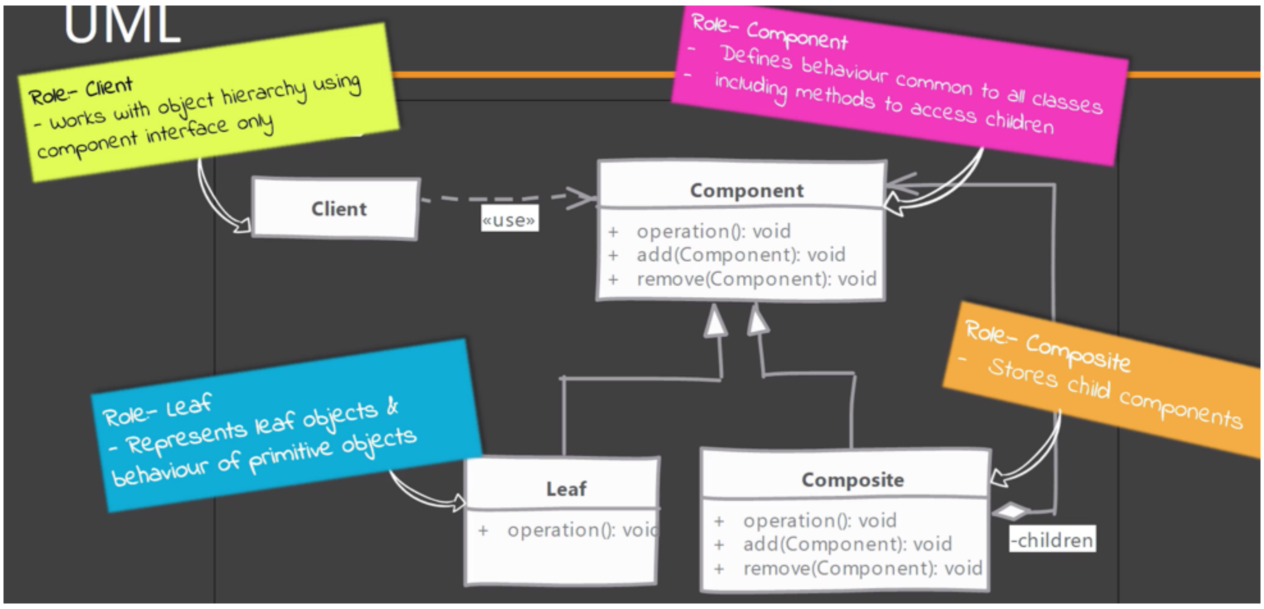
- Using this pattern often results in **large number of classes** being added to system, where each class adds a small amount of functionality. You often end up with lots of objects, one nested inside another and so on.
- Think of decorators as a thin skin over existing object, don't use it as a replacement of inheritance in every possible scenario.

Friday, 1 April 2022

COMPOSITE

What Is It?

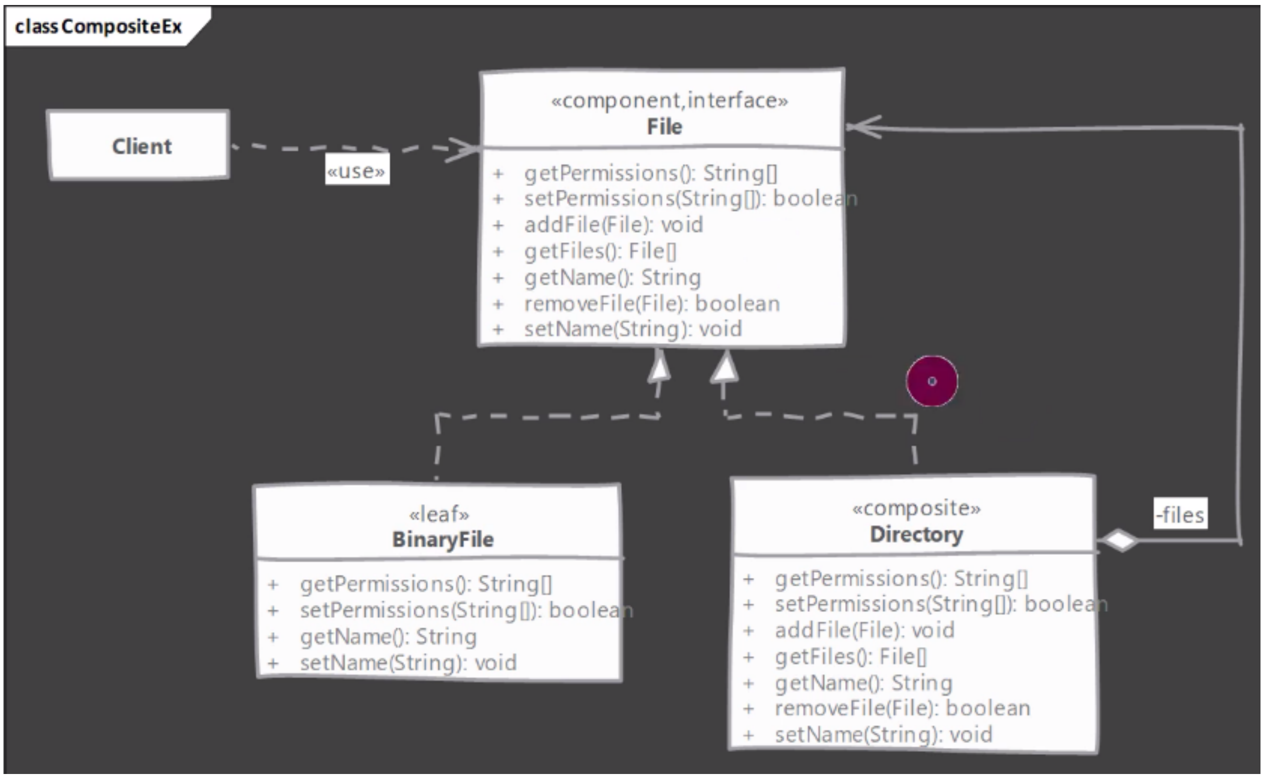
- A structural design pattern use to treat all objects in part-whole relationship or hierarchy uniformly (composites and leafs are treated the same).
- It is **not** a simple composition concept from OOP but a further enhancement to that principal.


UML
Role- Client
- works with object hierarchy using component interface only
Role- Component
- Defines behaviour common to all classes
- including methods to access children
Client
«use»
Component
+ operation(): void
+ add(Component): void
+ remove(Component): void
Role- Composite
- Stores child components
Role- Leaf
- Represents leaf objects & behaviour of primitive objects
Leaf
+ operation(): voi
Composite
+ operation(): void
+ add(Component): void
+ remove(Component): void
-children

What Are Composite Implementation Steps?

- Start by creating an abstract class / interface for component:
  - Component must declare all methods that are applicable to both leaf and composite.
  - Choose who defines the children management operations, component or composite.
- Implement the composite - an operation invoked on composite is propagated to all its children.
- In leaf nodes, handle the non-applicable operations like add/remove a child if they are defined in component.
- Composite allows to write algorithms w/o worrying whether node is leaf or composite.

Example UML Diagram:



Composite Implementation & Design Considerations:

- You can provide a method to access **parent** of a **node** - this will simplify traversal of the entire tree.
- You can define the collection field to maintain children in base component instead of composite but again, that field has no use in leaf class.
- If leaf objects can be repeated in the hierarchy, then **shared leaf nodes** can be used to save memory and initialisation costs but again, the number of nodes is major deciding factor as using a cache for a small total number of nodes may cost more.
- Decision needs to be made about where **child management operations** are defined. Defining them on **component** provides transparency but leaf nodes are forced to implement those methods. Defining them on **composite** is fair but client needs to be aware of composite.
- The goal should be making the client code easy when using composite - this is possible if it works with component interface only and doesn't need to worry about leaf-composite distinction.

Composite vs. Decorator:

- **Composite** deals with tree structure of objects **vs. decorator** simply contains another single object.
- Leaf nodes and **composites** have same interface and composites simply delegate the operation to children **vs. decorators** add or modify the behaviour of contained object and have no notion of children.

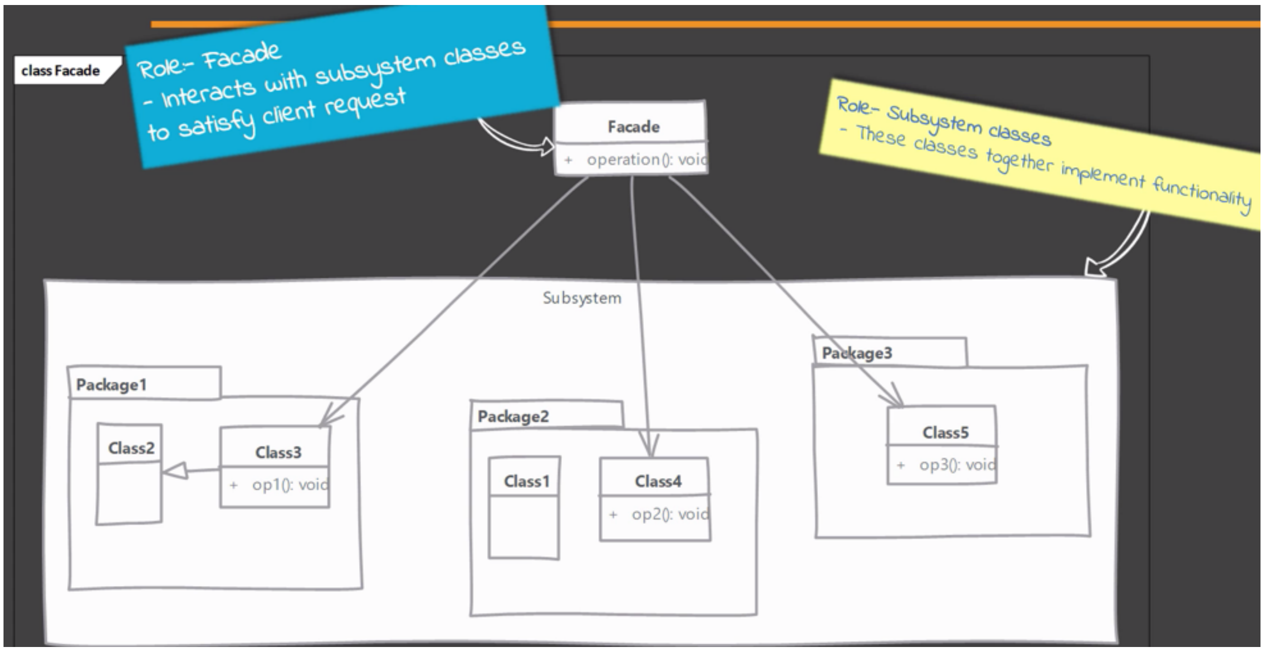
Composite Pitfalls:

- **Difficult to restrict** what is added to **hierarchy** - if multiple types of leaf nodes are present in system, the client code ends up doing runtime checks to ensure the operation is available on the node.
- Creating the original hierarchy can still be **complex implementation** especially if you are using caching to reuse nodes and number of nodes is high.

FACADE

What Is It?

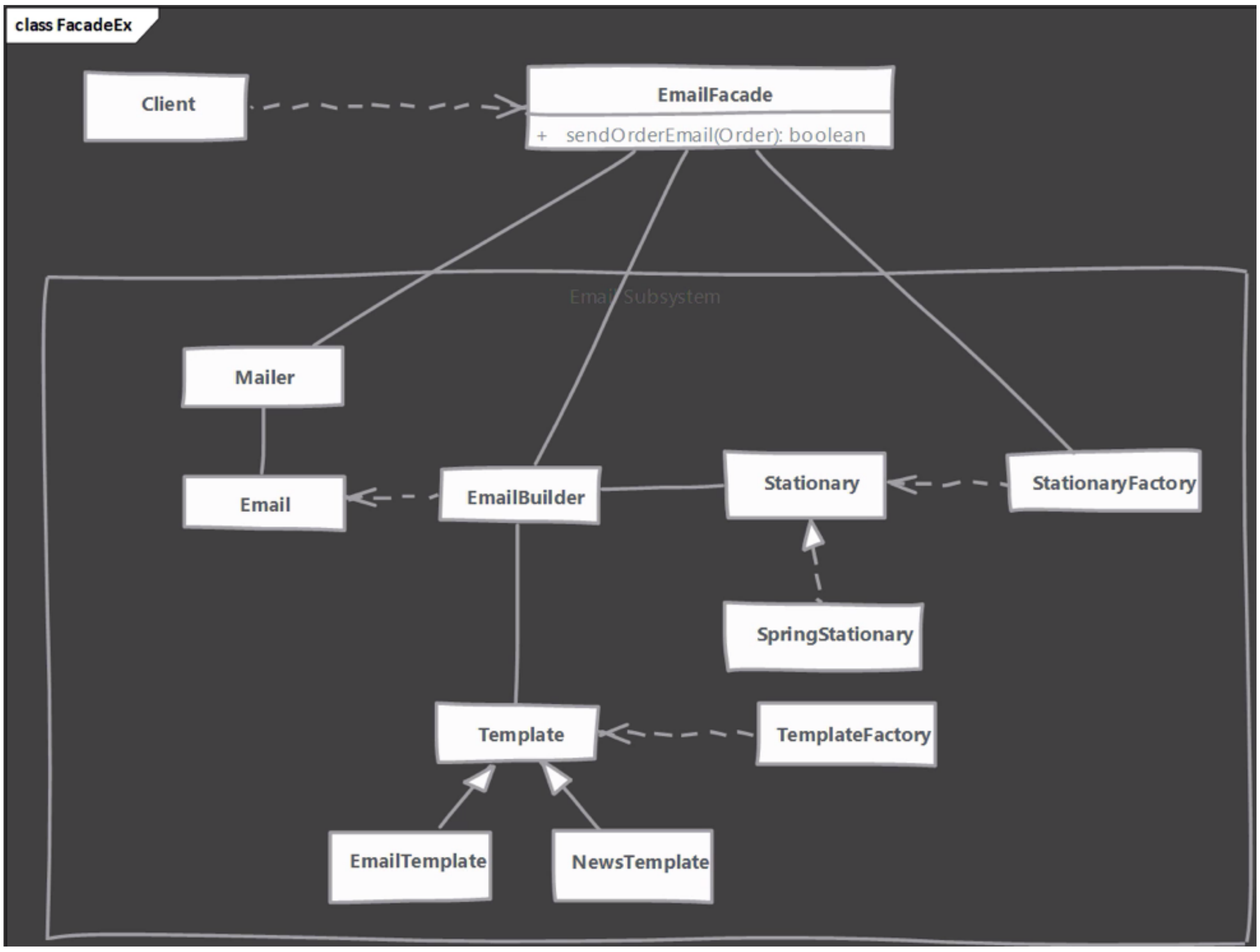
- A structural design pattern used when a client needs to interact with a large number of interfaces and classes in a subsystem to get a result.
- Facade solves the problem of the client getting tightly coupled with those interfaces and classes.
- It provides a simple and unified interface to a subsystem which client interacts to get the same result.
- Facade is not just a one-to-one method forwarding.



What Are Facade Implementation Steps?

- Start by creating a class that will serve as a facade:
  - Determine the overall “use cases” / tasks that the subsystem is used for.
- Write a method that exposes each task:
  - This method will take care of working with different classes of subsystem.

Example UML Diagram:



Facade Implementation & Design Considerations:

- Facade should minimise the complexity of subsystem and provide usable interface.
- Facade can be provided as an interface or abstract class, and client can use different subclasses to talk to different subsystem implementations.
- Facade is not a replacement for regular usage of classes in the subsystem - your subsystem class implementations should not make assumptions of usage of facade by client code.
- Facade is a great solution to simplify dependencies - it allows you to have weak coupling between subsystems.
- If the only concern is the coupling of client code to subsystem's classes and not simplification - use abstract factory pattern.

Facade vs. Adapter

- **Facade** is there to simplify the usage of subsystem for client code **vs. Adapter** is meant to simply adapt an object to different interface.
- **Facade** is not restricted by any existing interface - it often defines simple methods which handle complex interactions behind the scenes **vs. adapter** is always written to confirm to a particular interface expected by client code - it has to implement all the methods from interface and adapt them using existing object.

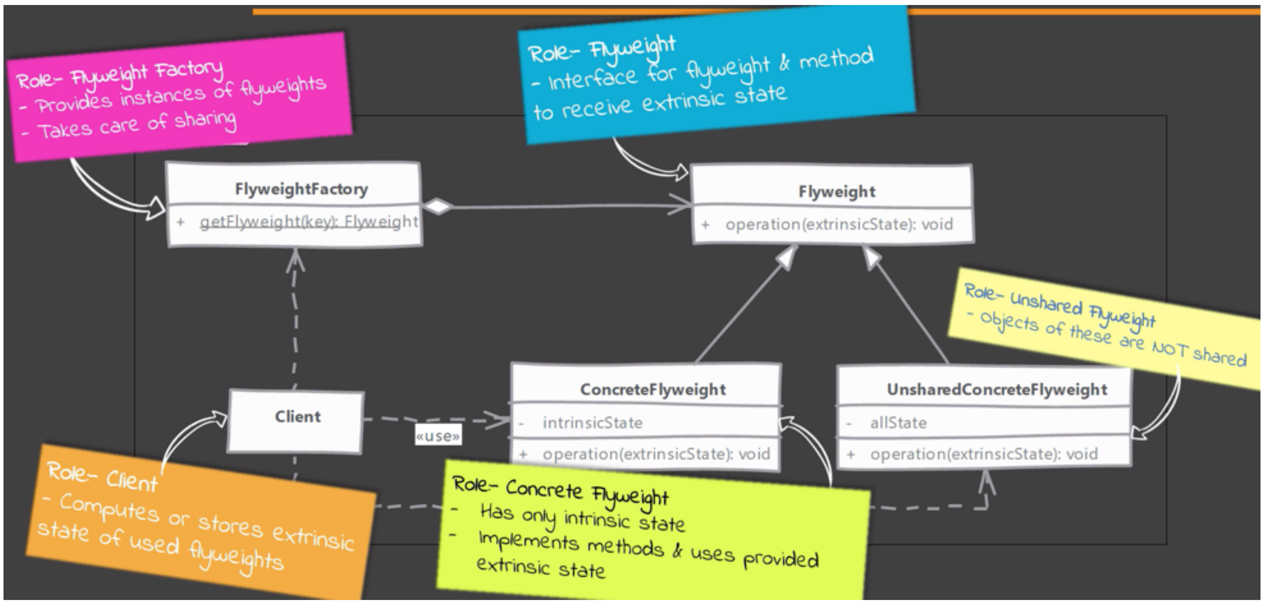
Facade Pitfalls:

- Often an overused/misused pattern & can hide improperly designed API - a common misuse is to use them as “containers of related methods”.

FLYWEIGHT

What Is It?

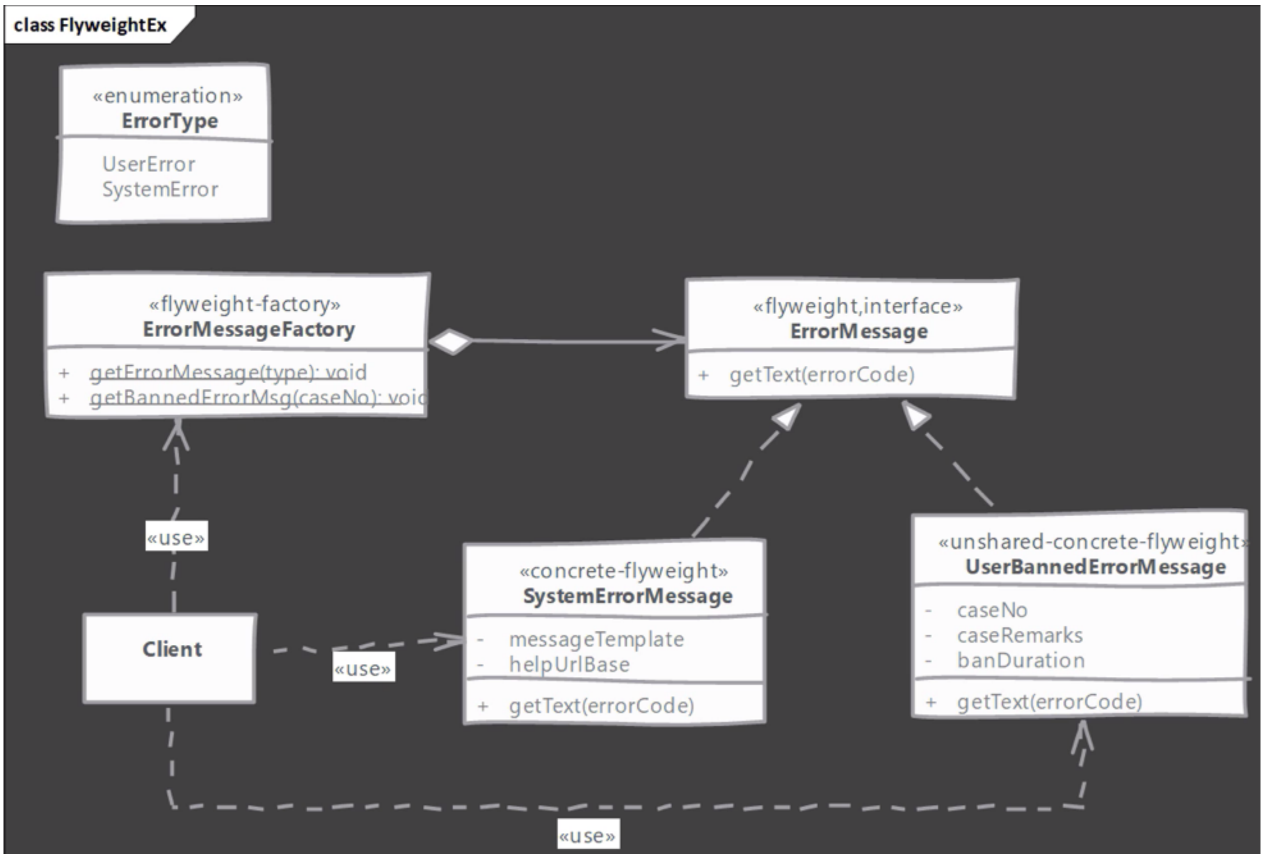
- A structural design pattern used when system needs a large number of objects of a particular class and maintaining these instances is a performance concern.
- It allows to share an object in multiple contexts but instead of sharing entire object, we divide object state in two parts: intrinsic (the one that is shared in every context) and extrinsic (context-specific) state. We then create objects with only intrinsic state and share them in multiple contexts.
- Client/user then provides the extrinsic state to the object to carry out its functionality.
- We also provide a factory so that client can get required flyweight objects based on some key to identify it.


Role- Flyweight Factory
- Provides instances of flyweights
- Takes care of sharing
Role- Flyweight
- interface for flyweight & method to receive extrinsic state
FlyweightFactory
+ getFlyweight(key): Flyweight
Flyweight
+ operation(extrinsicState): void
Role- Unshared Flyweight
- objects of these are NOT shared
Client
«use»
ConcreteFlyweight
- intrinsicState
+ operation(extrinsicState): void
UnsharedConcreteFlyweight
- allState
+ operation(extrinsicState): void
Role- Client
- Computes or stores extrinsic state of used flyweights
Role- Concrete Flyweight
- Has only intrinsic state
- Implements methods & uses provided extrinsic state

What Are Flyweight Implementation Steps?

- Start by identifying the intrinsic and extrinsic states of the object:
  - Create an interface to provide common methods that accept extrinsic state.
  - Add intrinsic state and implement methods in implementation of shared flyweight.
  - Ignore the extrinsic state argument in unshared flyweight implementation.
- Implement the flyweight factory which caches flyweight and provides a getter.
- In the client, either maintain the extrinsic state or compute it on the fly.

Example UML Diagram:



Flyweight Implementation & Design Considerations:

- A **factory** is **necessary** with flyweight pattern as client code needs an easy way to get hold of shared flyweight. Also, number of shared instances can be large so a central place is good strategy to keep track of all of them.
- The **intrinsic state** should be **immutable** for successful use of flyweight pattern.
- Usability of flyweight is dependent upon presence of **sensible extrinsic state** in object which can be moved out of object w/o any issue.
- Some other design patterns like **state** and **strategy** can make best use of flyweight patterns.

Flyweight vs. Object Pool:

- State of the **flyweight** is divided and client must provide a part of it **vs.** a **pooled** object contains all of its state encapsulated within itself.
- In a typical usage, client will not change the intrinsic state of **flyweight** instance as it is shared **vs.** clients can and will change state of **pooled** objects.

Flyweight Pitfalls:

- An additional **runtime cost** for maintaining the **extrinsic state** - client code has to either maintain it or compute it every time it needs to use flyweight.
- It is often difficult to find perfect candidate objects for flyweight. **Graphical** apps benefit heavily from this pattern but a typical web app may not have a lot to gain.

Notes on Flyweight:

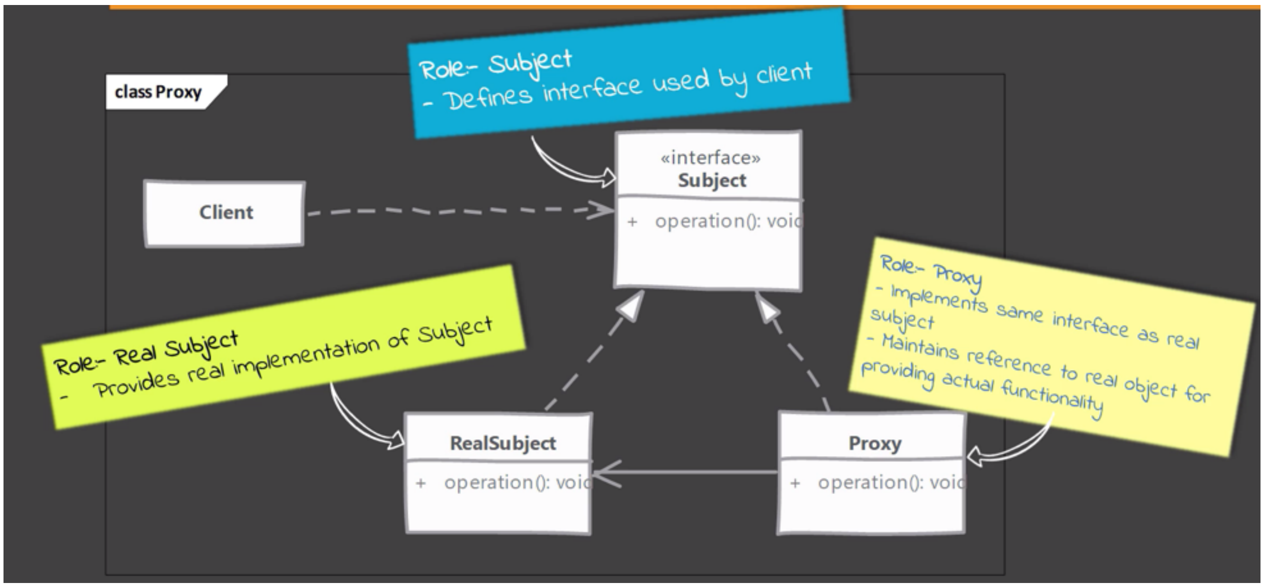
- A good usage case is **graphics** - provide the system configuration as intrinsic state that is to be shared, and provide, for example, the location coordinates as an extrinsic state every time you need to change the position of a player.

Friday, 1 April 2022

PROXY

What Is It?

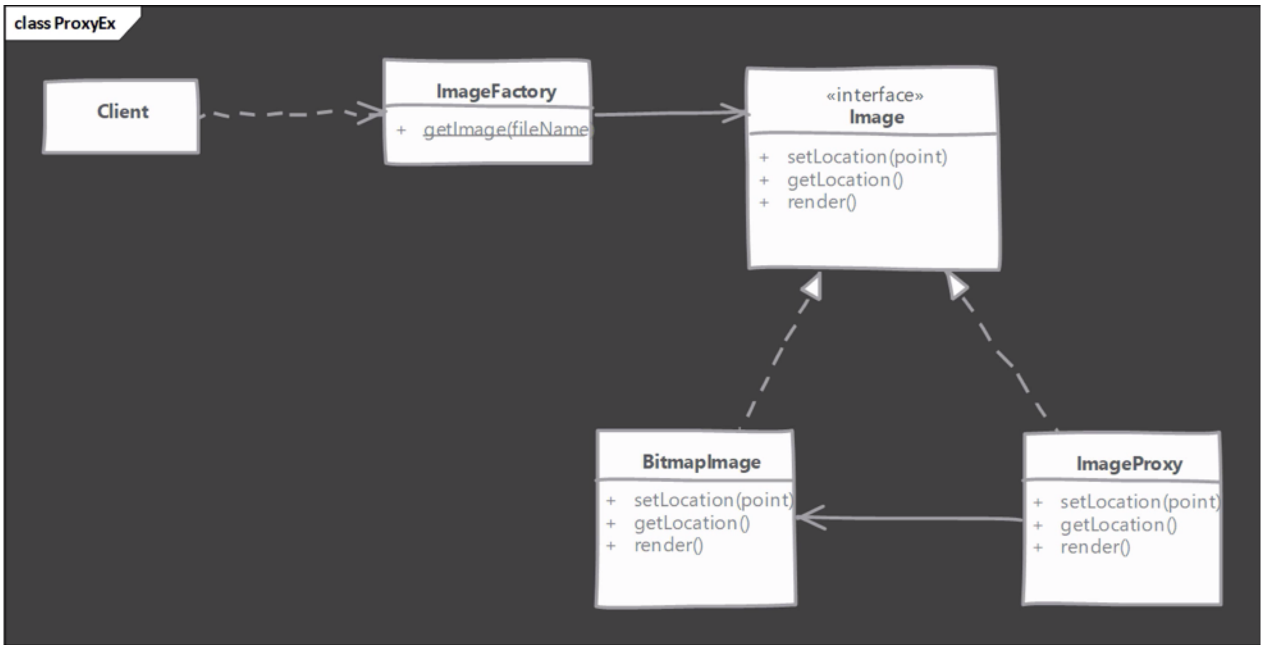
- A structural design pattern used when you need to provide a placeholder or a surrogate to another object.
- Proxy acts on behalf of the object and is used for lots of reasons, some of which are:
  - **Protection** proxy - to control access to original object's operations.
  - **Remote** proxy - to provide a local representation of remote object.
  - **Virtual** proxy - to delay the construction of the object until absolutely necessary.
- Client is unaware of existence of a proxy.


class Proxy
Role- Subject
- Defines interface used by client
«interface» Subject
+ operation(): void
Client
Role- Real Subject
- Provides real implementation of Subject
Role- Proxy
- Implements same interface as real subject
- Maintains reference to real object for providing actual functionality
RealSubject
+ operation(): void
Proxy
+ operation(): void

What Are Static Proxy Implementation Steps?

- Start by implementing a proxy:
  - Proxy must implement same interface as the real subject.
  - Either create an actual object later when required or ask for one in constructor.
  - In method implementations of the proxy, implement proxy's functionality before delegating to real object.
- How to provide a client with proxies instances is decided by the app. We can provide a factory or compose client code with proxies instances.

Example UML Diagram



What Are Dynamic Proxy Implementations Steps?

- Java allows you to implement a **dynamic proxy** - allowing the creation of the proxies at runtime.
- Start by implementing *java.lang.reflect.InvocationHandler*:
  - Invocation handler implement invoke method that is called to handle every method invocation on proxy.
  - Take action as per the method invoked - cache the *Method* instances on image interface so that you can compare them inside invoke method.
  - The invocation handler will accept same argument in constructor as needed by the constructor of the real object.
- Actual proxy instance is created using *java.lang.reflect.Proxy* by client.

Proxy Implementation & Design Considerations:

- How proxy gets hold of the real object depends on what purpose proxy serves.
  - For creation **on-demand type of proxies**, actual object is created only when proxy can't handle client request.
  - **Authentication proxies** use pre-build objects so they are provided with object during construction of proxy.
- Proxy itself can maintain/cache some state on behalf of real object in creation on demand use cases.
- Pay attention to **performance cost** of proxies as well as **synchronisation issues** added by proxy itself.
- Proxies typically do not need to know about the concrete implementation of real object.
- In Java, you can use **dynamic proxy** allowing you to create proxies for any object at runtime.
- Proxies are great for implementing **security** or as **stand**-**ins** for real objects which may be a costly object that you want to defer loading.
- Proxies also make working with **remote services**/**APIs** easy by representing them as regular objects and possibly handling network communication behind the scene.

Proxy vs. Decorator:

- Depending on the type of **proxy** it doesn't need real object all the time **vs. decorator** needs to have a real object for it to work.
- Purpose of **proxy** is providing features like access control, lazy loading, auditing, etc. **vs. decorator** is meant to add functionality to existing functionality provided by object & used by client directly.

Proxy Pitfalls

- Java's **dynamic proxy** only works if your class is implementing one or more interfaces - proxy is created by implementing these interfaces.
- If you need proxies for handling **multiple responsibilities** like auditing, authentication, as a stand-in for the same instance, then it's better to have a single proxy to handle all requirements. Due to the ways some proxies create objects on their own, it becomes **difficult to manage** them.
- **Static proxies** look quite **similar to other patterns** like decorator & adapter patterns. It can be confusing to figure it out from code alone.