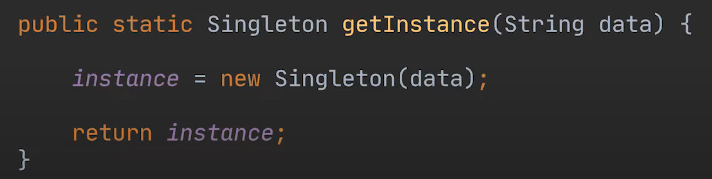
**CREATIONAL DESIGN PATTERNS**

**Singleton – Creational DP**

* **Ensures that only once instance of our class exists**
* **Provides a single point of access to our object**

**Implementation**

* **­­Create a class with one private static field, the type of that field being the type of the class. (The singleton class). This class can have as many other fields as it needs, representing the data of our singleton object.**
* **We create a private constructor, in which we initialize the fields of our singleton object. (But not the instance field)**
* **We made the constructor be private because we only need access to it from a public static method which will initialize and return our single instance.**

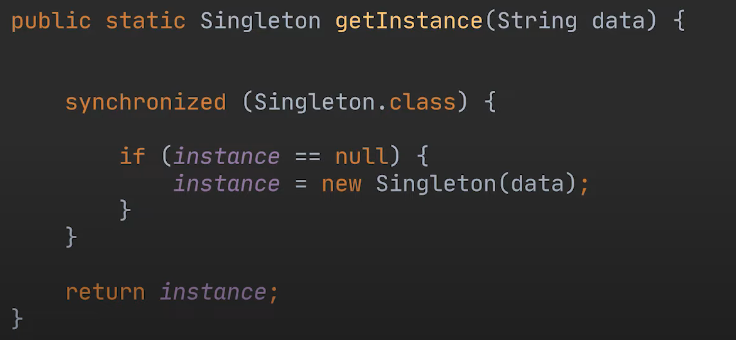
****

* **Checks to have in mind**
  + **How do we ensure that the same instance is returned everytime?**

**We wrap the initialization of our singleton instance inside an if statement which only initializes it if it hadn’t been initialized before.**

* + **What if we are working in a multi-threaded environment? How do we ensure that multiple threads do not access this piece of code at the same time?**

**We wrap the initialization of our singleton instance within a synchronized block**

****

* + **But this means we created unnecessary overhead in our application! Now, every thread has to wait its turn in order to get the singleton instance**

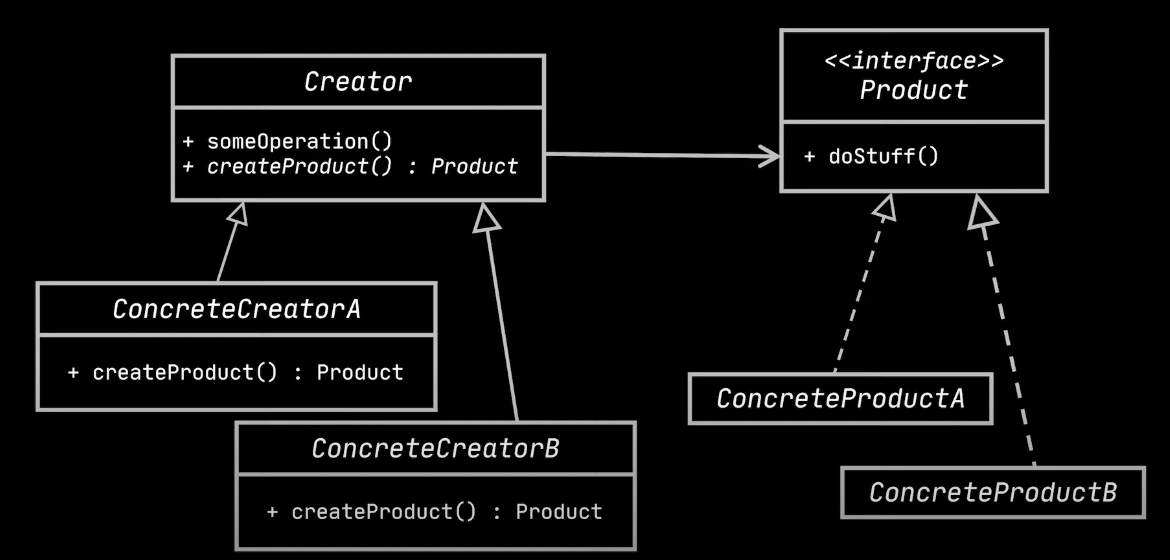
**This can be solved by using the Double checked locking idiom.**

**We wrap the synchronized block inside an if statement, again checking if the instance was previously created or not, and if it was, fetching it directly and not blocking threads.**

* + **There could be one more problem. What happens if a thread sees that the instance hasn’t been initialized and starts initializing it. But at the same time, another thread enters this piece of code. If our object is partially initialized by threadA, it will be immediately updated in the memory and threadB will see it as not null, fetch it and then our application would crash.**

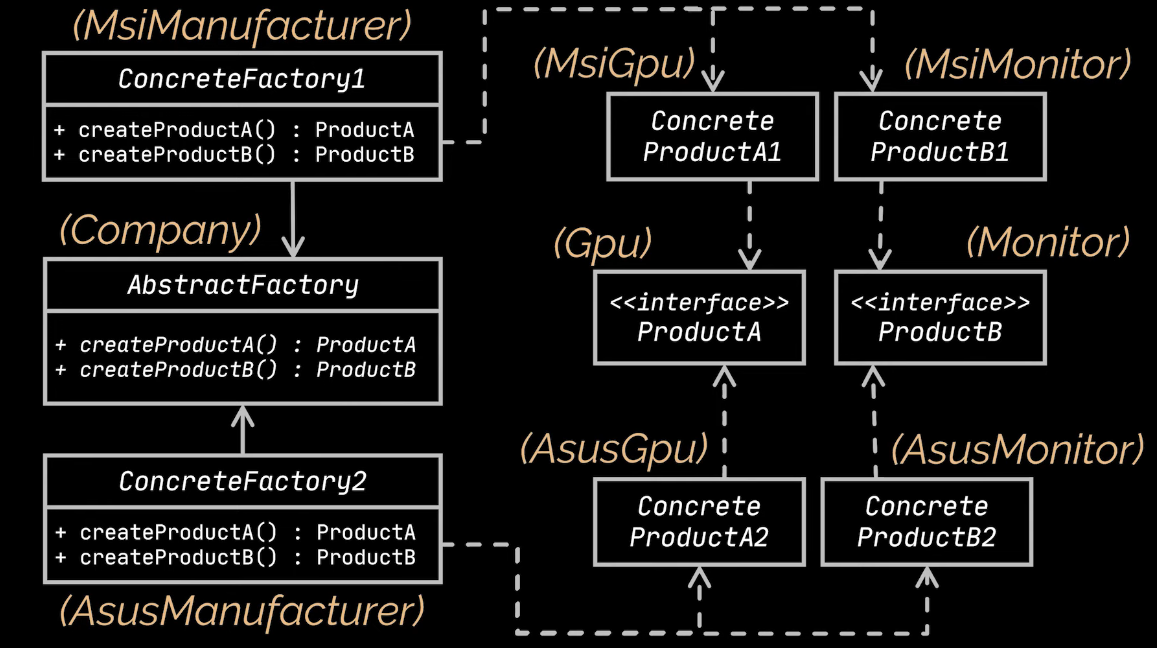
**Thus, we set our instance variable to be volatile.**

**Factory – Creational DP**

****

* **Allows us to create related objects without specifying their concrete classes**
* **It loosens the coupling of our code by separating an object’s construction from the code that uses this object**
* **It is useful to use when we have no idea beforehand of the exact same types of the objects we will work with.**
* **The factory DP makes it easy to extend the product construction code independently from the rest of the application. Thus we can easily add new products to our application without breaking or modifying our previous work.**
* **It respects the Single responsibility principle and the open closed principles**
* **We create an abstract class which represents our factory, with an abstract method called createProduct()**
* **By extending this class, we create as many concrete classes as we need. Each of these classes will be a factory, whose sole responsibility will be to create an object of a certain type.**

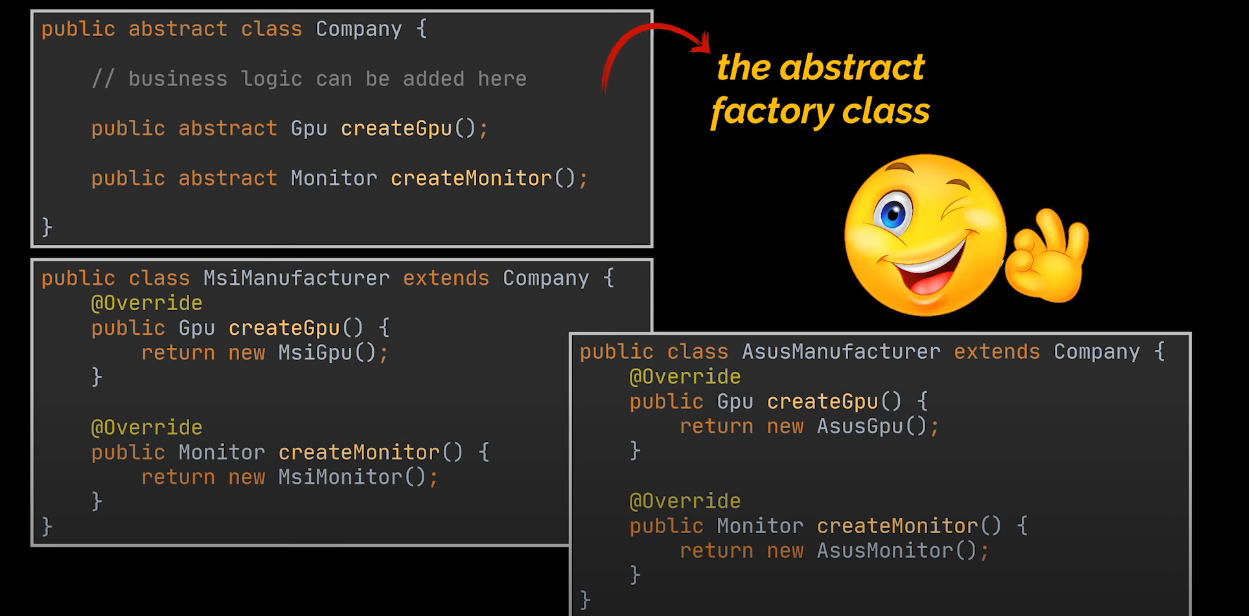
**Abstract Factory – Creational DP**

****

* **allows us to create families of related objects without specifying their concrete classes**
* **It works only when we have related objects in all the families (for CDH for example, we don’t have a RawTile and a Tile or a RawSkirting and a Skirting for this DP to work)**
* **We should:** 
  + **Explicitly create interfaces for each family type of objects and create concrete classes implementing them**

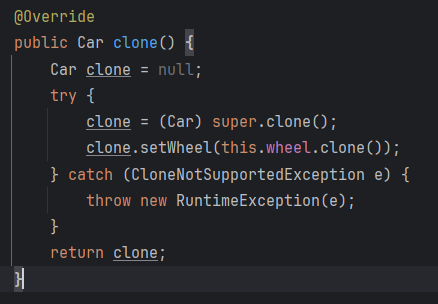


* + **Create an abstract factory which will have different methods for creating an object of each family**



**Prototype – Creational DP**

* **Helps us create objects by copying existing objects without needing subclassing. (If we have a base class, and we want similar products, we would typically create subclasses)**
* **Creates an easy way of creating copies without worrying about private fields, or shallow copies being made by mistake.**
* **Might be helpful to improve performance. If the creation of an object is time consuming, cloning it and working with a copy might be faster.**
* **We start by creating the object we want to copy (which will be called a prototype) and we have to implement the Cloneable interface.**
* **SIDENOTE: We don’t necessarily have to implement the Cloneable interface since all classes inherently implement the Object class and the Objects class implements the Cloneable interface. But it is recommended for clarity !**
* **We override the clone method inherited from this interface**



**which will be the method used for cloning.**

**The clone method automatically creates a shallow copy of the object we call it on.**

**We have to be careful about shallow copies.**

**In case our objects have other objects embedded into them, by simply initializing:**

**cloneObject.embeddedObject = originalObject.embeddedObject.**

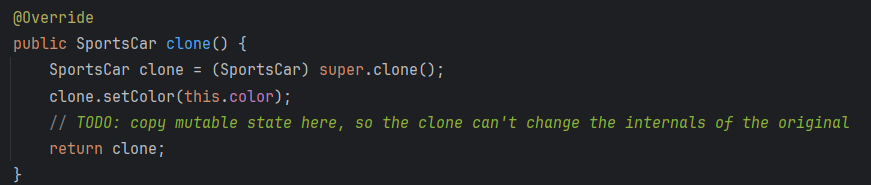
**We are making the embedded objects of both the original and the clone, reference the same objects in memory. Thus, any changes on one of the embedded objects would be reflected on the other embedded object.**

**That is why, each instance variable which is an object has to implement the cloneable interface too, just like in the above example.**

**2) Prototype DP works well with inheritance also. If we had subclasses extending our main class (the one which implements the cloneable interface) we would have to override the clone method, calling the super.clone() thus copying all the fields of the parent.**

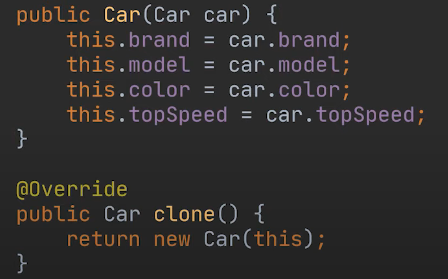
**But also, we would have to add to the clone all the mutable data of the object we would want to clone (so all its fields).**

**Just like that, we are making a shallow copy of the subclass.**

****

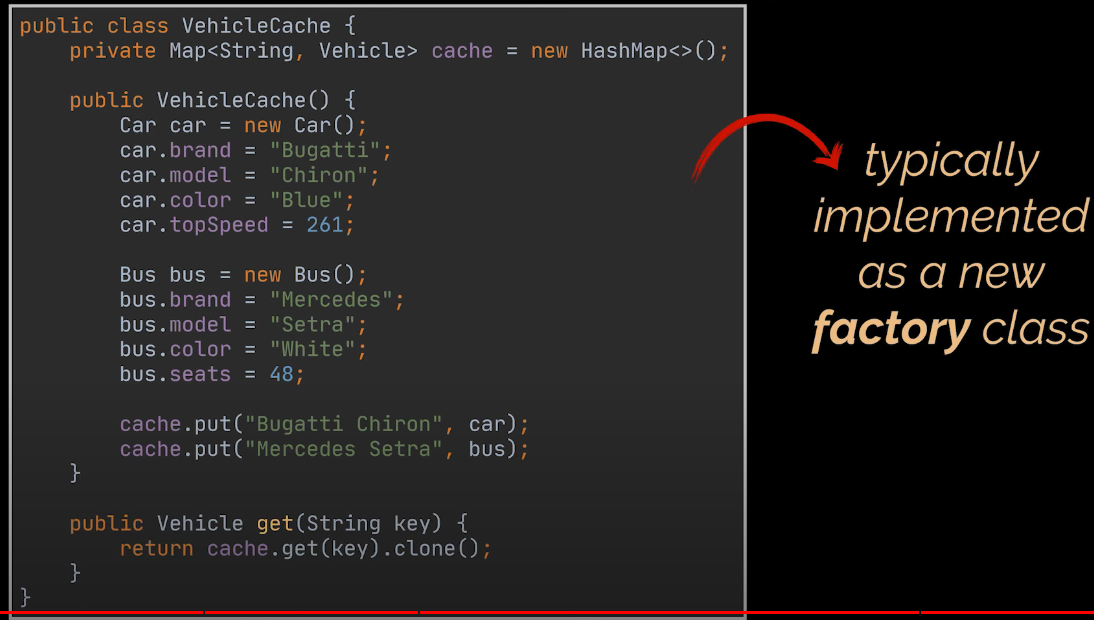
**3)**

* **There is another method of implementing this design pattern**
* **We don’t implement the Cloneable interface, but instead create a constructor in the object we want to clone which takes as parameter an object of itself**
* **We create a clone() method which simply returns a call to the constructor we just created**

****

**4) Sometimes, when certain objects are cloned multiple times, it is useful to implement something called a Prototype Registry. It represents a class which holds our most commonly cloned object, for ease of access and performance reasons.**

**The most basic example would be holding a cache map for these cloned objects.**

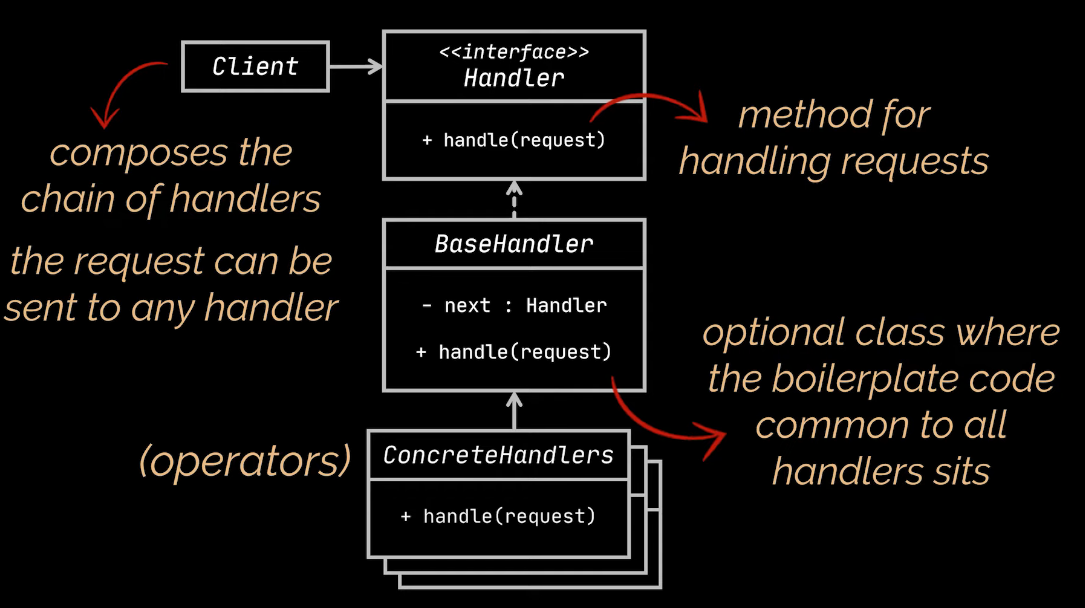
****

**BEHAVIORAL DESIGN PATTERNS**

**Chain of Responsibility – Behavioral DP**



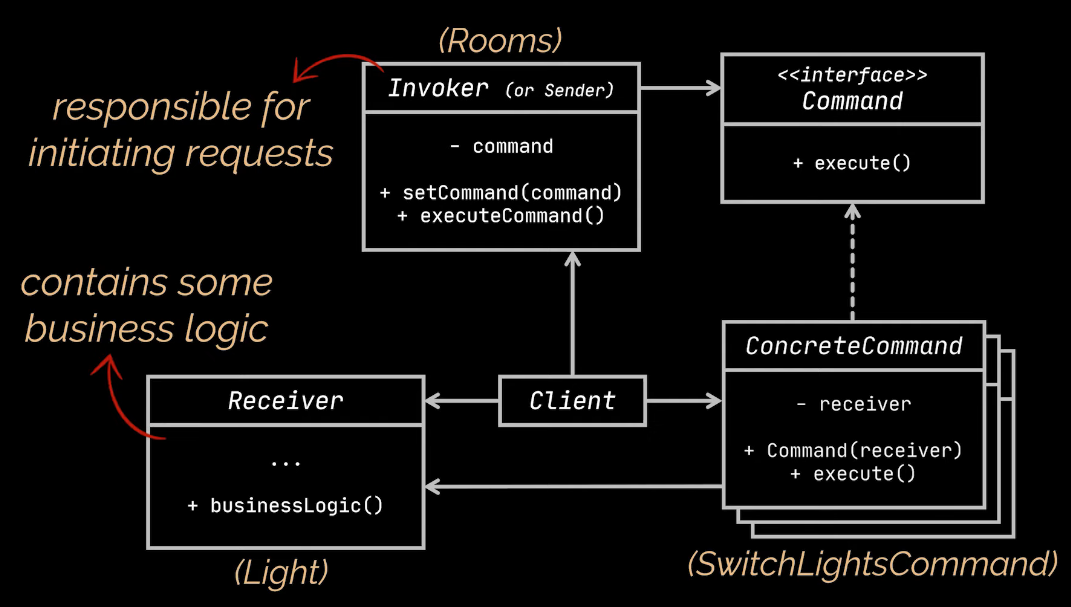
* **Transforms particular behaviors into stand-alone objects called handlers**
* **Each handler is responsible for one particular task**
* **Each handler can decide whether to process the request or pass it onto the next handler**
* **Handler can be removed/inserted dynamically**

****

**Example:**

* **Imagine we have a login page. In order to login, a user must pass several checks.**
  + **Validate Username**
  + **Validate Password**
  + **Check the role of the User**
* **In our example, we will not create an interface storing the handle method. We will simply add it to our abstract BaseHandler**
* **Each concrete handler, provides its own logic and implementation to the handle() method. If the conditions are satisfied, it passes the logic to the next handler, by calling the handleNext() method.**

**Command – Behavioral DP**

****

* **Turns a specific method call into a standalone object**
* **It is useful when we have repeated behaviors in our app, because we can store them in those objects called commands**
* **It is useful because it allows us to store code which will be used at a later date**
* **Objects called invokers transfer the command to other objects called receivers and the receivers are responsible for executing the code**
* **We implement concrete commands by giving them an instance variable of the object on which the command executes(this object is called the receiver). Thus, in the execute command of the concrete command object, we call the method from our instance variable.**
* **Whenever we want to execute a command, we create an object referred to as an invoker. These invokers have as instance variables commands. The logic from an invoker will call the execute from its Command Object, which in turn will call the logic from its Receiver Object.**

**Invokers**

**|**

**| have commands as instance variables**

**|**

**Commands**

**|**

**| have receivers as instance variables**

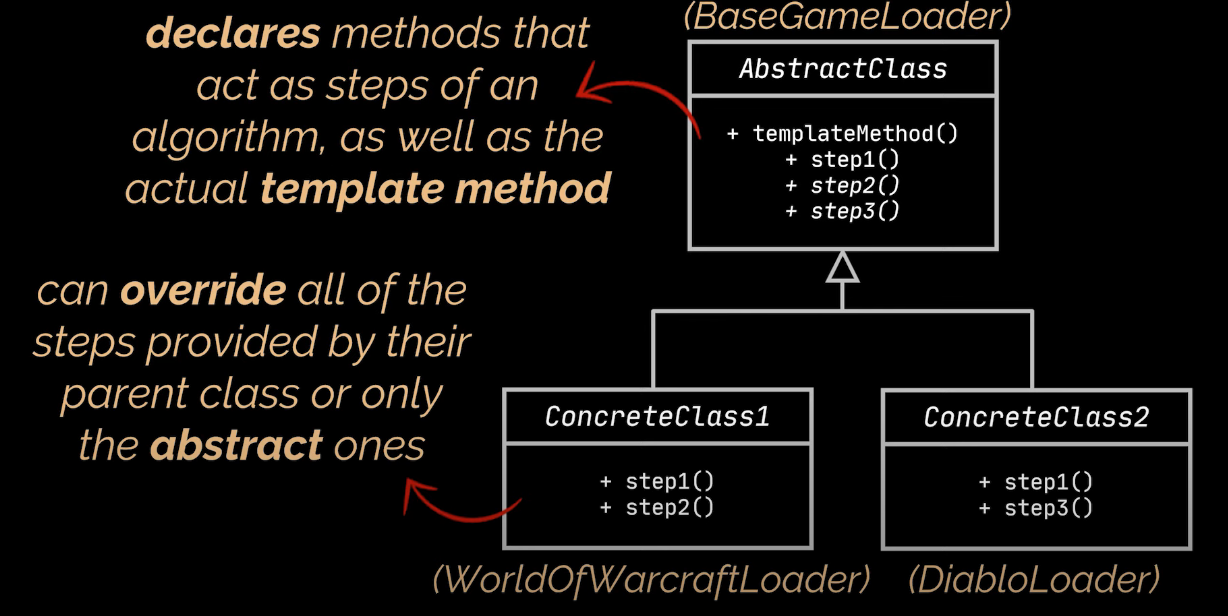
**|**

**Receivers ( Hold the logic, the repeated logic in our app)**

* **Every class gets its own responsibility (Solid respected)**
* **We decouple the classes which call some operations from the classes which actually perform these operations.**

**Template Method – Behavioral DP**

* **Allows us to turn an algorithm into a series of individual methods**
* **Defines the skeleton of an algorithm in the superclass but lets the subclasses override specific steps of the algorithm, without changing the algorithm’s structure**
* **Very useful when dealing with a lot of duplicated code**
* **The code that varies SHOULD remain in the subclasses**

****

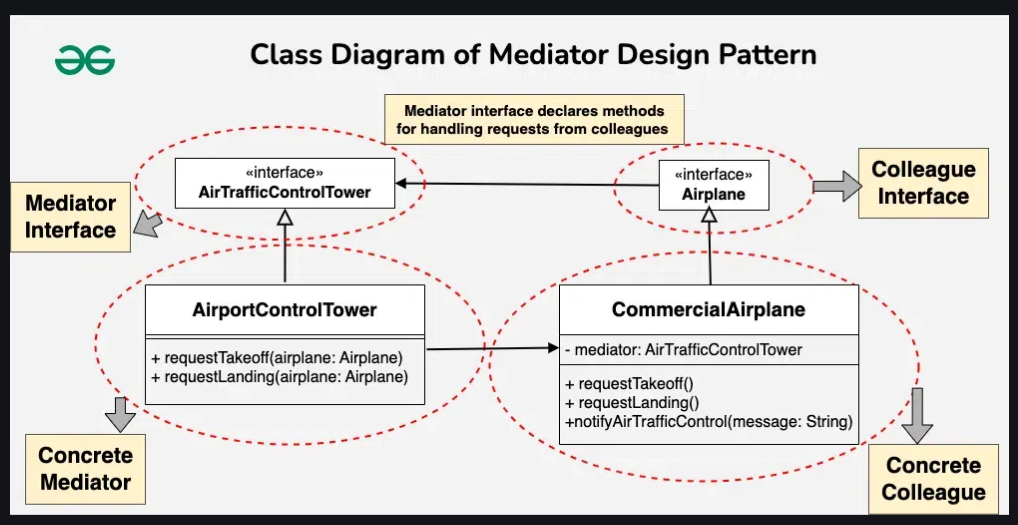
**To implement:**

**In the superclass:**

1. **We break down an algorithm(functionality) into a series of methods**
2. **Put all those steps (method calls) inside a single template method**
3. **The steps can either be abstract or have some default implementation**

**In each subclass, we provide the implementation to the abstract methods, and call the default ones. This depends on however we would like to implement our algorithm.**

**Mediator – Behavioral DP**



* **Instead of objects communicating directly with each other, they communicate through the mediator => reduces dependencies, coupling and promotes modularity**
* **It’s good when our implementation needs a centralized control**

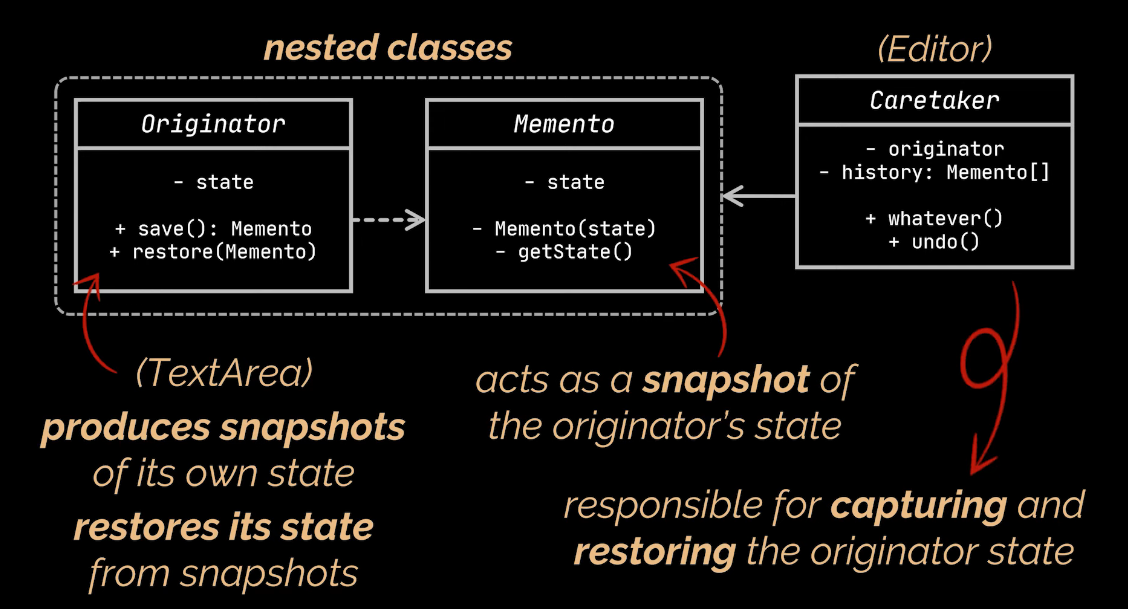
1. **Mediator interface**
   1. **Defines the communication contract between objects. It holds the methods for objects to send and receive data.**
2. **Concrete mediators**
   1. **Implements the mediator methods**
   2. **These methods will have colleagues in their parameter list, calling the needed logic from these colleagues**
3. **Colleagues**
   1. **This is an optional interface defining the methods of the colleagues**
4. **Concrete Colleagues**
   1. **These are the objects which need to communicate between one another**
   2. **Instead of communicating directly, they will be using the mediator which is injected into each one**

**Tip: Usually, the mediator methods are called from colleague methods with the same name**

**Memento – Behavioral DP**

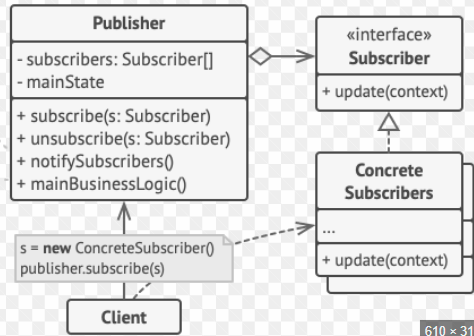
* **Helps us create a snapshot of an object’s value at some point**
* **Delegates creating the snapshot to the actual owner of that state**
* **Using the memento pattern to create snapshots of an object’s state is safer because, since the object itself is responsible for creating its own snapshot, no other object can access the object’s state**
* **We call the object whose snapshots we want to take the Originator. It will have methods for modifying its data as well as taking a snapshot of its data and restoring its data from a snapshot. (Only class which directly deals with the Memento)**
* **Another object, called the Caretaker is responsible for holding the list of snapshots. This object contains the write/undo methods which**

1. **Update the Originator’s state**
2. **Manages the snapshot list, fetching the required snapshot when needed.**

****

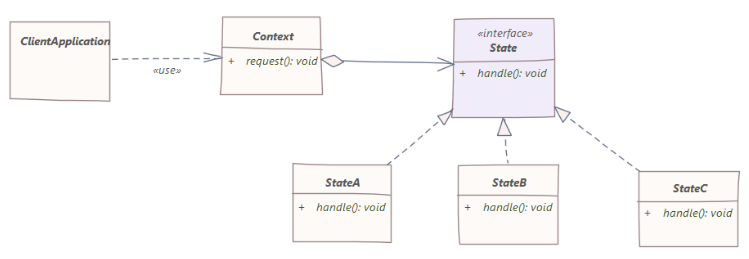
**Observer – Behavioral DP**

* **Used for creating a subscription mechanism. Classes can choose to subscribe to an event, and they will receive whatever necessary information/ call their own logic when that event occurs.**
* **Allows us to take action on a set of objects if the state of another object changes.**
* **The list of subscriber can change at runtime**
* **The objects which subscribe to something are called Subscribers/Observers.**
* **The object which notifies the subscribers when an event occurs is called the Publisher/Observable**
* **The subscribers/observers implement an interface with only one method, called update(). This method will hold the logic needed for execution after the event has been fired by the publisher/observable;**
* **The publisher/observable will hold 3 main methods**
  + **the subscribe(): adding an element to its list of subscribers**
  + **the unsubscribe(): removing an element from its list of subscribers**
  + **the notifySubscribers(): The “Event Firing” Method. In this method, we iterate over the list of subscribers and call the update() method on each one.**

****

**State – Behavioral DP**

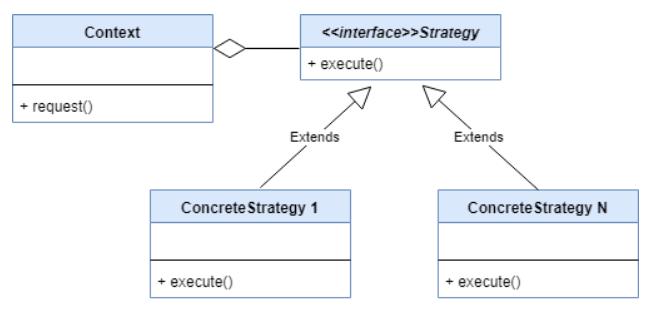
* **It lets an object alter its behavior (same method with different functionalities) when its state changes**
* **We can change state at runtime**
* **Respects S and O from Solid.**
* **We create an interface called State which holds** 
  + **The methods next and previous which have our Context Object as parameter and set the state of this object based on our FSM logic.**
  + **Different methods whose logic will differ based on the concrete state implementing the interface.**
* **Multiple concrete states which implement this interface.**
  + **Each concrete state will have its prev and next state hardcoded (this is a drawback of using this DP)**
  + **Each concrete state will provide its implementation for the methods inherited.**
* **And now we have a concrete object which can be in those different states**
  + **It will hold a reference to the State object**
  + **Its methods will call the methods from the state object it has as instance variable.**

****

* **To use it**
  + **We create the context object**
  + **We allocate its initial state**
  + **We only call the methods from the context (which behave differently based on the current state)**

**Strategy – Behavioral DP**

* **it lets us define a family of algorithms, place each algorithm in a separate class and make their objects interchangeable**
* **We can change strategies at runtime**
* **Respects S and O from Solid.**

****

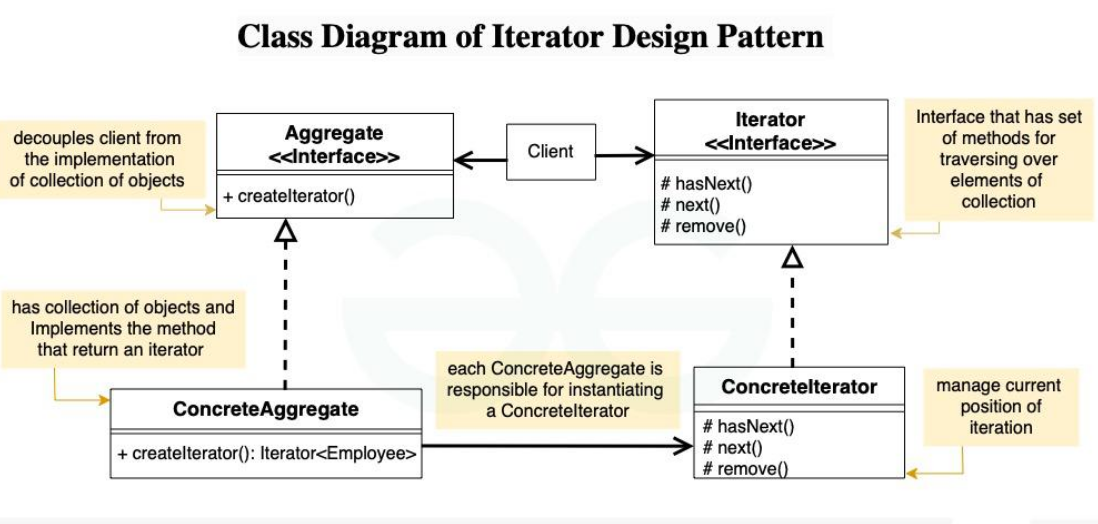
* **We have an object on which we want to use the strategies. We call this object the context. An abstraction of type Strategy which we define will be given to the context as instance variable.**
* **Each concrete strategy will implement the needed logic**
* **Using this pattern presumes instantiating the context object, setting a strategy and then using the methods**
* **The implementation is very similar to that of the state DP**

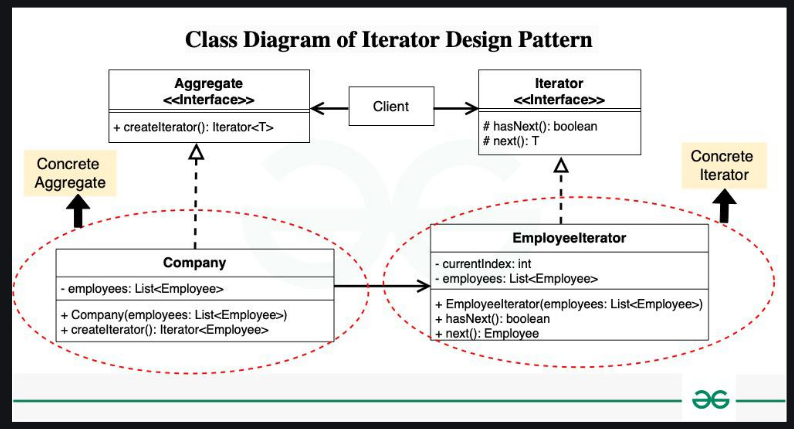
**State vs Strategy vs Chain Of Command**

* **Strategies are completely independent and unaware of one another. On the other hand, States are dependent and our logic should imply that we must change from certain states to other certain states**
* **The strategy patters mostly presumes that we have multiple strategies for achieving the same output. On the other hand, the State pattern presumes that we do things different, based on the state, hence the output may vary.**
* **We can think of the State Pattern as an extension of the Strategy( or a strategy pattern with additional restrictions, like the fact that states are aware of other states)**
* **The chain of command pattern tries to accomplish one thing in a certain way, and if it fails, it either stops the execution or passes the obligation to next handler in the chain. Neither State nor strategy have this logic. We shouldn’t expect any of those two to be obliged to move from one state/strategy to another.**

**Iterator – Behavioral DP**

* **This DP allows us to access the elements of an aggregate object(list, custom object etc…) without exposing the underlying structure of this object**
* **Every Collection in java implements an Iterator**

****

****

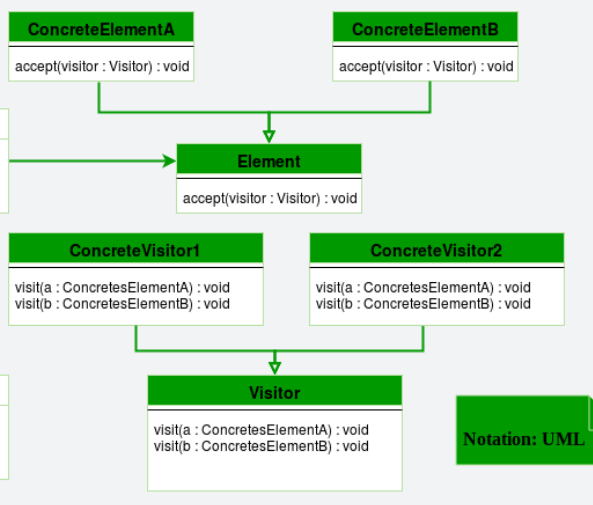
* **We define an Iterator interface which will be implemented by all concrete iterators with some common methods being hasNext(), next(), remove() etc…**
* **Concrete Iterators implement this method and define the implementations of these methods.**
* **We define an Aggregate interface which defines the method of creating an iterator. For less code, the Iterator and aggregate will be defined as generic.**
* **The Aggregate Iterators hold our collection of objects and implement the method for creating our iterator**
* **To use this pattern, we initialize an aggregate object and call the createIterator method on whatever list we want to iterate over.**

**Visitor – Behavioral DP**

* **We use the visitor DP when we want to call a method on a group of similar kind of objects. (Maybe by iterating over a list of child elements of the same parent element)**
* **Classical way of doing this?**
  + **Creating an abstract class, defining some abstract methods and then creating concrete classes which extend our abstract class and implement this method.**
  + **Create a list of <Parent> and populate it with different instances of Child1, Child2…. Iterate over the list and call the method on the objects.**

**What if we now have to create a new method for these objects? We change the parent class => We need to change all the child classes.**

**That’s where the Visitor Design Pattern comes in.**

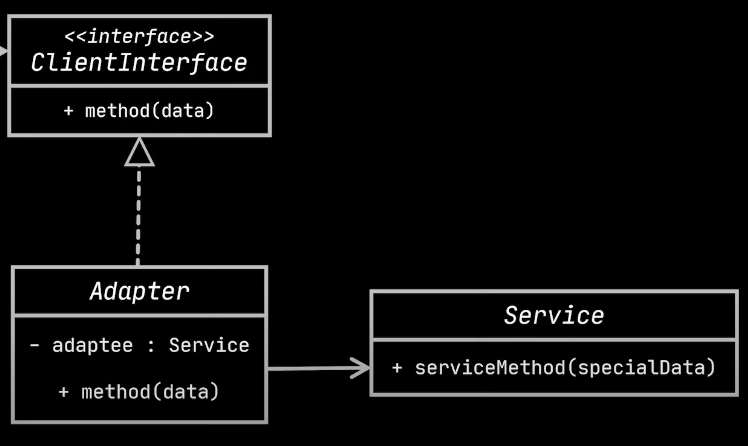
****

* **Visitor:** 
  + **The visitor is an interface, containing only the visit method. It will have as many visit methods as many concrete elements we have.**
* **Visitable(Element in our diagram)**
  + **The visitable is the parent class in our previous example.**
  + **It can be either an abstract class or an interface and its only role will be that of defining the accept method, which receives as parameter a Visitor Object**
* **The concrete objects (concrete visitables)**
  + **Implement the visitable and provide the implementation to the accept method**
    - **accept(Visitor visitor)**
    - **{visitor.visit(this)}**
* **The concrete visitors implement the visitor interface and provide the implementation for each type of concrete visitable objects**
* **Each time we want to add a new operation to our “visitables”, we create another concrete visitor and provide the implementation**
* **To use the code**
  + **create a list of visitable objects**
  + **create instances of the visitors who have the necessary implementations**
  + **Iterate over the list, calling the accept method on each “visitable”**

**STRUCTURAL DESIGN PATTERNS**

**Adapter – Structural DP**

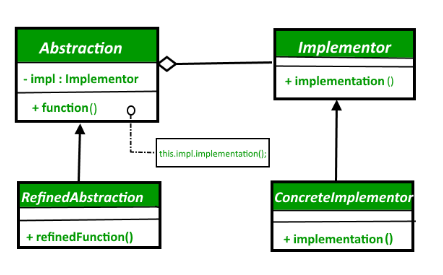
* **It help us incorporate our old code into new functionalities, without modifying it**
* **Adaptee**
  + **Existing class which needs to be adapted to work with the new functionality**
* **Target**
  + **The new “target” functionality, which we want to apply on our adaptee**
  + **Typically an interface, defining the skeleton of the new functionality**
* **Adapter**
  + **The class which will make our adaptee work with the target interface**
  + **It implement the target interface, it has the adaptee as an instance variable and implements the methods of the target interface using some of the old logic from the adaptee**

****

* **To use this pattern, we instantiate the adapter referencing the target interface and giving to the adapter constructor, an object of type adaptee. Then we simply call the methods on our adapter object.**

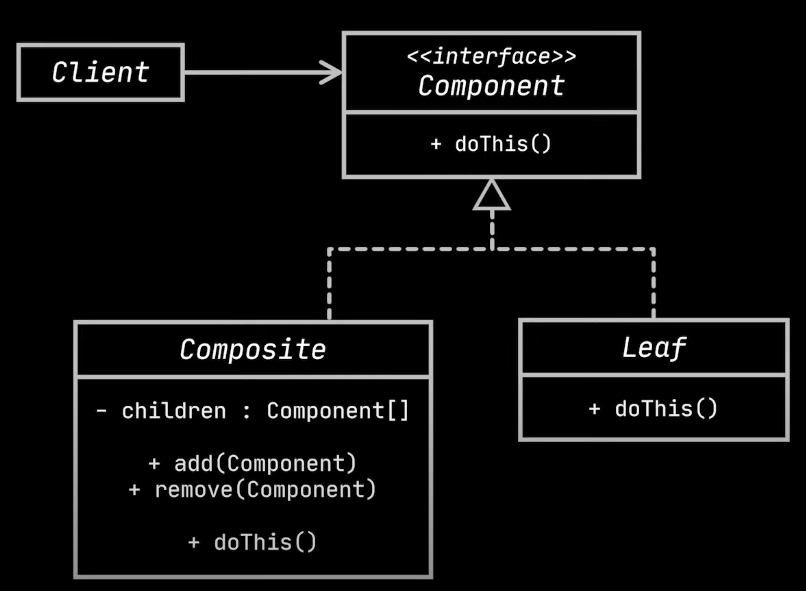
**Bridge – Structural DP**

* **It decouples an abstraction from its implementation so that the 2 can vary independently**
* **It loosens the coupling of our application**
* **Both the abstraction and the implementation will be abstract classes or interfaces:**
  + **Those who extends the abstraction are called refined abstractions**
  + **Those who extends the implementation are called concrete implementor**
  + **The abstraction will have the implementation as instance variable.**
  + **Since we are able to create an abstraction with any concrete implementor, we change the functionality at runtime.**

****

* **To use this pattern, simply create any combination of refined abstraction, concrete implementor suits your needs.**
* **In our example :** 
  + **We can create a new Circle(Blue) because we can define the operation Shape( Color)**

**Composite – Structural DP**

****

* **Only works for objects which can be represented as a tree (an object of a type has as instance variable a list of objects of the same type/ a simple object of the same type)**
* **It lets us treat composite objects just the same as we would the simple objects**
* **Common Interface**
  + **This is the interface which defines the methods we want to apply on our objects, be them leaves or composites.**
  + **Both the leaves and the composites will implement this interface**
* **Leaf objects**
  + **The simple objects, the building blocks of our compositions**
  + **They inherit the methods from the interface and provide the basic functionality**
* **Composite objects**
  + **Objects which hold references to other composite objects or to leaf objects**
  + **They inherit the methods from the interface and implement them by calling the “basic functionality” of each of their “child” objects.**

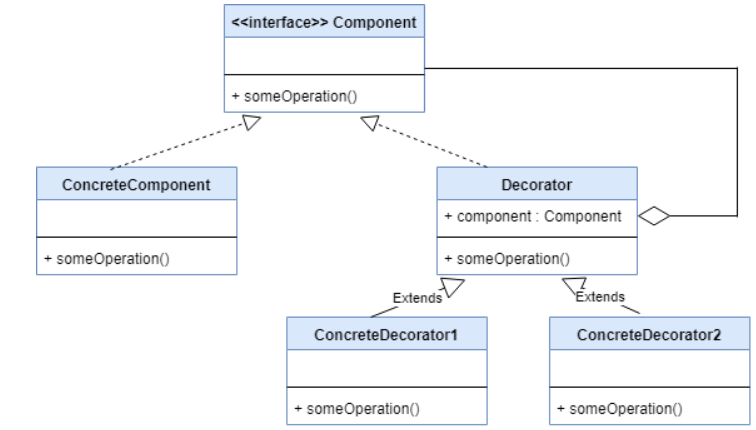
**To implement this pattern, we create the “tree structure” of objects and call the needed methods on whichever element we choose.**

**It is important to note that the references held by the composite objects(instance variables representing other composites/leaves) should be of the interface’s type.**

**This way, we ensure that no matter what type of object we will encounter when traversing the tree, the needed methods can be called.**

**Decorator – Structural DP**

* **Allows us to add new functionalities to an existing class, without modifying its structure**
* **It acts as a wrapper to the existing class**

****

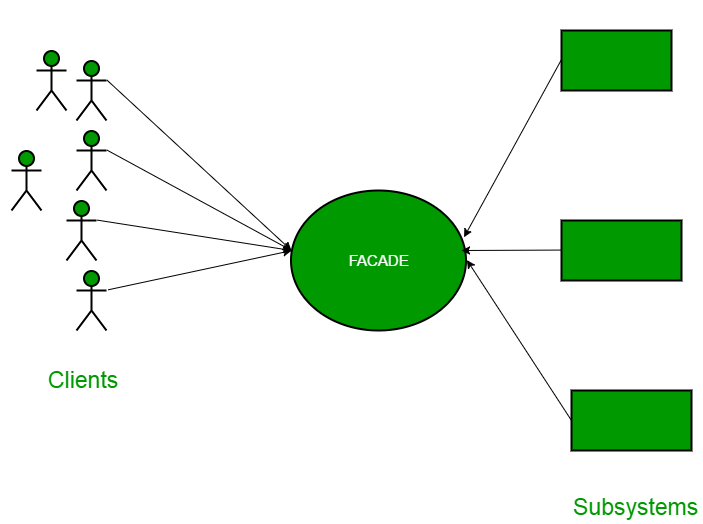
* **We define an interface which will hold the logic of our component and a concrete component implementing this interface. This will represent our base class, the class for which we will be creating the decorators s.t we can add additional functionalities to it.**
* **We create a concrete class representing our base decorator. This class will hold as instance variable, a variable of the interface type and will also implement this interface. This way, we will implement all the methods from our interface by calling the methods from our interface instance variable.**
* **By extending this decorator class, we can create as many decorators as we want. They will also implement the methods from the interface, first by calling the super.method() to retain the original functionality and then by adding additional logic.**
* **To use this pattern, we simply instantiate a decorator object and give to its constructor a base object.**

**Bridge VS Decorator**

* **They seem very similar in their end result. But we should use the bridge DP when we don’t want to “hard couple” two abstractions together(I am using the term abstraction here in the common Java way) while we should use the decorator when we already have some kind of logic withing an object, and we want to add to it without modifying it.**
* **I like to think about this difference from the perspective of a project’s lifetime. The bridge decorator would come at the beginning, when the initial logic of our app will be created. The decorator will appear later, if we need to add to something we already have without modifying it.**

**Facade – Structural DP**

* **Helps us centralize logic in one place, while restricting direct access to the inner workings of our logic**
* **It avoids having to repeat the same code in multiple places**
* **The client will only interact with the façade, which knows the behavior of the classes it encapsulates and will direct the client’s requests appropriately**
* **It helps minimize dependencies and combine multiple “behind the scenes” operation into one**

****

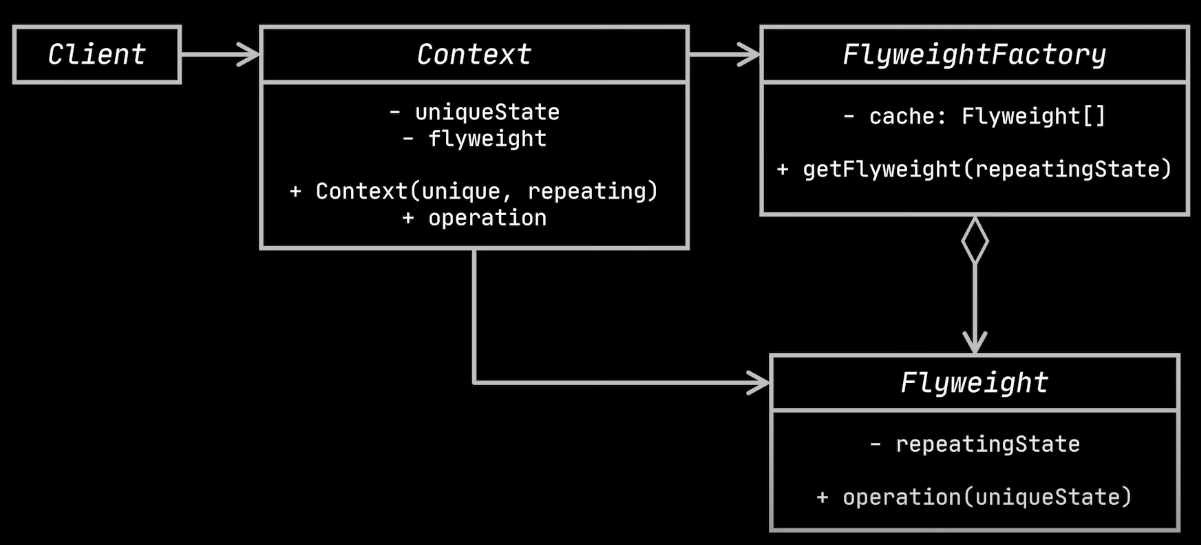
* **We create any number of classes, each with its different logic**
* **We create a façade class which will hold references to all the classes we want to encapsulate**

**The façade class can create methods from calling any number of methods from these classes.**

* **To use this pattern, simply instantiate a façade object and let the client use its functionality.**

**Flyweight – Structural DP**

* **Useful for dealing with RAM Problems, related to having a massive number of similar objects in the memory at the same time.**
* **Intrinsic Values**
  + **Properties that are shared between our objects (Maybe with a limited number of possible values)**
  + **These are the properties (it might be one property or multiple) we use as the key to our haspMap**
  + **These properties should be immutable!**
  + **While not mandatory, it is a good practice to move the intrinsic properties into their own object**
  + **We create only one object for a set of intrinsic values**
* **Extrinsic Values**
  + **These are the properties which vary for each object and should be mutable**

****

* **To use this pattern, separate the intrinsic values from the extrinsic ones.**
* **Create a Factory class, which holds as instance variable a hashmap where the key is an intrinsic value and the value is the object we want to create(the one which would be duplicated if we weren’t using this pattern.**

**This class will hold a method which checks if our hashmap contains a certain key.**

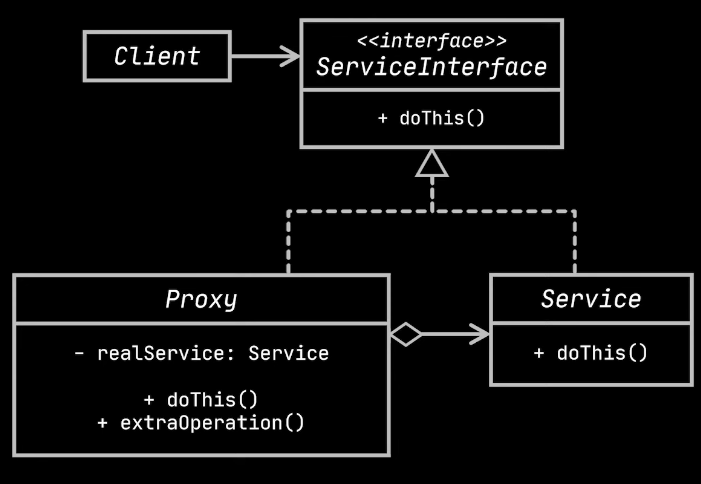
* + **If it doesn’t then it instantiates a new object, adds it to the map and returns this object.**
  + **If it does, then it simply returns the object from the map.**
* **Think of this pattern as an optimization pattern, or a “one time use object” pattern( for example, a one time render).**
* **It shouldn’t be used for creating object which need to be managed afterward since the extrinsic properties keep getting rewritten.**

**Proxy – Structural DP**

* **Adds an additional layer between the client and a certain object**

**Instead of having client <-> object, we will have client <-> proxy <-> object**

* **Allows us to control access to a particular object by performing something before or after the request reaches that object**

****

* **We create an interface which defines the business logic of our object, and a concrete object implementing it**
* **We create a proxy class which also implements this interface and has as instance variable, the concrete object.**

**The proxy class will provide the implementation to the interface’s methods by calling the same method from its instance variable (the object for which we created the proxy) and providing any additional needed functionalities.**

* **The client will directly interact with the proxy component, using it as it if was an instance of our original component.**

**The most common use cases for the proxy pattern are lazy loading, caching, access control (we could restrict access to an object based on certain conditions), logging, virtual proxies and remote proxies.**

**Proxy vs Decorator**

* **These 2 patterns are very similar in their implementation but they serve different purposes**
* **When it comes to the implementation, think of the initial Decorator class as the equivalent of a proxy. But a proxy, doesn’t have any more concrete proxies extending it, while the decorator does.**
* **Implementing the decorator doesn’t necessarily mean that we won’t use the base object anymore. We can use both the original object and its decorators whenever we would need them. Implementing the proxy typically means that we won’t access the original object directly anymore, but instead use the proxy in its place**
* **Typically, we would only have one instance of a proxy for a certain object. But we could have multiple decorator instances. We could even chain these decorators to add multiple functionalities all at once**