# A Study of Gang of Four Design Patterns

By

Sagar Shah

Independent Study Report EECE 8850-01

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

December 11, 2019

Nashville, Tennessee

Submitted to:

_____

Dr. Abhishek Dubey, Ph.D.

**Abstract**

A pattern is a persisting solution to a standard problem. This report documents the 23 design patterns identified in the book 'Elements of Reusable Object-Oriented Software' also known as the Gang of Four (GoF) design patterns. The report follows the original template of classifying patterns based on their purpose. It offers a concise explanation of seven key points for each pattern. The seven key points cover the four basic elements of a pattern as identified in the book. The report presents a classification of patterns based on five different object-oriented concepts that are applicable to them. This serves as a reference for designers to pick a pattern based on their application's context. Each pattern is complemented with an example, its sample code and its brief explanation.

## I. Introduction

To rephrase what Christopher Alexander, a famous architect and design theorist, each pattern is a description of a problem that occurs frequently in our environment and its solution which when implemented could always be different even if done over a million times.[1] Design patterns are a template to solve a recurring problem. However, they do not represent a strict algorithm required to formulate a solution. A design pattern offers a solution to a persisting problem in software design. [2] They provide help in choosing between the design alternatives that are good for system reusability and the ones that reduce them.

This document offers a run-through of the 23 Gang of Four design patterns recorded in the 'Elements of Re-usable Object-Oriented Software' book.[3] Section II, III and IV cover these patterns, sectioned, based on their purpose. Each section is similar in its organization and discusses up to seven key points that wraps around the four basic elements of a pattern. The first element of a pattern is a pattern name used to identify it and have its vocabulary. The second element is a problem which gives an idea about the applicability of a pattern. The solution is a third element which describes design, relationships, responsibilities and collaboration between the classes present in the part. It however is not a concrete design or implementation but a template that can have different use cases. Finally, consequences include the advantages, disadvantages and the trade-offs of a pattern.

Section II on creational patterns is related with the object creation process where a class creation type pattern uses inheritance for instantiation process and an object creation type pattern uses delegation in the object creation process. Section III on structural pattern deals with the composition of classes or objects. Like with the creational patterns the structural class-creation patterns use inheritance to compose interface while object-patterns uses composition to obtain a new functionality. Section IV is about behavioural patterns which is used to characterize the interaction between classes and distribution of responsibilities among them.[2] Section V offers a brief discussion on inheritance and composition and the classification of patters based on object-oriented concepts.

## II. Creational Patterns

Creational design patterns aim to decrease the complexity of a software design by creating objects in a controlled manner. It removes the onus of instantiation from the client to a different class. There are at least, two commonalities between every creational design pattern – encapsulation of knowledge of concrete classes in the system and how the object itself is created. Essentially, the only thing the client knows about the object is its interfaces as defined by abstract classes. In other words, creational design patterns decouple the client from the actual initialization process. This section discusses the five creational design patterns, with focus on complexity, flexibility and overhead of using them how the system is parametrized in each of them.

### *A. Abstract Factory*

#### *1) Intent*

The intent of the abstract factory pattern is to offer an interface for creating a family of related objects, without specifically providing the class names. [4]

#### *2) Motivation*

The Abstract Factory pattern is suitable for instances when family of different products are to be created without putting the onus on the client to set the rules. Consider a furniture shop that sells both Victorian and Modern Furniture. The furniture includes chair, coffee-table and sofa. An abstract furniture factory's interface can be used to create chairs, coffee tables and sofas by using one of the two factories – *VictorianFurnitureFactory* or *ModernFurnitureFactory.*

#### *3) Applicability*

There are situations when products from same family must be used together. Abstract factory pattern allows this by imposing creation of same family product at the same time. Besides, this pattern is applicable in situations where implementation of product is to be hidden as well as when a system demands independence between creation, composition and representation of its products.

*4) Structure*

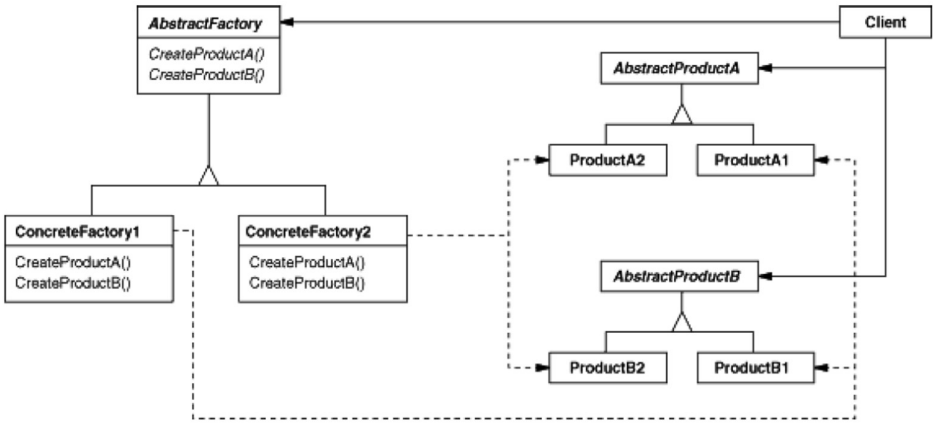



Fig. 1. Abstract Factory Pattern Structure

*5) Consequences*

Since the abstract factory separates out the control and creation of products, it isolates the client from the actual implementation. The client can easily change a concrete factory it uses since it only appears once i.e., when instantiated. Abstract Factory enforces creation of product objects from one family at a time thus maintaining the consistency between them. Also, extending abstract factories to add new product family can be tedious especially if there are a lot of concrete products.

*6) Implementation*

An application usually requires single instance of the ConcreteFactory class per group of products. Thus, factories can be implemented as Singleton. Changing factory to support creation of new products is not easy but can be solved by using a flexible but a less secure approach i.e., by parametrizing the MakeProduct method which identifies the type of the product method.

*7) Example*

The example is of a furniture factory that creates Modern and Victorian furniture. The type of furniture it creates include a chair, a coffee table and a sofa. An abstract furniture factory provides an interface to create chair, coffee table and a sofa and is implemented by a Victorian Furniture Factory and a Modern Furniture Factory. Once one of the two factory methods are instantiated the client can simply invoke a method on them to create whatever the type of furniture that is available by being able to know that the objects return will be of the type defined by the factory that they have instantiated. So, by instantiating Victorian Furniture a Victorian chair can be created by invoking create chair on that object.

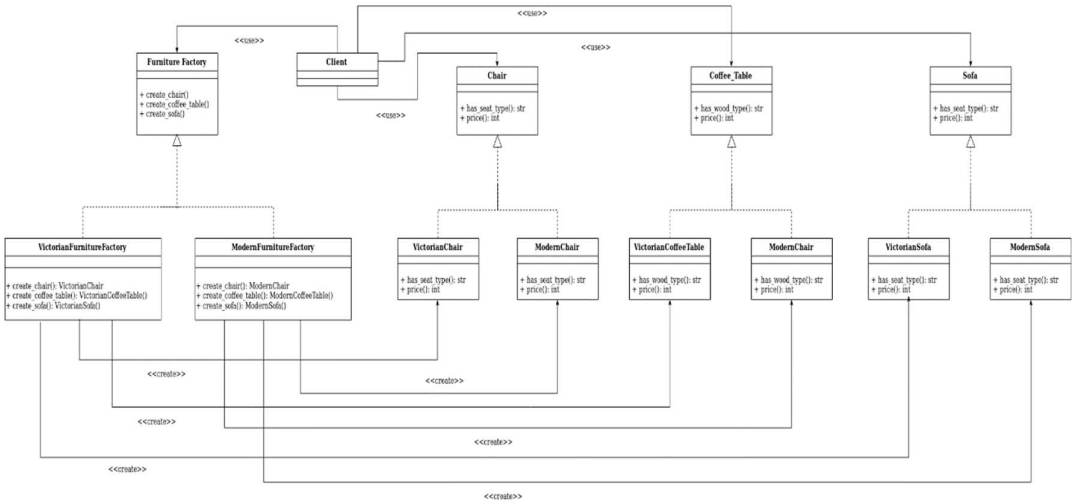



Fig. 2. Abstract Factory Pattern Example

*B. Builder*

*1) Intent*

The intent of the builder pattern is to build a complex object such that a same set of construction steps can be used to create different object representations.[4]

*2) Motivation*

Consider an example where a house is to be built. Each house can be built different by building different components of the house separately using different build methods. A set of these build methods can be called by a constructor class that would call all those builder methods specific to that type in an order and return a built house object.

*3) Applicability*

The builder pattern is used when the information about creation of a complex object is to be separated from the parts it is formed using. It is also applicable where there can be different implementations/interfaces of an object's parts.

*4) Structure*

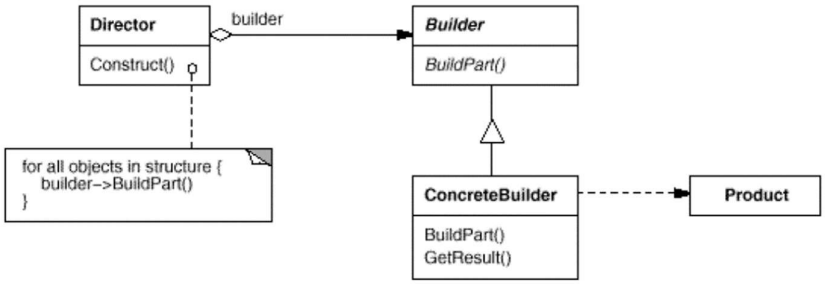



Fig. 3. Builder Pattern Structure

*5) Consequences*

Builder patterns let you vary a product's internal representation. The director has only access to an abstract interface and so internally it knows nothing about how the product is structured or assembled. This allows creating a new kind of builder without much effort and any burden of implementation on the director. Secondly, since the information about the way a complex object is constructed and represented is encapsulated, knowledge of all the classes that participate in the product's construction need not be revealed.

*6) Implementation*

The builder class interface should be flexible so as products for all kinds of concrete builders can be created. Also, it should be accommodating enough so that access to parts of the product created earlier is possible. The products usually differ a lot in the representation so much that they don't need an abstract interface. Also, the client knows the concrete builder that it is using and so it can handle its products accordingly.

*7) Example*

Consider a house as a final product that is to be built. A house builder defines all the steps necessary in order to build a house. It returns a house object using gethouse method. Now this house can be constructed using different kinds of concrete house builder classes which implements the original abstract house builder. That said, now a director in the form of a civil engineer knows how to put together these different steps from the builder to construct a house. A construct house function is invoked to construct the house.

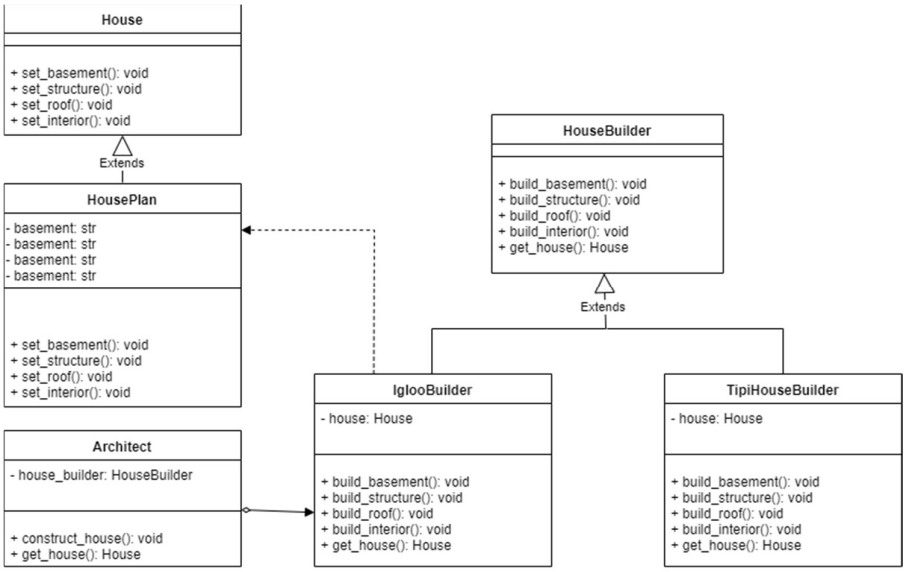



Fig. 4. Builder Pattern Example

## C. *Factory Method*

### *1) Intent*

The intent of the factory pattern is to provide the interface for creating an object but letting the subclass choose the object's concrete type. [4]

### *2) Motivation*

Often an application or a client isn't aware of which subclass object to instantiate from a set of inherited subclasses and so it uses the parent class' interface to demand an object. Based on the context of the application, expansion of requirements or enhancements or some configuration settings, the factory method instantiates the required object and returns it as an instance of the parent class type.

### *3) Applicability*

The factory design pattern is applicable when a client isn't aware about the object it should create or when a class wants to delegate the object creation responsibility to its subclass. It is also helpful in a situation where the knowledge of which helper subclass is the delegate is to be localized.

### *4) Structure*

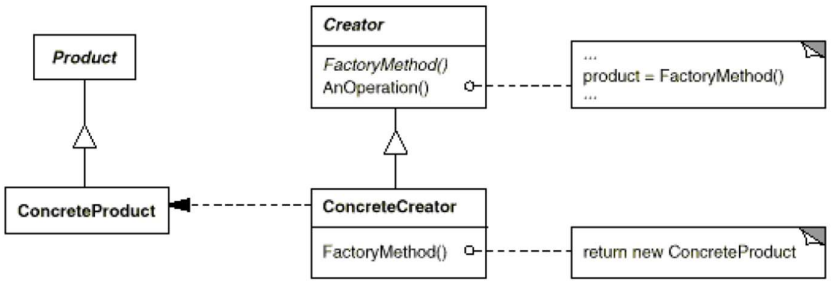



Fig. 5. Factory Pattern Structure

### *5) Consequences*

As consequences, two major advantages of the factory pattern are that the client is only dependent on the interface and doesn't need to specify the class name or worry about the details of the class' object making it more flexible. However, because of the factory pattern clients might have to subclass the creator class just to create a ConcreteProduct object.

*6) Implementation*

There are two variations in the implementation of the factory design pattern – one where the creator class is abstract and doesn't provide an implementation for the factory method and the other one where the creator itself is a concrete class and provides a default implementation of the factory method. As one more variation, the creator class can be parametrized with a type that is used to create a product from a family.

*7) Example*

A SongSerializer factory provides a method to serialize a song just by parametrizing the method with the serialization format. The SongSerializer is a parent class which is overridden by different serializer classes namely JSON Serializer and XML Serializer. When the SongSerializer method is called with a serialization format it returns one accordingly. For example, by providing a song and format name as JSON, an object of the JSON serializer class gets returned. A get json method can be called on the received object to get a json serialized song.

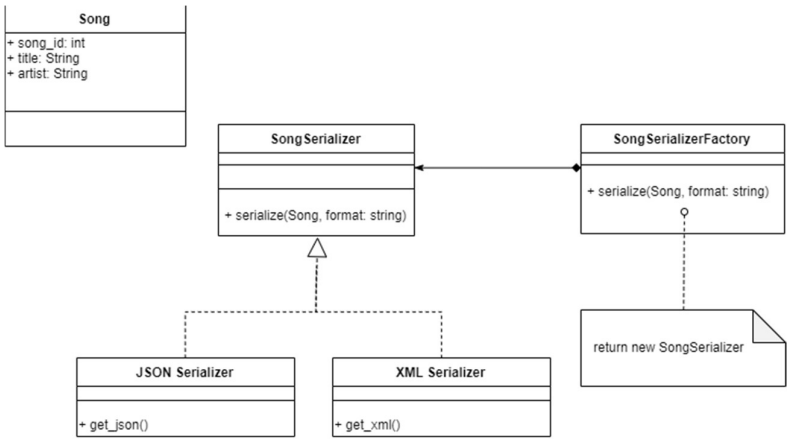



Fig. 6. Factory Pattern Example

*D. Prototype*

*1) Intent*

The intent of the prototype pattern is to specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.[4]

*2) Motivation*

There are times when the process of object creation is expensive and doing that for many objects can worsen the system performance considerably. The prototype design pattern clones an object rather than creating it. It allows one object to create and customize objects without completely knowing the internal details of how that object is created.

*3) Applicability*

A prototype pattern is applicable in a situation where object creation is costly. It is also useful when the classes to instantiate are specified at run-time for example in dynamic loading. Another situation is when instances of a class can have one of a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with an appropriate state.

*4) Structure*

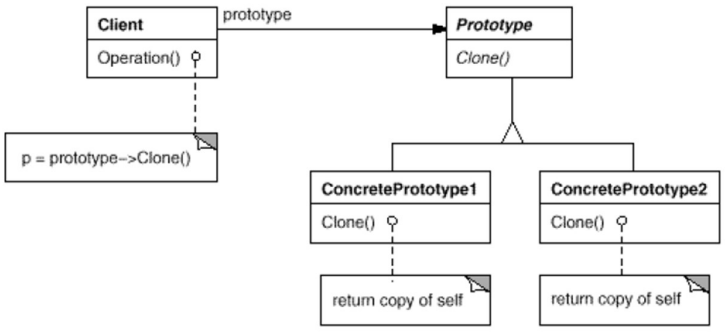



Fig. 7. Prototype Pattern Structure

*5) Consequences*

Because of this pattern, we see that a client can exhibit a dynamic behaviour by delegating responsibility to the prototype. Adding and removing products at run-time simply by registering or de-registering a prototypical instance with the client provides the pattern with right flexibility. However, a major liability of the Prototype pattern is that since each subclass of Prototype must implement Clone operation, implementing it when creation of an object doesn't support copying or have circular references can become quite difficult.

*6) Implementation*

One implementation issue that must be considered while cloning is an object with a circular reference. It is possible to have a deep copy of the object containing a pointer to somewhere but then not having that object point back as in case of a circular reference. Thus, its implementation is little tricky. While implementing clones, you must consider their initialization as not all client might like the exact deep copy of a prototypical instance. In such a case, initializing the object structure using a clone interface might look like an obvious choice at first but since different prototype classes can take multiple parameters it precludes a uniform cloning interface. Depending upon, how the prototype classes are set up, an initialization operation may or may not be introduced

*7) Example*

A prototype class provides a clone operation to create a deep copy of the object. It provides getters and setters for different objects contained by it including a circular reference object. The clone operation creates a copy of each of these aggregated objects assigns them to its own private variables. It also handles the circular reference object by setting that object's reference to itself. And then finally returns a deep copy of itself.

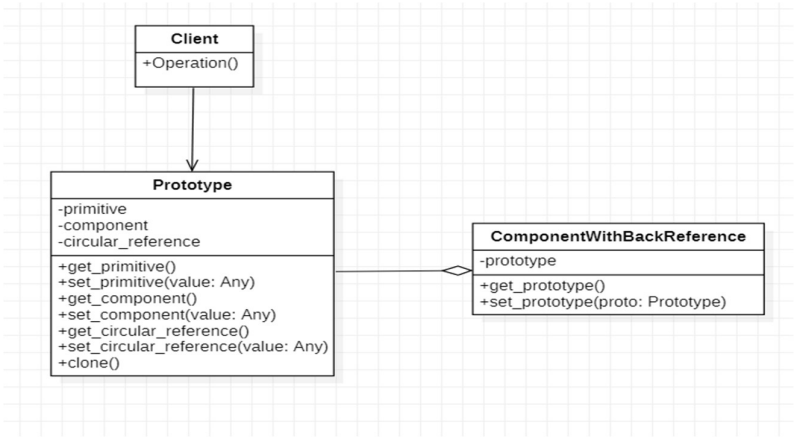



Fig. 8. Prototype Pattern Example

*E. Singleton*

*1) Intent*

The intent of the singleton pattern is to always allow only one instance of a class to be created and then provide a global point of access to it.[4]

*2) Motivation*

Sometimes, we just want a single object of a class in the system. For example, while dealing with objects in the universe we only want one universe object. Having multiple universe defeats the whole purpose. As another example, there can be different connections to a printer in a system but there is always one printer spooler that can be accessed at a time.

*3) Applicability*

This pattern is applicable when there must be exactly one instance of a class and it must be accessible to clients from a well-known access point. It is also useful when the sole instance should be extensible by sub-classing, and clients should be able to use an extended instance without modifying their code.

*4) Structure*

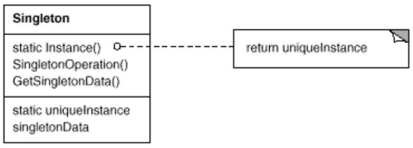



Fig. 9. Singleton Pattern Structure

*5) Consequences*

Since the singleton pattern encapsulates its sole instance, it is much easier to control the access to that instance. There is a lesser namespace pollution using this pattern versus using global variables that store these instances. It is easier to change the singleton class to a normal class. However, using a static function to realize a singleton pattern means the subclasses cannot override them polymorphically.

*6) Implementation*

We can use a static method to return a static instance of the singleton object by creating the object only when getInstance method is invoked which is also known as 'lazy evaluation'. However, this method can be a problem in a concurrent application which can be solved by early evaluation i.e. creating the instance before the threads ask for one. While sub-classing a singleton class, one approach takes the implementation of the getInstance out of the parent class and puts it in the sub-class whereas the other approach instantiates a singleton based on some pre-set context/environment which essentially utilizes factory design pattern.

*7) Example*

This pattern is implemented in python and thus has a slightly different interface. The Singleton class has a private inner class called 'Implementation' class that return the id of the instance once it is created. As soon as a singleton's initialize method is invoked a flag indicating instantiation of a Singleton is set to the id of the first object. Upon requesting another object, the same id is always returned ensuring that the application refers to the same object even though, other object might have actually been created internally the client doesn't have any access to it.

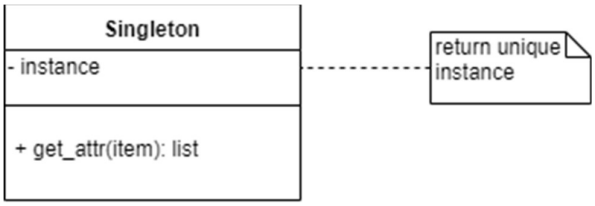



Fig. 10. Singleton Pattern example

*F. Discussion*

While sometimes creational design patterns fit the context of an application more than the others, other times patterns can be used in each other's conjunction to solve a problem. An abstract factory, a builder pattern and a prototype pattern, all of them, create a factory object. Whereas abstract factory is used to create factory objects of several classes, builder is used to build a complex factory object using a complex protocol. In the Prototype pattern, a factory object is created by copying a prototype object. An abstract factory object can be built using a factory pattern as well as be complemented with a prototype by storing a set of prototypes using which objects can be cloned and returned.[5] A builder pattern can be used with the rest of the creational patterns to implement component building. Finally, each of abstract factory, builder and prototype can use singleton objects in their implementation. The choice of a creational design pattern depends on many factors. It is easy to extend a factory pattern by adding new subclasses. However, a factory may generate unnecessary overhead as subclasses proliferate. Abstract factory could be an improvement over a factory pattern if a factory hierarchy already exists. Prototype pattern reduces the number of classes but implementing a clone operation as observed can be little complicated.[6] It is often seen with creational patterns is that the factory pattern is started with and depending upon the requirement of the system other patterns are incorporated.

## III. STRUCTURAL PATTERNS

Structural patterns are a way to set up objects and classes into larger structures in a flexible and efficient manner. In other words, they are concerned with how the classes and objects are composed to form larger structures. They describe ways to compose objects to realize new functionality rather than composing interfaces or implementations. The fact that object composition can be changed at run-time is a reason for the flexibility, which is impossible with static class composition. This section describes seven structural patterns each with a focus on enhanced reusability, decreasing complications by providing elegant and simple interface and increased efficiency.

*A. Adapter*

*1) Intent*

The intent of this pattern is to convert the interface of a class into another interface that the clients expect. The pattern lets otherwise incompatible interfaces to work together. [4]

*2) Motivation*

Sometimes a class that's designed for re-use isn't reusable only because its interface doesn't match the domain-specific interface an application requires. For example, in a drawing editor which lets user draw and arrange

different shapes implementing text can be quite difficult. However, if there is already an implemented class that provides a partial functionality but is incompatible with client's interface then adapter pattern can be used.

*3) Applicability*

Adapter pattern is especially useful when there is a need to use an existing class, but its interface doesn't match the one needed. In a formalized manner, it is applicable when a target interface cannot be changed but a legacy/adaptee provides the operations that the target interface wants to expose. An adapter matches that requirement between a target and an adaptee interface.

*4) Structure*

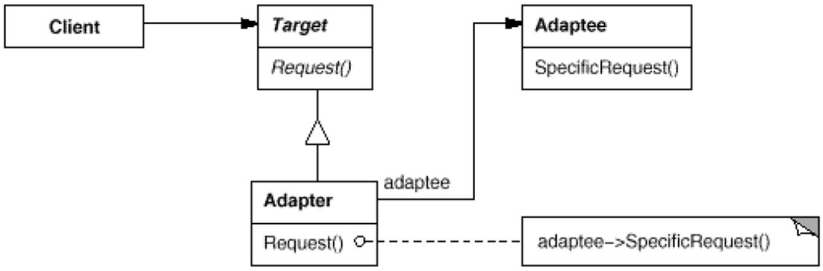



Fig. 11. Adapter Pattern Structure

*5) Consequences and Implementation*

A class adapter uses inheritance over composition in that it inherits from both the target and adaptee and invokes requests for target to adaptee. However, since such adapter must commit to a concrete class it cannot support all the subclasses of an adaptee. This problem is addressed using an object that composes an adaptee object and invokes request from the target accordingly. However, an issue with this approach is that overriding the behaviour of the adaptee becomes harder. The choice should be made depending upon the context of the application. As another issue to consider is how much adaptation is allowed since a variety of conversions can be achieved using an adapter.

*6) Example*

In the following example, a drawing editor wants to use a functionality called bounding box provided by an abstract shape class. Now, we also have a legacy class called TextView which has 'get extent' method that provides exactly the same functionality the client that the 'DrawingEditor' wants. However, the client always expects the same interface as that of shape and so an adapter class (text shape class) overrides the target class (Shape) and calls the request method (get extent) of the adaptee class (TextView) to provide a uniform interface to the client. The client still calls the bounding box method but achieves the functionality of get extent function.

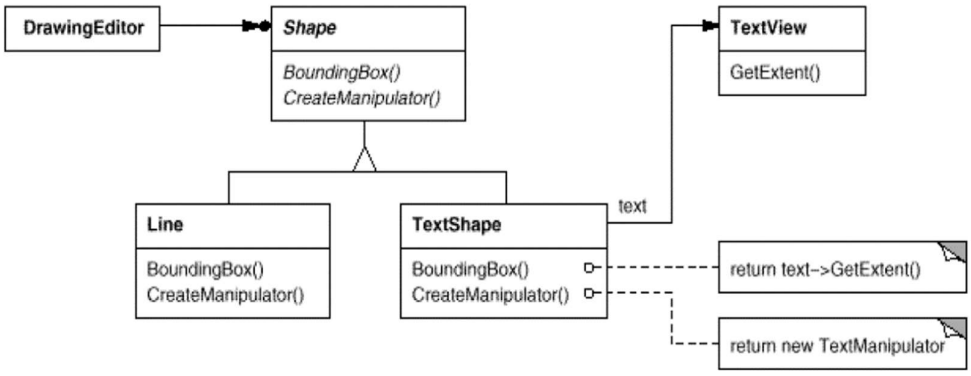



Fig. 12. Adapter Pattern Example

*B. Bridge*

*1) Intent*

The intent of this pattern is to decouple an abstraction from its implementation so that the two can vary independently. [4]

*2) Motivation*

An abstraction can have multiple implementations. Modifying or extending an implementation independent of the abstraction is difficult. In other words, it is difficult to modify, extend or reuse abstraction and implementation independently. Bridge pattern lets you disassociate the implementation from abstraction and vary both independently. In that, the abstraction holds a 'has-a' relationship with the implementor.

*3) Applicability*

The bridge pattern is applicable when there is a need to avoid permanent binding between an abstraction and its implementation. It is applicable in situations where both the abstractions and their implementations should be extensible by sub-classing. It is also an important pattern in situations where changes in the implementation of an abstraction should have no impact on clients i.e., the client code should not have to be recompiled.

*4) Structure*

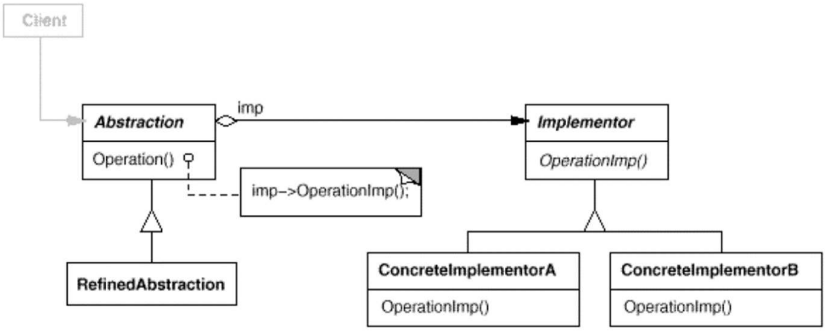



Fig. 13. Bridge Pattern Structure

*5) Consequences*

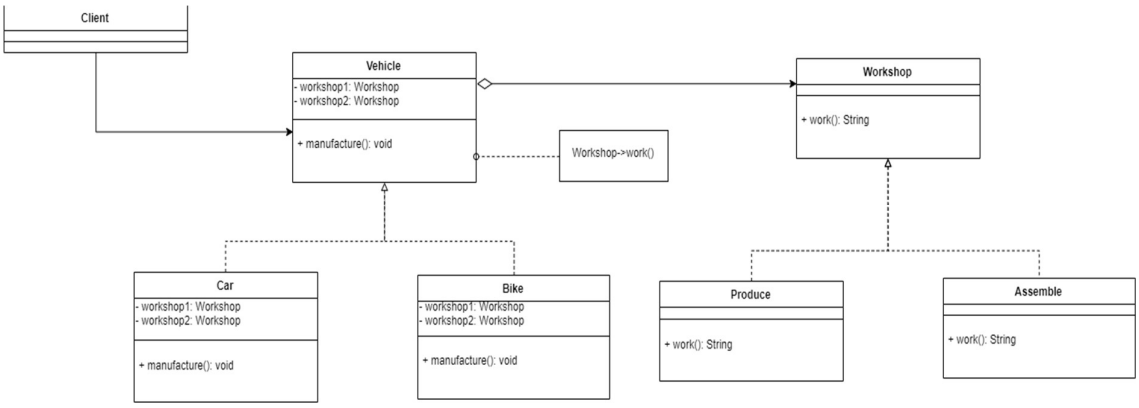
Since in this pattern the implementation is not permanently bound to an interface, the implementation of an abstraction can be configured at run-time. In fact, it also allows an object to change its implementation at run-time. Bridge pattern provides improved extensibility by allowing abstraction and implementor hierarchies to evolve independently. Using this pattern, the implementation details of the implementor can be hidden from the client.

*6) Implementation*

It may not be useful to create an abstract class for a single concrete implementor. The decision to instantiate an implementor object can be delegated to another object altogether thus allowing the abstraction to be not coupled directly to any of the implementor classes. Also, instantiation of implementor can be taken based on some parameter passed to the constructor of abstraction class. Multiple inheritance can be used by a class to inherit publicly from an abstraction and privately from a concrete implementor. This however might lead to a permanent binding of implementation with interface thus hindering realization of a true bridge.

*7) Example*

In the bridge pattern, an abstraction holds a reference to its implementation. In the following example, the Vehicle class is abstraction which holds a reference to the workshop class' concrete implementation as workshop1 and workshop2. The Car and the Bike are called the refined abstraction as they implement the original abstraction. Produce and Assemble are called the concrete implementation of the workshop (implementor). Depending upon which class is instantiated when the manufacture method is called, the work methods i.e. the operations are called on the Implementor's concrete classes. The vehicles and the workshop can evolve independently.




*C. Composite*

*1) Intent*

The intent of this object is to allow clients to treat individual objects and compositions of the object from the same class uniformly. The objects are composed into tree structure to represent a part-whole hierarchy.[4]

*2) Motivation*

Sometimes primitive objects i.e. the object which has no object composed within itself and container objects i.e. objects which contains other objects must be represented differently even if most of the time the user treats them identically which makes the application complex. Composite pattern provides a way to treat these objects uniformly from the client side.

*3) Applicability*

This pattern is applicable when objects must be composed recursively and there should be no distinction between individual and composed elements i.e. objects in the structure can be treated uniformly. The other way to describe the applicability of this design pattern is when an application has a hierarchical structure and needs generic functionality across the structure or when an application needs to aggregate data across a hierarchy.

*4) Structure*

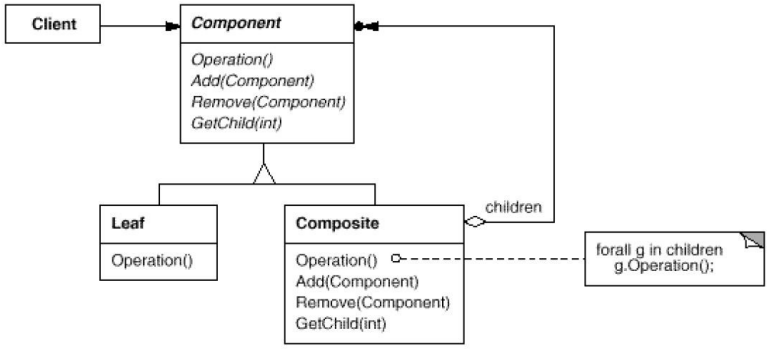



Fig. 14. Composite Pattern Structure

*5) Consequences and Implementation*

The components are treated the same irrespective of its composition thus providing the consistency in how clients treat the objects of that class. Since the client treats the object uniformly, sub-classing to add new components is much easier and thus this pattern is flexible. Sharing a component inheriting from multiple parents to reduce storage requirements is difficult. Sometimes composites should have only certain components but with this pattern you can't rely on the type system to enforce those constraints.[7]

*6) Example*

The idea behind the composite pattern is to treat primitive as well as complex objects uniformly. In the following example, a 'employee class' contains a method to show an employee's details. This class is extended by two leaf nodes i.e., a developer and a manager class which provides the same method to show employee's details. A composite class called company directory also implements the same interface as that of the employee class. But it is also composed of other objects such as a developer object and a manager object. If a show employee details method is called on a company directory, details of all the employee should be listed out. The client as it is seen in

this pattern can treat the company directory object in the same way as a developer or a manager object and therefore the objective of this pattern is realized.

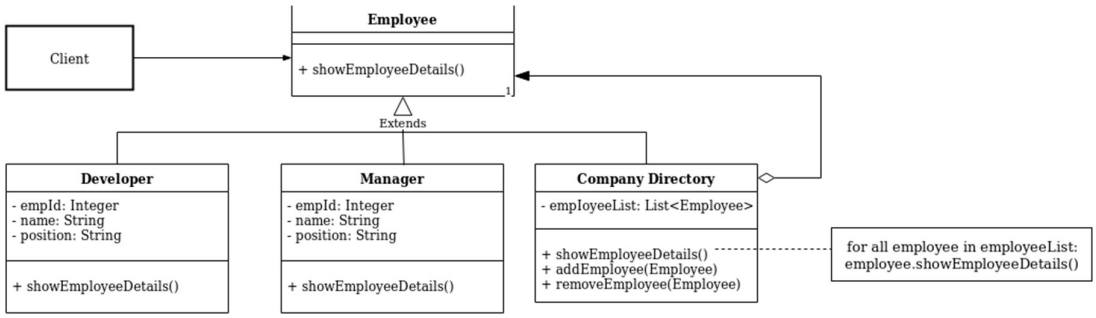



Fig. 15. Composite Pattern Example

*D. Decorator*

*1) Intent*

The intent of this pattern is to attach additional responsibilities to an object dynamically. The decorators provide a flexible alternative to sub-classing for extending functionality.

*2) Motivation*

Sometimes we want to add behaviour to an object dynamically rather than at the compile time which can be done using inheritance. Another requirement could be that the behaviour is to be attached to an object and not an entire class. The decorator pattern provides that flexible alternative.

*3) Applicability*

This pattern is applicable when a system requires an additional responsibility(ies) to be attached to individual object dynamically and transparently without affecting other objects. It is applicable in cases where extension using sub-classes is impractical.

*4) Structure*

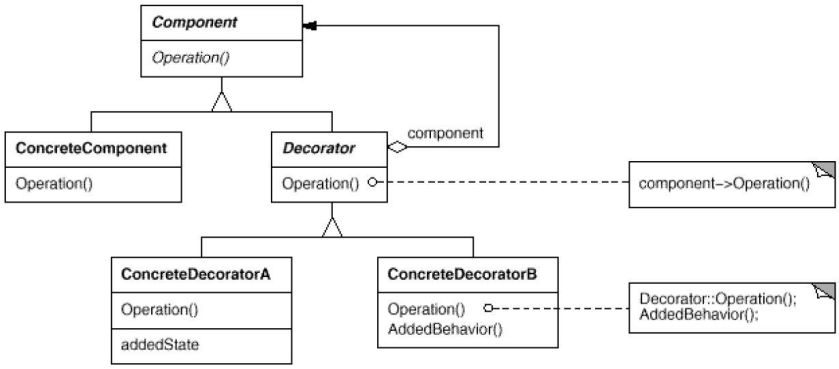



Fig. 16. Decorator Pattern Structure

*5) Consequences*

As one consequence of this pattern, there is no need to add features high up in the hierarchy. Since static inheritance wouldn't any behaviour to be added to the original structure, this pattern which allows run-time behaviour change is flexible. However, it is possible to confuse the decorator pattern with the composite pattern since there is some form of composition in both the patterns. And hence, the object identity shouldn't be relied on while using decorators. It is possible to plague the system with lots of light-weight decorators by losing track of them. This makes the system hard to learn and debug.

*6) Implementation*

For the decorator to work, both the component it is decorating and itself should have the same interface. This means that both the components and the decorators must descend from a common component class. If only one

responsibility needs to be added to the component, the need for having an abstract decorator class should be done away with.

*7) Example*

Consider a pizza formed with multiple topping a pizza can be formed with multiple toppings. Not all toppings may be required for every pizza. Also, the price of toppings may vary which should reflect in the pizza. Rather than inheriting such toppings for different kinds of pizza, a topping decorator with a behavior can be added to a pizza object.

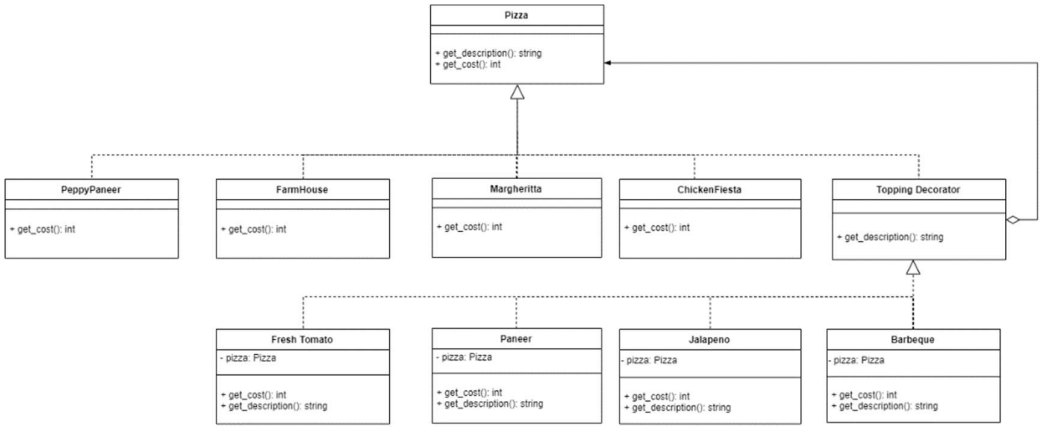



Fig. 17. Composite Pattern Example

## *E. Facade*

*1) Intent*

The intent of this pattern is to provide a simple wrapper around a complicated subsystem. In other words, it provides a unified interface to a set of interfaces in a subsystem.[4]

*2) Motivation*

Structuring a system into subsystems helps reduce complexity. One important target of any design is to reduce the dependencies and interaction between subsystems. This can be achieved by placing a façade object as an entry point that provides a simple interface to access the subsystem [1]

*3) Applicability*

The façade pattern is applicable when a simple interface to a complex subsystem is required. It is also useful in systems where there are many dependencies between clients and the implementation classes of an abstraction. Another utility of façade pattern is when subsystems want to communicate with each other. Since an entry point to each subsystem level is defined in this pattern, they can use facades to communicate with each other.

*4) Structure*

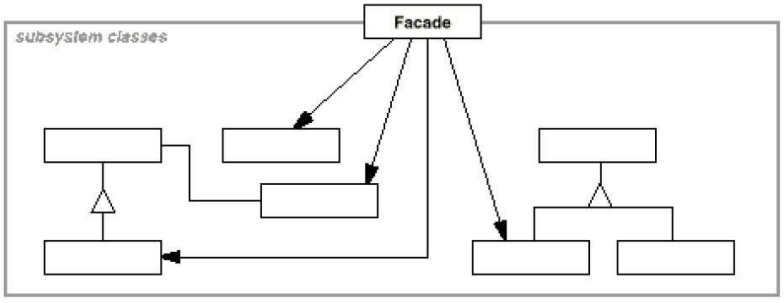



Fig. 18. Facade Pattern Structure

*5) Consequences*

By using this pattern, the subsystems are automatically shielded from the clients, thereby reducing the number of objects that the client has to deal with. This makes the subsystem easier to work with. This pattern promotes

weak coupling between the subsystem and its clients i.e., both the subsystem and the client can evolve independent of each other.

*6) Implementation*

While implementing the façade pattern if the Façade is made an abstract class then that can reduce the client-subsystem coupling even more. One implementation involves keeping the subsystem classes private so that they are not exposed to the clients however not all languages support private classes and hence it may not always be possible.

*7) Example*

The intent of the façade pattern is to provide a wrapper around a complicated subsystem. In the following example, the bank account façade is composed of all the components of a subsystem. It provides a simple interface to a client to get an account number, a security code or to withdraw cash or to deposit the money. Internally, however it calls different methods of different components to perform various checks before spitting out the result to the client. It is called a façade because it isolates the communication between components from the clients and provides an easy to work with interface while managing internal complexities.

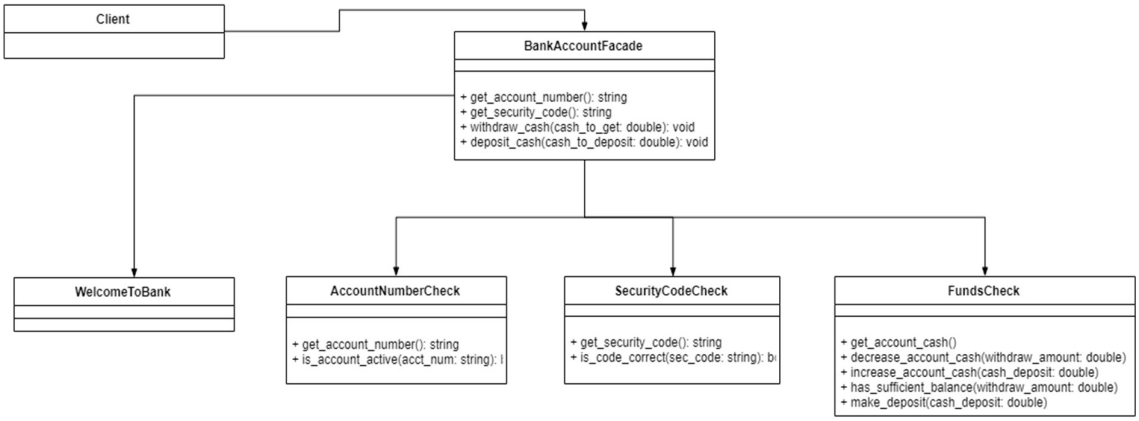



Fig. 19. Facade Pattern Example

## *F. Flyweight*

*1) Intent*

The intent of this pattern is to share objects to support large numbers of fine-grained objects efficiently.

*2) Motivation*

Let's say an object-oriented document editor wants to represent characters as objects to provide greater flexibility. However, this will consume a lot of memory if designed in this way. The flyweight pattern describes sharing of objects to allow their use at fine granularities without prohibitive cost.

*3) Applicability*

The flyweight pattern is applicable when the number of objects that the application has to create is huge. It is useful in situations where creating objects is expensive. However, the classes should be such that their objects properties can be separated into intrinsic and extrinsic properties i.e., there should be some properties that are unique to the object and some that can be shared. If the properties cannot be shared, then the object cannot be shared either.

*4) Structure*

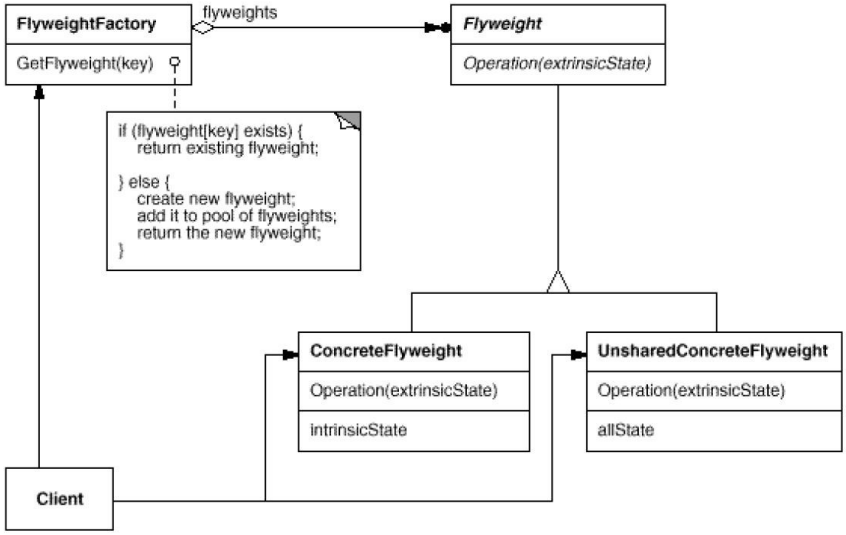



Fig. 20. Flyweight Pattern Structure

*5) Consequences*

The flyweights created in the flyweight pattern do incur run-time costs but for a high number of shared instances there are enough space savings that can outweigh the run-time costs. In this pattern, it must be ensured that in a graph with shared leaf nodes, leaf flyweights cannot store a reference to their parent. Rather, the parent reference is to be passed to the flyweight as part of its extrinsic state.

*6) Implementation*

Superficially, it might look like that removing the extrinsic state would reduce the storage costs however if there are different kinds of extrinsic states as there are object before sharing then this may not be possible. Since the objects are shared, a client shouldn't instantiate them directly rather use a factory that allows the client to look up objects of their interests.

*7) Example*

The Flyweight pattern intends to reduce the same type of objects by maintaining a registry of all the items of different type that have been created. Whenever, a similar object is demanded, rather than creating a new one the one from the registry which is usually a dictionary is returned. For example, in a game there can be two types of players – a terrorist and a counter-terrorist. If there are n players who want to play the game rather than instantiating n player objects, from the dictionary depending upon if the object has already been created the player is returned or created brand new. Now all terrorists will have same mission and counterterrorist will have same mission and hence this can be shared among all players for a given type. A player can have different weapons and hence the client supplies the weapon for a player. The player provides a method to change weapons but the mission and task remains strictly tied with the type of a player.

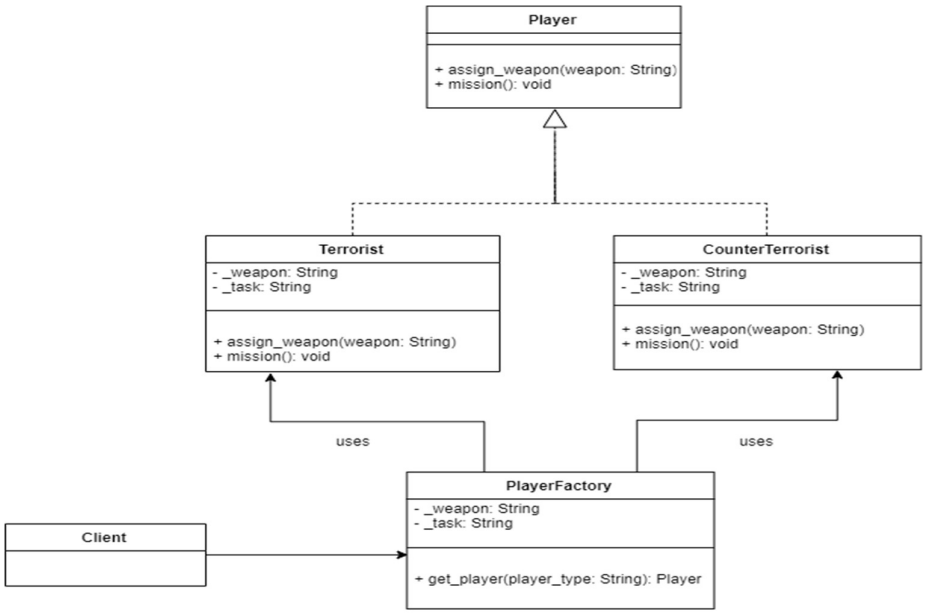



Fig. 21. Flyweight Pattern Example

## G. Proxy

### 1) Intent

The intent of this pattern is to provide a proxy for another object to manage access to it.

### 2) Motivation

At times, we need to be able to administer control of an object. It is possible to delay costly operations and perform them only if required completely. Till that period, another interface can provide a light-weight access thus saving those costly operations. Such accesses are called proxy access and the related objects are called proxy objects. These objects will perform such costly operations only if necessary.

### 3) Applicability

This pattern is applicable for at least three different types of proxies namely virtual proxy, remote proxy and protection proxy. A virtual proxy is used for delaying the creation and initialization of expensive objects until needed, where the objects are created on demand. A remote proxy is applicable where a local representation for an object in another address space is required. The proxy pattern is also applicable where in access to the RealSubject methods are controlled by giving access to some objects while denying access to others.

### 4) Structure

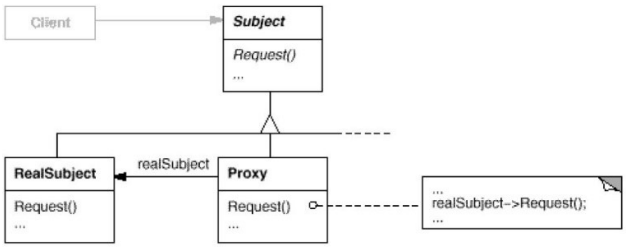



Fig. 22. Proxy Pattern Structure

### 5) Consequences

A level of indirection while accessing an object is introduced as a consequence of this pattern. The pattern can however show a disparate behavior of the clients are accessing both RealSubject as well as the Proxy classes' objects.

*6) Implementation*

In this pattern, the proxy object doesn't have to necessarily know all the types of RealSubject i.e., if it can deal with the object through a uniform abstract interface. One implementation issue that can arise with this pattern is on how to refer to a subject before it is instantiated. This essentially means that some form of address space-independent object identifiers must be used to refer to them.

*7) Example*

The proxy pattern provides a proxy for another access to manage access to it. In the following example, both the Real Internet and Proxy internet share the same interface with an abstract Internet class. The ProxyInternet however is lightweight and doesn't have all the responsibilities of RealInternet class. Rather than directing the request from the client to directly connect to the real internet, by calling a proxy internet few checks can be made regarding the request. In this case, the proxy internet checks if the request is made to a banned site in which case it doesn't need to forward the request to the real internet. This is essentially introducing a level of indirection. The ProxyInternet is a proxy object here.

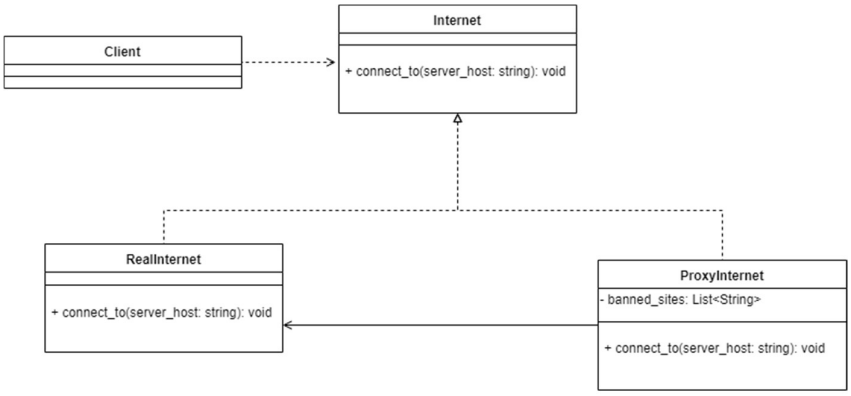



Fig. 23. Proxy Pattern Example

*H. Discussion*

The structural patterns look similar based on the classes or objects that are part of a design and how they perform their responsibilities. However, they differ in their intent. For example, adapter and bridge both introduce a level of indirection by forwarding requests to the object from a different interface. However, an adapter intends to resolve the incompatibilities between interfaces whereas bridge bridges an abstraction and its many implementations. Adapter is generally used to make two class work together without replicating code when the coupling is not decided whereas a bridge client understands that there are several implementations and the abstraction and implementation can vary independently. Similarly, composite and decorator also look similar because both patterns use recursive composition however, they are different in their intent as well. A decorator allows adding behavior dynamically to objects without sub-classing and thus saving the subclass explosion in trying to cover different combinations. The composite pattern is primarily aimed at treating primitive objects and objects with deeper hierarchy in a uniform manner. In fact, decorator pattern can be used to complement composite pattern. Again, both proxy pattern and decorator pattern look similar because they compose an object and provide an identical interface to clients but differ in intent. The intent of the proxy pattern is to fill in a proxy for a subject which has either restricted access or is persistent or inconvenient to access. Although rare, it is possible to have a proxy-decorator hybrid pattern.

## IV. BEHAVIORAL PATTERNS

*A. Chain of Responsibility*

*1) Intent*

The intent of this pattern is to reduce dependencies between a sender of a request and its receiver by giving more than one objects a chance to handle the request.

*2) Motivation*

Often the requestor isn't aware of the object that handles that request. A way to decouple the object that initiates the request from the object that serves it, is the motivation behind using a pattern like chain of responsibility. As the name suggests, the request gets passed along a chain of objects until one of them handles it.

*3) Applicability*

The chain of responsibility is suitable when more than one object can handle the request, but the handler isn't known beforehand. This can be extended to say that it is suitable when multiple objects can receive the request without them being specified explicitly. Also, when the set of objects that can handle the request are not pre-specified and have to be selected at run-time, this pattern can be quite handy.

*4) Structure*

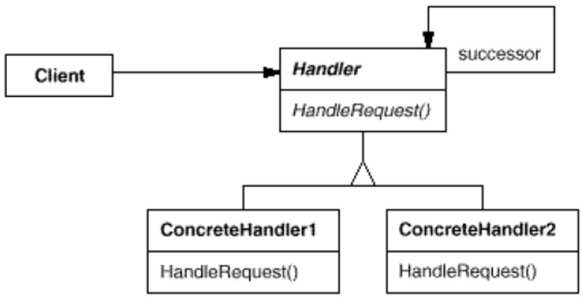



Fig. 24. Chain of Responsibility Pattern Structure

*5) Consequences*

Laying out this pattern means the sender handler object will have reduced coupling i.e. the sender doesn't have to know which object will handle the request. Another advantage of this pattern is flexibility. By adding to or changing the chain, responsibilities can be added or changed respectively at run-time. There is however one issue that can arise i.e. a request may fall off the chain without ever being handled and this pattern provides no guarantee about that.

*6) Implementation*

A successor chain can be implemented by either defining new links or by using the existing links. Although the example in the next section implements a new link, an existing link can be useful when they support the chain that is required. It saves time in specifying the link explicitly and space in generating it. Otherwise, newer links can always be defined. A request handler can provide a default implementation for handling the requests if there are no preexisting references.

*7) Example*

Consider a number object that can be processed as zero, negative or positive by their respective processors. In the following example, there exists a chain of such processors in the order negative processor, zero processor and positive processor. If a positive number is passed it first goes through the zero processor which parses it to see if it can process which it can't and therefore passes it to next in chain the negative processor which again cannot process and therefore is passed off again. Finally, the positive processor can process it. Essentially, in this pattern hooks to next in chain and to process requests are provided. If the request can be handled, then it is processed else the process request of the next in chain is called. The downside of this example and the pattern is that it could be possible that the request doesn't get handled after passing through the entire chain.

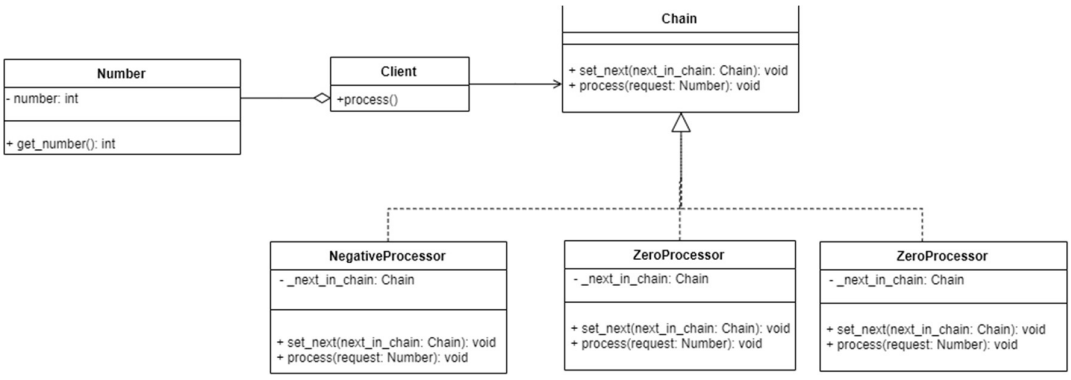



Fig. 25. Chain of Responsibility Pattern Example

*B. Command*

*1) Intent*

The intent of this pattern is to encapsulate requests as an object so that the clients can be parametrized with different requests which would allow operation of otherwise undoable operations.

*2) Motivation*

There are situations when a client wants to make a request but doesn't know anything about the underlying operations or the target of the request. All it wants is a command/request to be fulfilled. Command design pattern basically decouples the client object from the one who knows how to perform it. The commander makes requests about unspecified application objects by turning the request itself into an object. The object can then be stored and passed around until the target who knows what operations to perform on it gets it.

*3) Applicability*

The command pattern is applicable under various situations. It can be used where there is a need for a callback. The command can be encapsulated as an object and then other objects can be parametrized with it. This can be expressed as an object-oriented callback when required. The command pattern can come in handy if specifying, queuing and executing requests has to happen at different times. It is also useful for applications where a previous state can be reached by exposing an undo interface. Command pattern's execute operation can store the previous state which can be retrieved as necessary. Besides it is also useful for loggers.

*4) Structure*

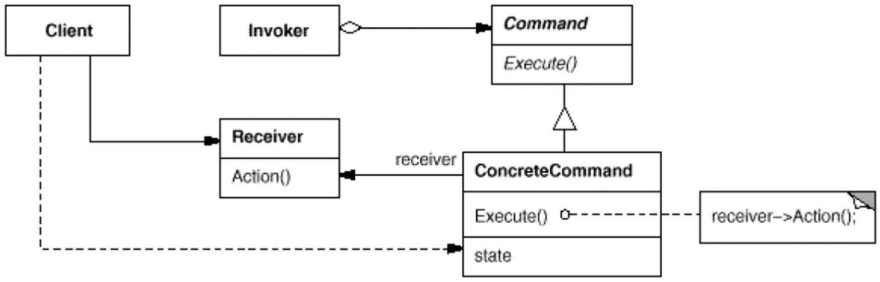



Fig. 26. Command Pattern Structure

*5) Consequences*

The command pattern is a disassociation between the object that invokes the operation from the objects that handles it. Command/Request encapsulated as a request can be manipulated and extended like any other object. The pattern makes it easy to add new commands because the existing classes do not need to be changed.

*6) Implementation*

The implementation of the command pattern allows the implementor to decide how much a command can do. The implementor should avoid making the command just so that it is a link between the target and the actual operation performed or the command implements everything without delegating any responsibility to the receiver. The undo/redo operations can be easily achieved using this pattern however a proper choice should be made to either store the entire snapshots of a program using this pattern or by storing some metadata about the states and implementing some functionality to regenerate the state when required. The composite pattern can be used in conjunction with this pattern to group existing commands in new command while adding a new command. This could basically become macros.

*7) Example*

The objective of the command pattern is to encapsulate commands by converting them into command objects. The commands are set as classes and they inherit and override from an abstract command class an execute function. The commands then hold a reference to an object they are controlling and depending upon which command is called the respective commands on the controlled object are called. In this example, LightOnCommand, LightOffCommand, StereoOnWithCDCommand and StereoOffCommand are the command class that inherit from the command class and implement the execute method. The first two hold reference to the Light class and the last two hold reference to the stereo class. If LightOnClass is instantiated or set as command as in this case using setcommand of the SimpleRemoteControl class, then the button_was_pressed or the execute method in the LightOnCommand class calls on method of the light class object that it is holding a reference to. Similarly, depending upon which command is set with the remote control different objects/devices can be controlled.

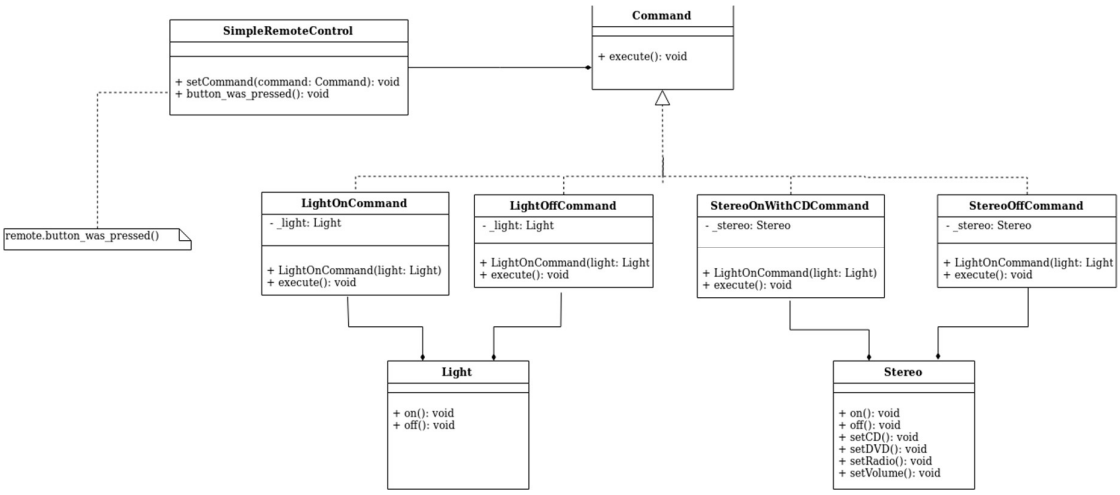



Fig. 27. Command Pattern Example

## C. *Interpreter*

### 1) *Intent*

The intent of the interpreter pattern is to determine a representation of grammar of a given language and to provide an interpreter that can be used to interpret the sentences in the language.

### 2) *Motivation*

Often in the real world, there are a set of problems that occur frequently. These problems are as well a part of well-defined and understood domain. If the domain could be marked with a language, then problems can be easily solved much easily. Interpreter design pattern essentially allows instances of problems to be expressed as sentences and then converting them using an interpreter.

### 3) *Applicability*

The interpreter pattern is applicable in a domain where the grammar is simple and relatively stable. Also, when it is applied efficiency should not be a critical concern.

### 4) *Structure*

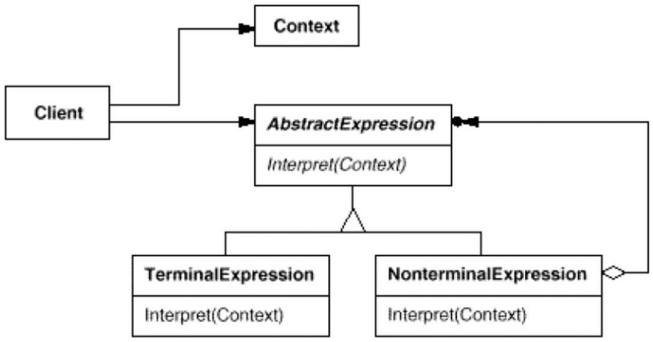



Fig. 28. Interpreter Pattern Structure

### 5) *Consequences*

It is quite easy to change and extend the grammar with the interpreter pattern and thus it provides good extensibility. Implementing grammar for a well-defined domain is easy with the interpreter pattern. However, maintaining grammar could become difficult as the complexity increases in the application. Also, as the grammar rules increase, the number of classes and thus objects increase in the application thus making it storage wise inefficient.

*6) Implementation*

The implementation of interpreter pattern usually involves creating an interpret operation to match the context of an application. Usually, a class is removed out to represent a grammar rule for a given language.

*7) Example*

The interpreter pattern is used to represent the grammar of a language. In the following example, the grammar is parsing boolean operations between expressions. There is an abstract expression class which is extended by three expression namely AndExpression, OrExpression and TerminalExpression. For a given context each of these expression classes have different interpretation. The Terminal expression evaluates/interprets if the contained data (string) is present in a given context (string). The AndExpression and OrExpression are binary operators in a way and need two expression to interpret. The AndExpression interpretes if both the expressions are interpreted as true whereas OrExpression interprets if either of the two expressions are interpreted as true.

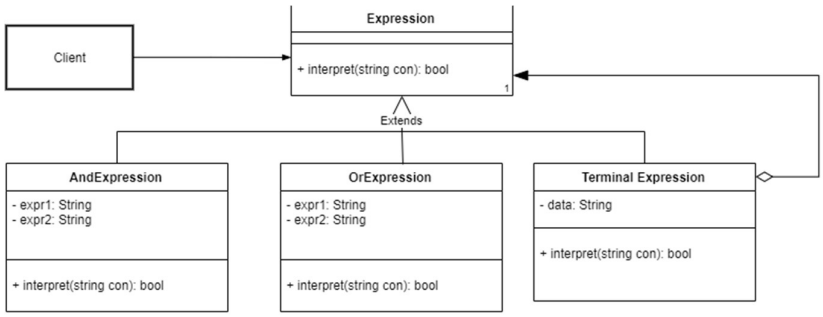



Fig. 29. Interpreter Pattern Example

*D. Mediator*

*1) Intent*

The intent of this pattern is to promote loose coupling by encapsulating the communication method with which two dissimilar objects communicate with each other.

*2) Motivation*

An object-oriented design with different classes interacting with each other requires independence. Such frameworks need a mechanism to facilitate the interaction between objects in a manner that they are not aware of each other's existence. A mediator object realized using mediator pattern can address this issue.

*3) Applicability*

This pattern is applicable when a set of objects communicate in well-defined but complex ways. Mediator pattern provides a way to resolve the unstructured and difficult to understand interdependencies between these objects. Again, if reusability of an object in an application is reduced because of too much intercommunication then this pattern can be quite useful. The mediator pattern can be quite handy when several classes together implement a behaviour and that behaviour should be dynamic.

*4) Structure*

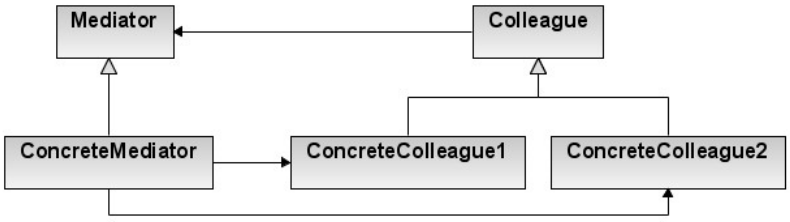



Fig. 30. Mediator Pattern Structure

*5) Consequences*

The mediator pattern localizes behaviour and thus limits sub-classing. It promotes loose coupling between the dissimilar objects acting as colleagues in the system. Another advantage of the mediator pattern is that it simplifies object protocols by replacing many-to-many interaction with one-to-many interactions. Although, mediator intends

to simplify the complicated communication it itself can become more complex than any other colleague and thus trading the complexity of interaction for complexity in itself.

*6) Implementation*

In the implementation an abstract mediator class is not needed. An observer pattern can be implemented to realize the communication between a colleague and a mediator.

*7) Example*

In the following example, IATCMediator is an abstract class which defines the interface for communication between flight and runway. The ATCMediator is the concrete implementation of mediator that coordinates the communication between flight and runway. The command class defines the interface for communication with other colleagues. The flight and runway are the colleague objects which talk to each other view a mediator.

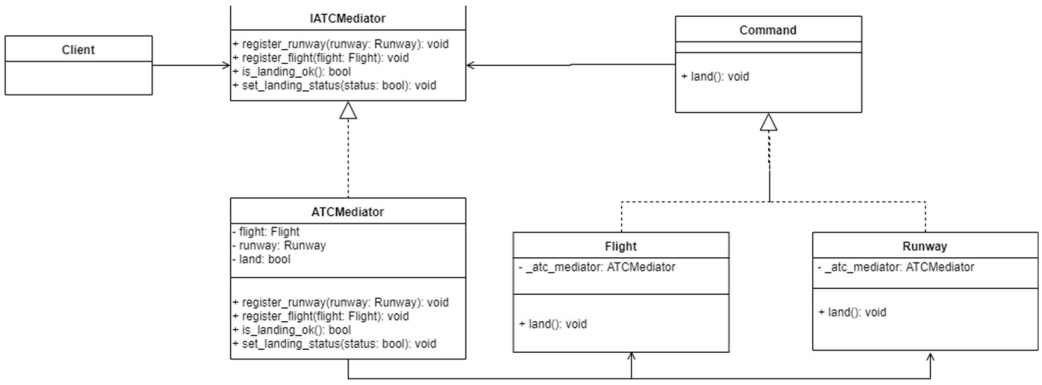



Fig. 31. Mediator Pattern Example

## *E. Memento*

*1) Intent*

The intent of the memento pattern is to capture an object's internal state so that the object can be restored to this state later.

*2) Motivation*

Sometimes the internal state of an object needs to be recorded. Implementing checkpoints and undo mechanisms lets users back out of a tentative operation and/or recover from errors. Saving the state information enable the restoration of objects to their previous states.

*3) Applicability*

This pattern is applicable for a system where an object's state must be saved so that it can be restored later. Also, since an object is encapsulated in the memento pattern, the pattern is applicable where a direct interface to obtaining the state would expose implementation details of the class and break the object's encapsulation.

*4) Structure*

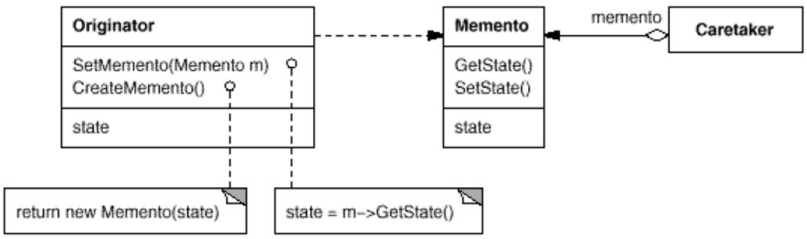



Fig. 32. Memento Pattern Structure

*5) Consequences*

The advantage of memento pattern is that it protects the encapsulation boundaries and therefore prevents exposure of the object's implementation details. A client manages the state of the system rather than the originator which removes the burden of storage management from the originator. However, a challenge with this pattern could

be privacy of a memento's state. Ensuring only originator can access the memento's state be difficult. The pattern can become expensive once mementos are large and high in number or if the frequency of their creation becomes too high.

*6) Implementation*

If the language supports private members then the wide interface of Memento class should be accessible by an originator class but only a narrow interface should be accessible by other classes. This can be achieved by keeping most methods private but accessible to originator whereas a small number of methods made accessible to everyone. If the mementos get created and passed back to their originator in a sequence that can be predicted, then storing incremental changes could be a good way of reducing storage requirements.

*7) Example*

The memento pattern is used to save an object's internal state that can be restored later. In a memento pattern, the originator is an object whose state is to be saved. The memento is an object that maintains the state of the originator A caretaker manages multiple mementos. In the following example, a caretaker manages multiple checkpoints of the originator in the form of mementos. It maintains a list of mementos and whenever backup is called the originator's save method is called which returns a concrete memento referring to the originator's state. If an undo operation is called, then the last memento is popped from the list and the state of the originator is set to the last memento. In fact, if the caretaker class can provide a method to return a memento based on it's id then the state of the originator can be set to that without repeatedly popping the mementos.

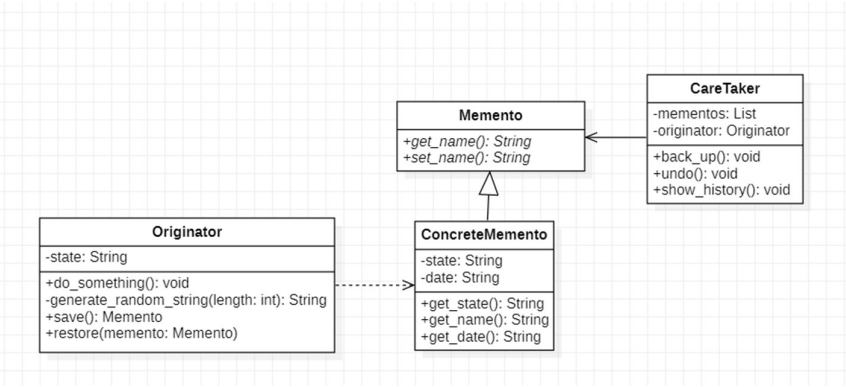



Fig. 33. Memento Pattern Example

## *F. Observer*

*1) Intent*

The intent of this pattern is to create one-to-many dependencies between different objects so that when one object changes state, all its dependents are notified and updated.

*2) Motivation*

Often there are cooperating classes which need to maintain consistency between related objects. However, tightly coupling them reduces their reusability and flexibility. Observer pattern provides a design in which if one object changes state, all its dependents are notified.

*3) Applicability*

The observer pattern is applicable when an abstraction has two aspects and one is dependent on the another i.e. if there are changes in one class then they should be reflected on the other class in some way. Usually, such changes should be reflected to another class independent of how many dependents exist. The pattern is practical for situations where an object should notify other unknown objects.

*4) Structure*

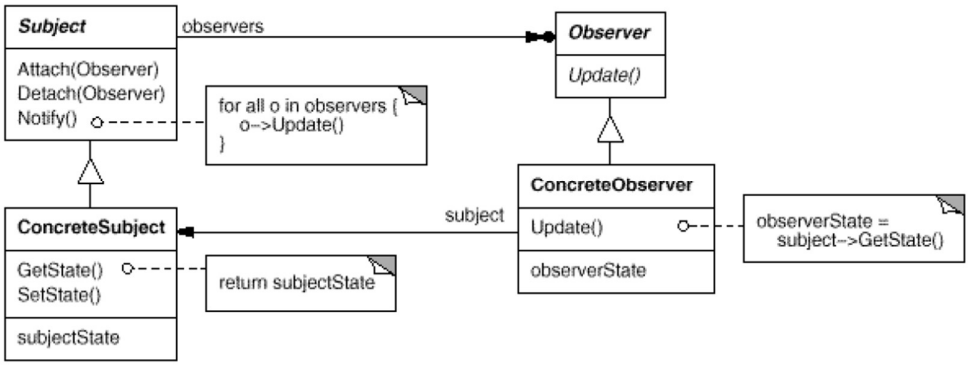



Fig. 34. Observer Pattern Structure

*5) Consequences*

The pattern provides modularity in that the subject and observers can vary independently of each other without any kind of coupling. Observer pattern is also extensible as defining and adding any number of new observers can be handled quite easily by only registering with the subject. Since, different observers can be registered with the subject easily, different observers can provide different views of a single subject. However, managing unexpected updates since observers have no knowledge of each other where an observer may not actually need to receive a type or content of an update or at a time could become difficult. To resolve this, filters can be used which would be an additional overhead.

*6) Implementation*

Storing references to observers in a situation where there are many subjects and few observers can become quite expensive. In the implementation of the pattern therefore a hash table can be used as associative look up to maintain the subject-to-observer mapping. The deletion of subjects should not cause dangling references. This can be avoided by letting the observers know of its deletion. Depending upon how much information is required, a push or pull model should be decided upon. The update efficiency in the implementation can be improved by making the interface that also allows registering observers for specific events of interests.

*7) Example*

The observer pattern is also sometimes known as a publish/subscribe pattern because the observers are updated about subscribers without strong coupling. In the following example, cricket data is a subject class which provides methods to register/attach or de-register/detach observers. It also maintains a registry of observers. The AverageScoreDisplay and the CurrentScoreDisplay are the concrete implementations of the Observer class. Whenever in the CricketData class, runs or wickets change the notify function is called which in turn calls the update method on all the observers and effectively changing their output. This way a change in subject is reflected as changes in the observer.

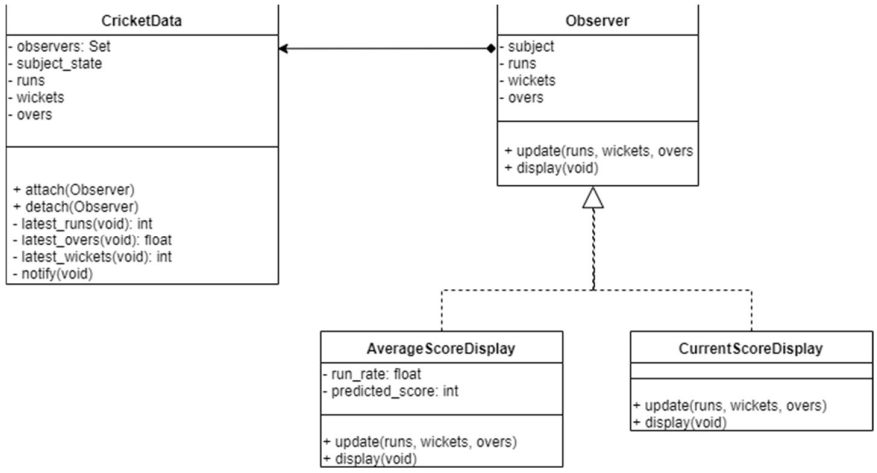



Fig. 35. Observer Pattern Example

*G. State*

*1) Intent*

The intent of the state pattern is to allow an object to change its behaviour based on its internal state changes. With this pattern it appears as if the object is changing its class.

*2) Motivation*

An object can have different states and behave differently upon request from other object depending on its current state. Such behavior is realized using State Pattern

*3) Applicability*

The state pattern is applicable where an object must change its behaviour at run-time depending upon which state it is in. The state pattern is also applicable in situations where several operations share a same structure that depends on the object's state.

*4) Structure*

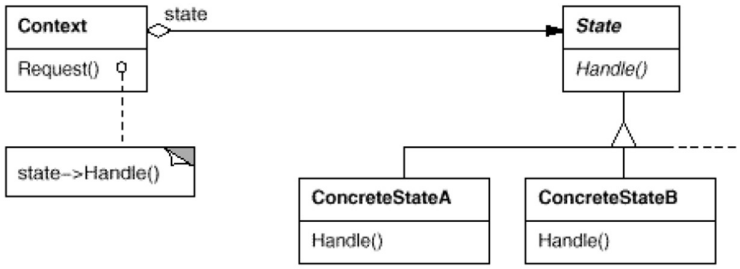



Fig. 36. State Pattern Structure

*5) Consequences*

One consequence of the state pattern is that it abstracts out a state pertaining behaviour and separates the behaviour for different states. Also, the state transitions become more explicit as there are objects that represent states. A consequence that follows it is that the context is protected from inconsistent states by state objects. State objects can be shared as well if they don't have instance variables.

*6) Implementation*

If the criteria for transition is fixed, then it can be implemented in context. However, state subclasses should be allowed to choose their successor state and timing of the transition. In order to change the behaviour of classes at run-time objects can delegate operations to other objects, effectively achieving a form of dynamic inheritance.

*7) Example*

The following example demonstrates how state pattern lets an object change its behaviour based on its internal state changes. The ATMState is an abstract class that is implemented by four different classes which represent different states an ATMMachine can be in. The ATMMachine contains all this four ATMClasses. It provides the client an interface to insert card, eject card, insert pin and request cash. Since it composes all the class representing the states it can be in, based on the input by the client it sets the state of the atm. All the actions then correspond to that action from there on. For example, if a client inserts a card then the state of ATMMachine asks for a PIN and the state get set to HasCard. From there on, if insert_pin is called and if correct the atm state change to HasPin. Finally, in the HasPin state request_cash is called and if there is enough amount to withdraw then the cash is given and the ATMMachine transitions to NoCard mode/state. However, if insert pin were called in the NoCard state it wouldn't make sense and hence a warning message would be printed.

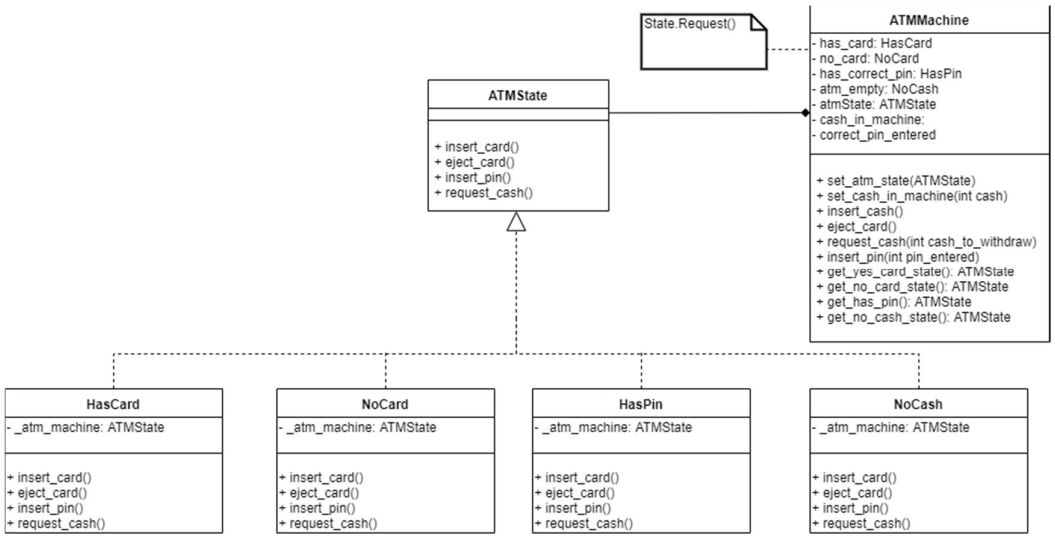



Fig. 37. State Pattern Example

## *H. Strategy*

### *1) Intent*

The intent of this pattern is to define and encapsulate family of algorithms and then make them interchangeable so they can be chosen dynamically. This pattern lets the algorithms also known as strategy independently of the clients that use them.

### *2) Motivation*

Often there are different algorithms that are appropriate at different times for different classes. Hard-wiring such algorithms for such clients can make the code more complex and make it more vulnerable to breaking. Also adding new algorithms or varying them can be a pain if the previous approach is used. These can be avoided by encapsulating the algorithms also known as strategy.

### *3) Applicability*

This pattern is applicable in situation when an object can be configured with different run time strategies. It is also important that these strategies or algorithms can be encapsulated for the strategy pattern to be applicable. It is useful for situations where a data structure that an algorithm uses shouldn't be exposed to a client. Strategy pattern can come in handy where a class wants to depict different behaviours but uses conditionals to do so. Instead, by encapsulating each conditional behaviour into a own strategy class can make the application maintainable.

### *4) Structure*

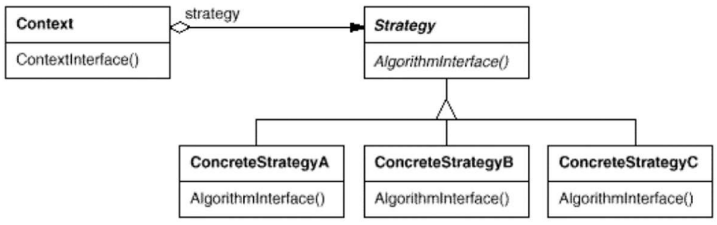



Fig. 38. Strategy Pattern Structure

### *5) Consequences*

The advantage of the strategy pattern is that algorithms can be changed dynamically to depict different class behaviour. In order to do so, it supports a variety of algorithms. Using inheritance can in fact, abstract out common behaviour of algorithms and thus promoting reuse. The strategy pattern also removes the need of using conditional statements from class where dynamic behaviour is desired. This is achieved by encapsulating the algorithm in a separate strategy classes letting you vary, extend or switch the algorithm independent of its context.

*6) Implementation*

Depending upon an algorithm and its data requirements the strategy can request data from context in different ways. Context can parametrize strategy class with some information or context can itself be passed to the strategy which it can call on explicitly or the strategy class can store a reference to the context interface effectively letting strategy request what it exactly needs.

*7) Example*

The idea behind the strategy pattern is to encapsulate family of algorithms and then choose them dynamically. Consider a fighter game in which characters have different moves such as roll, kick, punch and jump. While some characters can have one behaviour another might not. Rather than using inheritance to override the behaviour of an interface that provides these methods, we can pull out this behaviour as strategy and allow characters or context to adopt to it. In the following example, the KickBehavior and JumpBehavior are strategies and the Fighter class is a context. The Fighter is an abstract class which is extended by three characters Ramesh, Suresh and Mukesh. The KickBehavior is an abstract class for one strategy and JumpBehavior is strategy for another class. The LightningKick and TornadoKick implements KickBehavior and ShortJump and LongJump implements JumpBehavior. The Fighter class provides a hook to set a kick as well as jump behaviour meaning a character like Ramesh can set its jump behaviour to Short Jump and its kick behaviour to Lightning Kick.

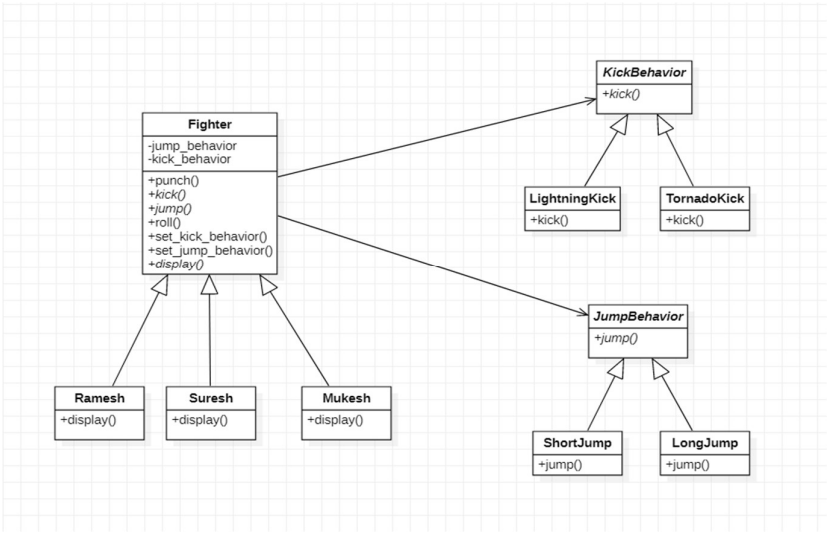



Fig. 39. Strategy Example

*I. Template Method*

*1) Intent*

The intent of template method is to provide a framework for an algorithm in a method, and then letting subclasses specify certain steps of an algorithm without changing its structure.

*2) Motivation*

Often there are situations where the order of steps that an algorithm follows are same even though the implementation of those steps can vary. The template pattern defines algorithms steps using abstract operations and fixes the ordering but lets the subclasses implement those steps thus providing a template as well as allowing different behaviour.

*3) Applicability*

Template pattern is applicable where the fixed parts of an algorithm can be implemented once, and the subclasses are delegated the responsibility to implement the non-fixed behaviour. This pattern is also important because a shared behaviour can be identified and separated out in a common class to avoid code duplication.

*4) Structure*

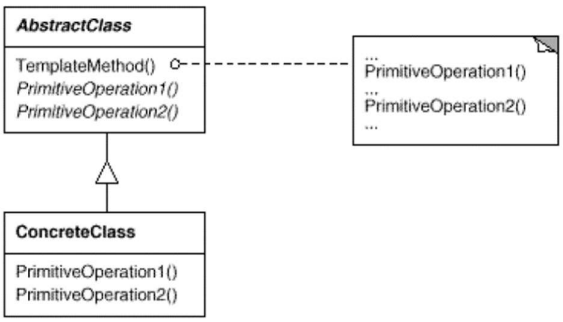



Fig. 40. Template Pattern Structure

*5) Consequences*

The template pattern leads to inversion of control. In this case, the parent class calls the subclass method which basically decouples components and layers in the system. Another consequence of this pattern is that it promotes code reuse. With template pattern, the overriding rules can be enforced. One problem with template pattern is that it forces use of subclasses to specialize behaviour.

*6) Implementation*

In the implementation of the template method an effort should be made to minimize the number of primitive operations that a subclass should override to complete the algorithm. Lesser the overriding operations better it is for the client. A naming convention should be followed to identify operations that needs to be overridden. For example, a prefix 'do-' can be added to a method name.

*7) Example*

The template method provides a structure of operations in the parent class and leaves the details of the implementation to the subclasses. In the following example, the process_order is a template method that calls do_select(), do_payment() and do_delivery() in an order and lets sub-classes like NetOrder and StoreOrder define the meaning of these methods. For example, a net order delivers to the shipping addresss whereas store order delivers over a counter.

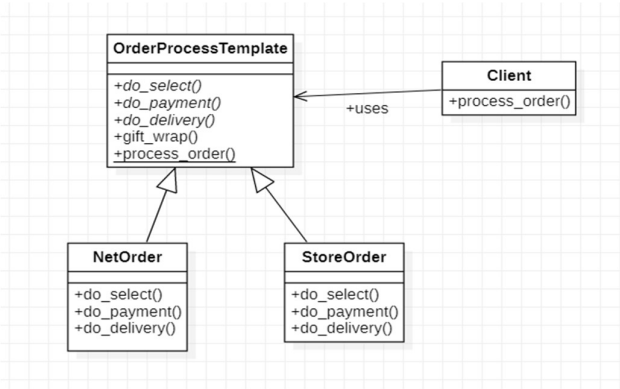



Fig. 41. Template Pattern Example

*J. Visitor*

*1) Intent*

The intent of the visitor pattern is to centralize operations on an object structure so that it can vary independently of the client but still behave polymorphically.

*2) Motivation*

Often adding new operations and distributing operations can become hard to understand, maintain and change. Visitor pattern allows newer operations to be added separately independent of the nodes on which they operate.

*3) Applicability*

The visitor pattern is applicable in situations where unrelated operations need to be performed on an object in an object's structure and to keep these classes clean visitor patterns allows to keep the related operations together

by defining them one class. Visitor pattern is also applicable when an object encapsulates many other objects with different interface, and it is required to perform operations on the objects which depend on their concrete classes.

*4) Structure*

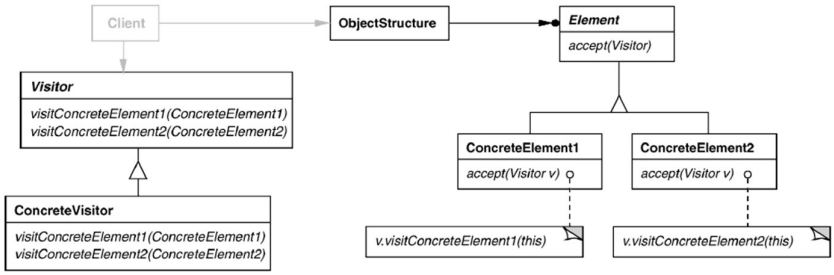



Fig. 42. Visitor Pattern Structure

*5) Consequences*

The advantage with visitor pattern that it is easy to add new operations as a new operation can be added by just adding a new visitor. The classes that are related are localized in a single visitor whereas unrelated set of operations are partitioned into different visitor classes. However, it is difficult to add a new Concrete Element using visitor pattern. Each new concrete element requires a new abstract operation on a visitor and a corresponding implementation for every visitor class. A default implementation can partially resolve this issue.

*6) Implementation*

Visitor pattern can be used to add operations to classes without changing them. It can be achieved by a technique known as 'double dispatch' which means that the execution of depends on the request type and the types of receivers. It effectively lets visitors request different operations on all classes of an element. There are different ways of setting the responsibility of object structure traversal namely, the object structure, in the visitor or in a separate iterator object. If the traversal is complex, then it is a good idea to put the traversal strategy in the visitor.

*7) Example*

The visitor pattern allows centralization of operations on an object structure so that it can vary independently of the client and behave polymorphically. The ShoppingCartVisitor plays the role of an abstract class which has declared the visit operation for all types of visitable classes. The ShoppingCartVisitorImpl is a concrete implementation of the ShoppingCartVisitor. The ItemElement is an interface extended by Fruit and Book classes. It provides an accept operation which enables an object to be visited by the visitor object. The Fruit and Book classes are its concrete implementations. In the example, the shopping cart visitor has visit methods for a book and a fruit which returns the price of a book and a fruit respectively. The fruit and book classes accept a visitor using accept method. The accept method calls the visitor's visit method parametrized by itself. The visit method is chosen polymorphically thus allowing the right method to be called. As mentioned, it then returns the price of the item element.

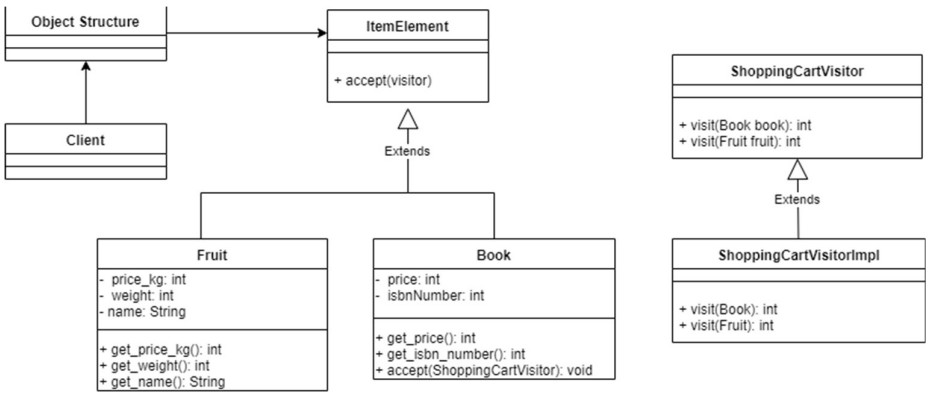



Fig. 43. Visitor Pattern Example

### K. Discussion

The essence of behavioral design pattern lies in encapsulating the varying aspect of a program. A strategy pattern encapsulates an algorithm. A state covers a state-dependent behavior whereas a mediator object is responsible for the protocol between objects. An iterator object is used to encapsulate the traversal of components of an aggregate object. In the behavioral patterns, objects are passed around as arguments. For example, in the visitor pattern the visitor object is passed as an argument to the polymorphic accept operation. In the chain of responsibility pattern, it is passed until it reaches an appropriate handler. Similarly, in command and memento pattern the objects are passed and invoked later. However, command pattern involves polymorphism because command object execution is a polymorphic whereas in the memento the mementos are passed only as values. Mediator and Observer patterns are often used as each other's alternatives. The observer distributes communication in an observer pattern whereas the mediator is responsible for centralization. The observer pattern provides decoupling and better reusability whereas mediators are difficult to reuse. However, it is easier to understand the flow of communication with the mediator pattern whereas in the observer pattern it becomes quite difficult to identify connections in later stages of an application. The indirection introduced by Observer makes a system harder to understand. A few behavioral patterns can be used to decouple senders from receivers in different ways. In the command pattern, a command object binds to communication between a sender and a receiver. In the observer pattern, an interface is defined to indicate changes in a subject to achieve decoupling. The mediator pattern uses a mediator object to refer to each other indirectly. At last, the chain of responsibility achieves decoupling between senders and receivers by passing the request along a chain of receivers. Finally, behavioral design patterns also complement each other well. In that chain of responsibility can use a template pattern while defining primitive operations determining request handling and object forwarding. An iterator can range through an aggregate object while visitor can apply operations to each object. There are examples of behavioral design patterns working well with other patterns as well.

## V. PATTERN CLASSIFICATION ON OOPS CONCEPTS

### A. Inheritance vs Composition

Inheritance lets one class be derived over by another class. Inheritance is easy to implement since it is provided by the language. It lets the subclass override operation of its super class and thus provide the finer functionality. However, with inheritance implementations can't be changed at run-time. Also, parent class' implementation is visible to the subclass which breaks encapsulation. While reusing subclasses implementation dependencies can cause problems. Object composition allows has a relationship of one object with another. Object composition requires a careful design of interfaces talking to each other. However, object composition allows changing implementation at run-time. Object composition over class inheritance helps in keeping classes encapsulated. Also, trouble of managing class hierarchies is dealt with using object composition. It is worth considering though that with object composition more object depend on each other and the system's behaviour is a result of interrelationships between them.

### B. Classification

The following table classifies the patterns based on the Object-Oriented Concepts used in them. The Abstraction is used to indicate the purpose of a class rather than its implementation. Inheritance is essentially the implementation of the existing super-classes. Association means there is a communication between two classes. Aggregation means that a child can exist independently of the parent whereas in composition the life-cycle of child is attached with the parent. In the following classification, association and aggregation are kept in the same column to indicate weak link or loose coupling whereas composition implies strong coupling between the two classes. A yes means the abstraction is necessary, a no means that it cannot be present. 'Not necessary indicates the pattern may or may not use a particular concept and N/A means the concept is not applicable to the pattern.

| Pattern | Purpose | Scope | Abstraction? | Inheritance? | Aggregation / Association? | Composition? | Remarks |
|---|---|---|---|---|---|---|---|
| Abstract Factory | Creational Patterns | Object | Yes | Yes | Yes | No | Aggregation since created objects can exist without factories. |
| Builder | | Object | Yes | Yes | Yes | No | Aggregation since life time of a builder is not associated with one director |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Factory Method | | Class | Yes | Yes | Yes | No | Aggregation since created objects can exist without a factory. |
| Prototype | | Object | Not necessary | Not necessary | No | Yes | |
| Singleton | | Object | No | Not necessary | N/A | N/A | Inheritance is possible but it is rare |
| Adapter | Structural Patterns | Object | Yes | Yes | Yes | No | Adapter class has association with adaptee |
| Bridge | | Object | Yes | Yes | Yes | No | Abstraction has reference to implementation |
| Composite | | Object | Yes | Yes | No | Yes | |
| Decorator | | Object | Yes | Yes | No | Yes | |
| Facade | | Object | Not necessary | Not necessary | No | Yes | The façade has reference to components in the subsystem |
| Flyweight | | Object | Yes | Yes | Yes | No | Aggregation with a flyweight factory |
| Proxy | | Object | Yes | Yes | No | Yes | Composition since work is delegated to a proxy object |
| Chain of Responsibility | Behavioral Patterns | Object | Yes | Yes | Yes | No | |
| Command | | Object | Yes | Yes | Yes | No | Request is encapsulated. Aggregation since command can exist independent of Invoker. |
| Interpreter | | Class | Yes | Yes | Yes | No | Aggregation since terminal expression has weak link with other expressions. |
| Iterator | | Object | Yes | Yes | Yes | No | A concrete aggregate has association with a concrete iterator |
| Mediator | | Object | Yes | Yes | No | Yes | Composition since a concrete colleague is composed using a mediator object |
| Memento | | Object | Not necessary | Not necessary | Yes | No | Aggregation since a caretaker stores different mementos |
| Observer | | Object | Yes | Yes | Yes | No | Loose coupling but subject maintains list of observers. Observers can exist |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | independent of the subject but wouldn't make more sense |
| State | | Object | Yes | Yes | Yes | No | Aggregation since the context has weak coupling with the state |
| Strategy | | Object | Yes | Yes | Yes | No | Aggregation since the context has weak coupling with the Strategy |
| Template Method | | Class | Yes | Yes | N/A | N/A | |
| Visitor | | Object | Yes | Yes | Yes | No | Association since visitor and concrete element refer to each other. |

*Table 1: Pattern classification based on OOPS*

## VI. CONCLUSION

The 23 Gang of Four design patterns are discussed and reasoned about in this report. The first section introduces the design pattern by explaining their purpose and their organization in this report. The second, third and fourth section delved deep into each pattern and expanded on the pattern using seven key points each describing a different aspect of a pattern. Design patterns help in determining the object granularity very well i.e., what level of detail should be composed as objects. The Façade pattern represents complete subsystem as objects whereas flyweight shows how huge number of objects with finest granularities can be supported. Objects created with patterns like abstract factory and builder are responsible for creating other objects. And objects with visitor and command yield objects who are responsible for implementing a request on one or more objects. Some patterns complement each other well while others can be competitors like abstract factory and prototype. Patterns that look similar in implementation can have different intents and thus must be chosen carefully. A composite and a decorator are good examples of that. Often the advantage of a pattern is seen at different stages of an application. For example, adapter has its effect after the application has been designed whereas bridge does it before designing it. [8] Finally, the fifth section is a precis of what composition and inheritance are and why and when composition should be preferred over class inheritance. It offers a classification of design pattern based on the OOPS concepts used. This can serve as ready-reference for designer to pick a pattern based on their application's context. These design patterns have also been a target for criticisms. For example, Peter Norvig demonstrates in his book Design Patterns in Dynamic Programming where he renders 16 out of 23 patterns useless.[9] Besides the patterns lack formal foundations, that they lead to inefficient solutions and doesn't differ significantly from the other abstractions are the common criticisms of these patterns. However, despite the patterns' general nature they serve as the foundation to more complicated patterns. It is recommended though to reference these patterns rather than enforcing them. [8] A map reduce pattern is extended in another independent study which borrows concepts from these patterns.

The implementation of the related code set of slides and the report is available at https://github.com/SKShah36/Design_Patterns. This work is carried out with Dr. Abhishek Dubey's guidance. Thank you, Dr. Dubey.

## REFERENCES


[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and A. Shlomo, *A pattern language : towns, buildings, construction*. .

[2] A. Saha and T. Praharaj, "Design Patterns | Set 1 (Introduction) - GeeksforGeeks." [Online]. Available: https://www.geeksforgeeks.org/design-patterns-set-1-introduction/. [Accessed: 10-Dec-2019].

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Pattern Catalog*. Addison-Wesley, 1994.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, no. 2. 2006.

[5] "Creational patterns." [Online]. Available: https://sourcemaking.com/design_patterns/creational_patterns. [Accessed: 09-Dec-2019].

[6] Saket Kumar, "Prototype Design Pattern - GeeksforGeeks." [Online]. Available: https://www.geeksforgeeks.org/prototype-design-pattern/. [Accessed: 09-Dec-2019].

[7] Saket Kumar, "Composite Design Pattern - GeeksforGeeks." [Online]. Available: https://www.geeksforgeeks.org/composite-design-pattern/. [Accessed: 03-Dec-2019].

[8] "Design Patterns." [Online]. Available: https://sourcemaking.com/design_patterns. [Accessed: 10-Dec-2019].

[9] P. Norvig, "Design Patterns in Dynamic Programming."