Design Patterns in Real Life Object-Oriented Software

Ashish Srivastava and Dr. Sanjay Gupta
Wipro Technologies, Bangalore, India

In the initial stage of project development phase, design patterns have an important role. After capturing all the requirements, it is crucial to decide which design pattern will be most suitable so that we can get the best result out of it in terms of code optimization, performance, maintainability etc. Design pattern helps to reuse code and architecture. It is essential that our design should be specific to the problem at hand but in the same time it should also general enough to address future problems and requirement.

1. Introduction

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. Design pattern describes a problem, which occurs over and over again in our environment, and then describes the solution to that problem. Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development. In general design pattern has four elements: - Pattern Name, Problem, Solution and Consequences.

The fundamental reason for using varies design patterns are to keep classes separated and prevent them from having to know too much about one another. There are a number of strategies that OO programmers use to achieve this separation, among them encapsulation and inheritance. Nearly all languages that have OO capabilities support inheritance. A class that inherits from a parent class has access to all of the methods of that parent class. It also has access to all of its non-private variables. However, by starting your inheritance hierarchy with a complete, working class you may be unduly restricting yourself as well as carrying along specific method implementation baggage.

Instead, Design Patterns suggests that one should always **Program to an interface and not to an implementation.** Purring this more succinctly, we should define the top of any class hierarchy with an abstract class, which implements no methods, but simply defines the methods that class will support. Then, in all of our derived classes we have more freedom to implement these methods as most suits our purposes. **Advantages:** Clients are unaware of the specific class of object they are using one object can be replaced by other object easily. Increase in flexibility**.** Improves opportunities for composition since contained object can be of any class which is implementing a specific interface. **Disadvantages:** Increase the design complexity

Another **approach** might be useful is **Favors object composition over inheritance.** This is simply the construction of objects that contain other objects. Encapsulation of several objects inside another one. While many beginning OO programmers use inheritance to solve every problem, as you begin to write more elaborate programs, the merits of object composition become apparent. Your new object can have the interface that is best for what you want to accomplish without having all the methods of the parent classes. **Advantages: (Over composition):** Internal detail of an object is not visible to other, object are accessed by containing

class through their interfaces, good encapsulation, Each class will have only one task, The composition can be defined dynamically at run-time through objects acquiring reference to other objects of the same type. **Disadvantages: (Over composition):** System will have more objects, Interface must be carefully designed in order to use many objects. **Advantages: (Over inheritance):** new implementation is easy since most of it is inherited, easy to modify or extend the implementation being reused. **Disadvantages: (Over inheritance):** Break encapsulation, since it exposes a subclasses to implementation details of it super class, sub classes may have to change their implementation if super classes changes, Implementations inherited from the super class can not be changed at run time.

2. Design Pattern Catalog

Design pattern vary in their granularities and level of abstraction. Because there is many design patterns, we need a way to organize them. This section classifies design patterns so that we can refer to families of related pattern. This classification will help us to learn the patterns in the catalog faster, and it can direct efforts to find the patterns as well.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	**Factory Method**	**Adapter (Class)**	**Interpreter** **Template Method**
	Object	**Abstract Factory** **Singleton** **Builder** **Prototype**	**Adapter (Object)** **Bridge** **Composite** **Decorator** **Façade** **Fly weight** **Proxy**	**Chain of responsibility** **Command** **Iterator** **Mediator** **Memento** **Observer** **State** **Strategy** **Visitor**

Fig. 1 Design Pattern Space

As shown in **Fig. 1**, we classify pattern catalog by **two** criteria. The **first** criterion, called **purpose,** and reflects what a pattern does. Patterns can have creational, structural or behavioral purpose. **Creational pattern** – concern the process of object creation, **Structure pattern** – deal with the composition of classes or objects and **Behavioral pattern** – characterize the way in which classes or objects interact and distribute responsibility.

The **second** criterion called scope - specifies that the pattern primarily to classes or objects. Class pattern deals with relationships between classes and their subclasses. These relationships are established thru inheritance, so they are static, fixed at compile time. Object pattern deal with object relationships, which can be changed at runtime and are more dynamic. **Creational** (purpose) pattern scope - **Class** – defer some part of

object creation in subclasses, Creational (purpose) pattern scope - **Object** – defer some part of object creation to other object. **Structural** (purpose) pattern scope - **Class** – use inheritance to compose classes. Structural (purpose) pattern scope - **Object** – describes ways to assemble classes. **Behavioral** (purpose) pattern scope - **Class** – use inheritance to describe algorithm and flow of control. Behavioral (purpose) pattern scope - **Object** – describes how a group of objects perform a task that no single object can carry out alone.

3. Pattern in real life object oriented softwares

As we have seen that there are lot of patters in design catalog, but some of them are used very frequently and very effectively. I will try to explain some of them, which I found to be the most useful.

3.1 The Factory Pattern

One type of pattern that we see again and again in Object oriented programs is the **Factory** pattern. A **Factory** pattern is one that returns an instance of one of several possible classes depending on the data provided to it. Usually all of the classes it returns have a common parent class and common methods, but each of them performs a task differently and is optimized for different kinds of data.

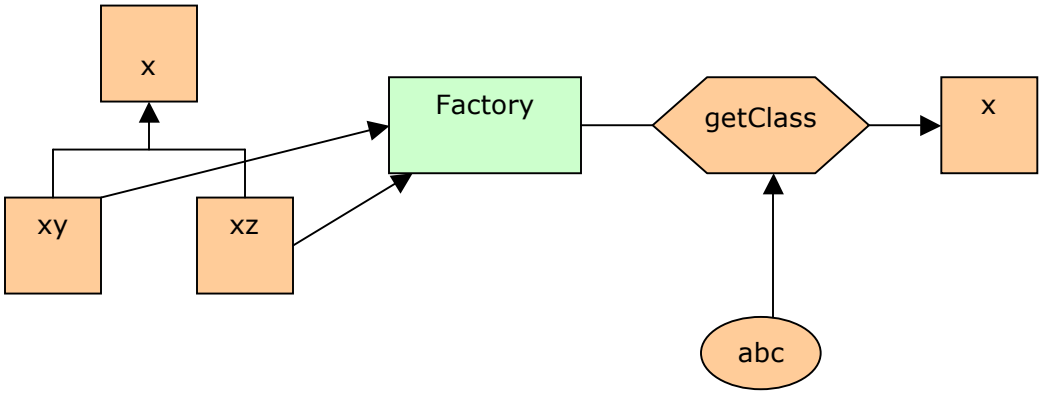

Fig 2: Factory Pattern

In F**ig 2**, x is a base class and classes xy and xz are derived from it. The Factory is a class that decides which of these subclasses to return depending on the arguments we give to it. On the right, we define a getClass method to be one that passes in some value abc, and that returns some instance of the class x. which one it returns doesn't matter to the programmer since they all have the same methods, but different implementations. How it decides which one to return is entirely up to the factory. It could be some very complex function but it is often quite simple. We should consider using a **Factory** pattern when A class can't anticipate which kind of class of objects it must create or a class uses its subclasses to specify which objects it creates or we want to localize the knowledge of which class gets created.

3.2 The Abstract Pattern

The **Abstract Factory** pattern is one level of **abstraction** higher than the factory pattern. We can use this pattern when we want to return one of several related classes of objects, each of which can return several different objects on request. In other words, the **Abstract Factory** is a factory object that returns one of several factories. One classic

application of the abstract factory is the case where our system needs to support multiple "look-and-feel" user interfaces, such as Windows-9x, Motif or Macintosh. We tell the factory that we want our program to look like Windows and it returns a GUI factory, which returns Windows-like objects. Then when we request specific objects such as buttons, check boxes and windows, the GUI factory returns Windows instances of these visual interface components.

In **Fig. 3** (see below), **Abstract Factory** – declares an interface for operation that creates abstract products objects, **Concrete Factories** – implements the operations to create concrete product objects, **Abstract Products** – declares an interface for a type of product object, **Concrete Products** - defines a products objects to be created by the corresponding concrete factory. We should consider using a **Abstract Factory** pattern when system is independent of how its product are created, composed and represented, When a system is configured with one of multiple families of product, We want to provide the class library of products and we want to reveal just their interfaces not their implementations

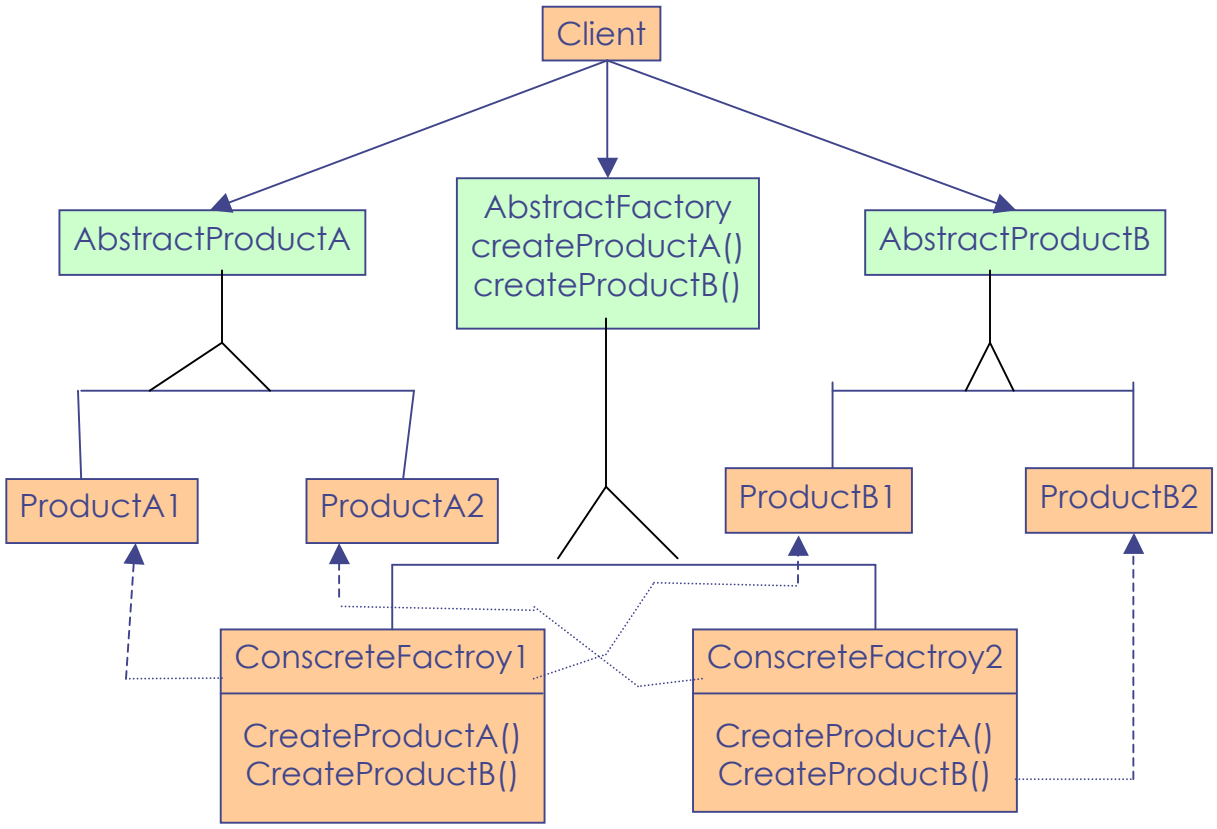

Fig 3: Abstract Pattern

3.3 The singleton Pattern

The **Singleton** pattern is grouped with the other Creational patterns, although it is to some extent a "non-creational" pattern. There are many numbers of cases in programming where we need to make sure that there can be one and only one **instance** of a class. For example, our system can have only one window manager or print spooler,

or a single point of access to a database engine. In other way, the singleton pattern ensures that a class can have only one instance, and provide a global point of access to it.

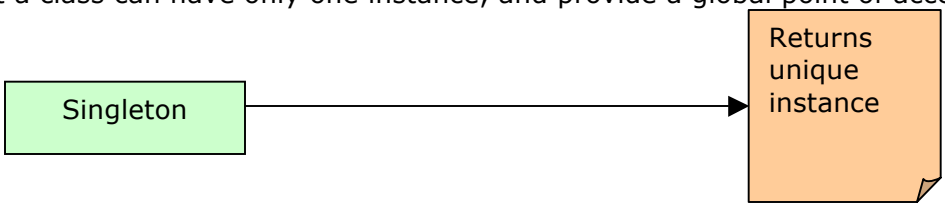

Fig 4: Singleton Pattern

In **Fig. 4**, **Singleton** – defines an instance operation that let clients access its unique instance, it may be responsible to create its own unique instance. We should consider using a **Singleton** pattern when there must be exactly one **instance** of a class, and it must be accessible to client from a well-known point.

3.4 The Adapter Pattern (class and object scope)

The **Adapter** pattern is used to convert the programming interface of one class into that of another. We use adapter pattern whenever we want unrelated classes to work together in a single program. The concept of an **adapter** is thus pretty simple; we write a class that has the desired interface and then make it communicate with the class that has a different interface. There are two ways to do this: by **inheritance** and by **object composition**. In the first case, we derive a new class from the nonconforming one and add the methods we need to make the new derived class match the desired interface. The other way is to include the original class inside the new one and create the methods to translate calls within the new class. In brief, the **Adapter** pattern convert the interface of a class into another interface client expect, and lets the classes work together that couldn't otherwise possible because of incompatible interface.

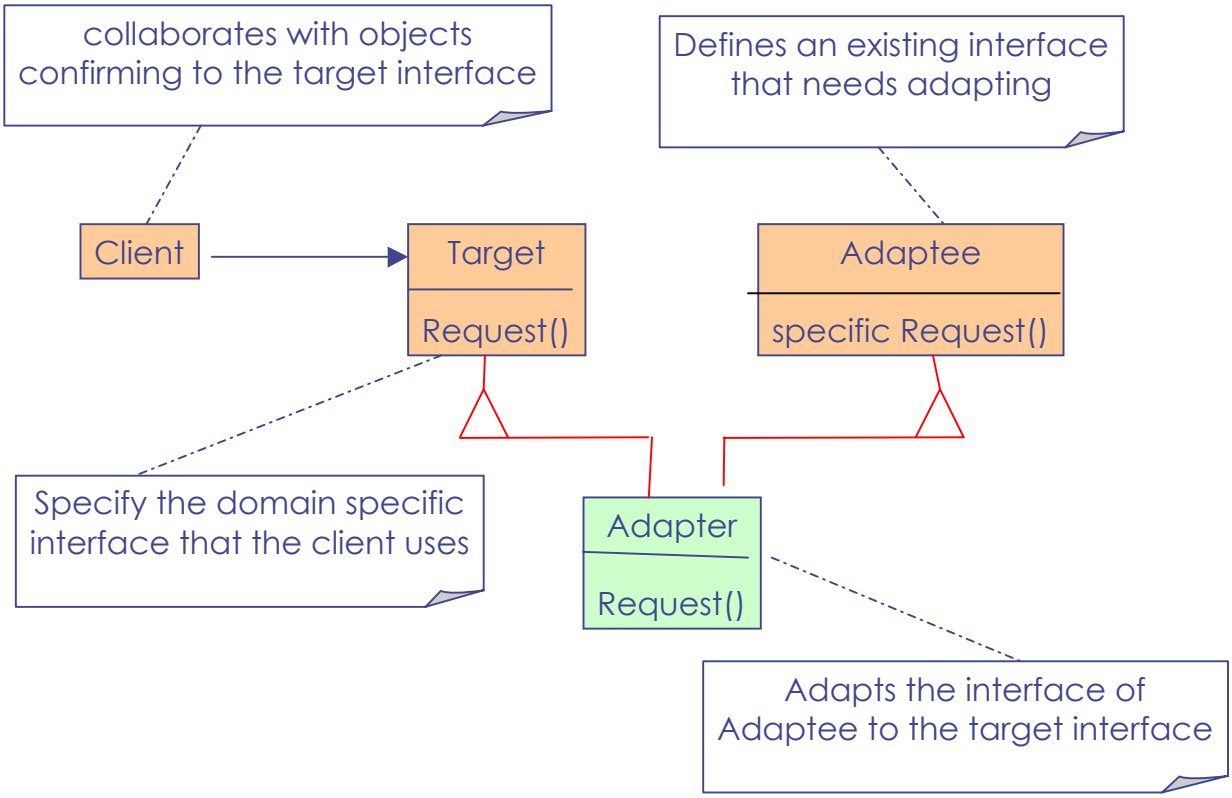

Fig 5: Adapter Pattern

In **Fig. 5**, **Target** – defines the domain-specific interfaces that the client uses, **Client** – collaborates with objects conforming to the target interface, **Adaptee** – defines an existing interface that needs adapting, **Adapter** – adapts the interface of Adaptee to the target interface. We should consider using an **adapter** pattern when we want to use an existing class and its interface doesn't match with the one we need, and when we want to create a reusable class that don't necessarily have compatible interfaces.

3.5 The composite pattern

Frequently we develop systems in which a component may be an individual object or it may represent a collection of objects. The **Composite** pattern is designed to accommodate both cases. We can use the **Composite** to build part-whole hierarchies or to construct data representations of trees. In summary, a **composite** is a collection of objects, any one of which may be either a composite, or just a primitive object. In tree nomenclature, some objects may be nodes with additional branches and some may be leaves.

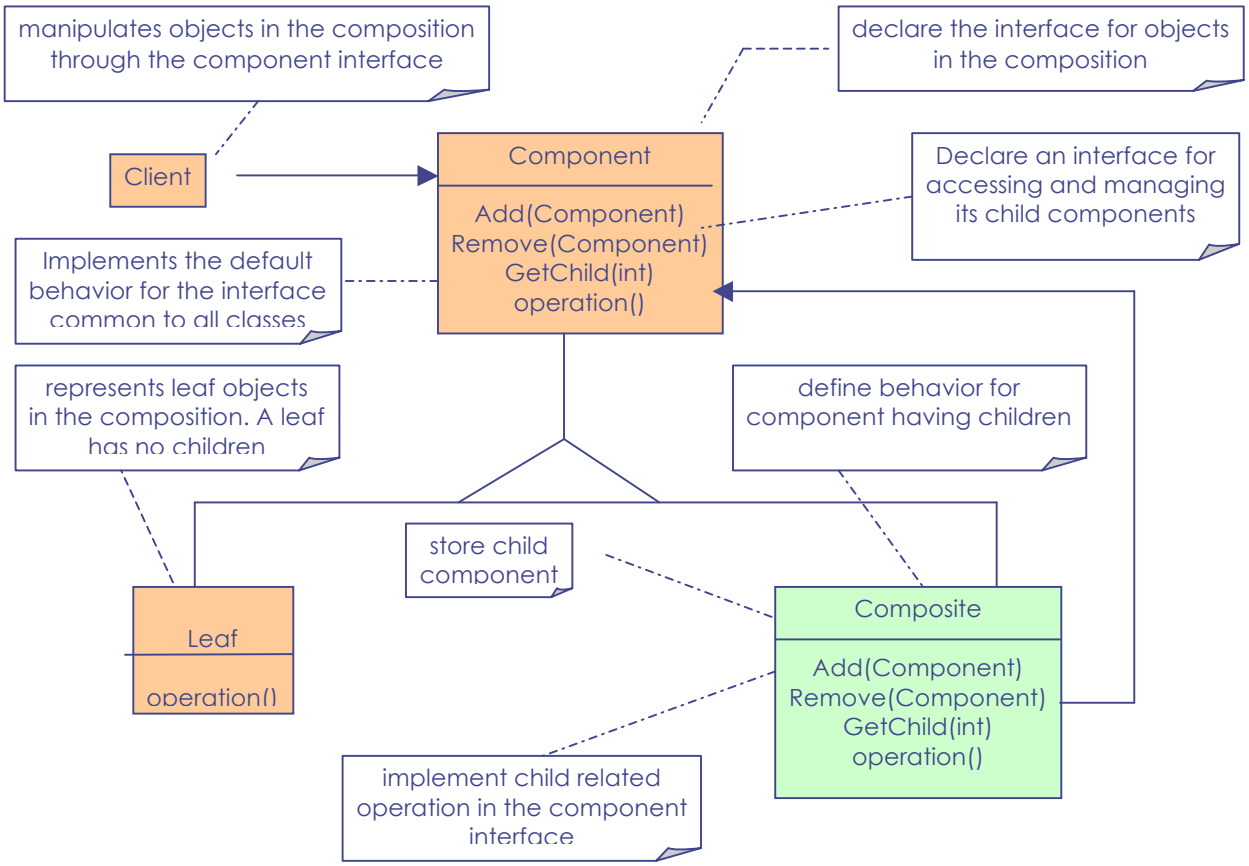

Fig 6: Composite pattern

In the **Fig 6**, **Component** – Declares the interfaces for objects in the composition, Implement default behavior for the interface common to all classes, as appropriate, Declares an interface for accessing and managing its child components (optional), Defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate, **Client** – manipulates objects in the composition through the component interface. **Leaf** – represents leaf objects in the composition. A leaf

has no children, Defines behavior for primitive objects in the composition. **Composite** – Defines behavior for component having children, Store child component, and Implements child-related operations in the Component interface. We should consider using a composite pattern when we want to represent hierarchies of objects, when we want client to be able to ignore the difference between composition of objects and individual objects.

3.6 The Decorator Pattern

The **Decorator** pattern provides us with a way to modify the behavior of individual objects without having to create a new derived class. Sometimes we want to add responsibilities to individual objects, not to an entire class. One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance. This is inflexible, however, because the choice of border is made statically. A client can't control how and when to decorate the component with the border. A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called **Decorator**. It attaches additional responsibilities to an object dynamically, provide a way to modify the behavior of individual objects without having to create a new derived class.

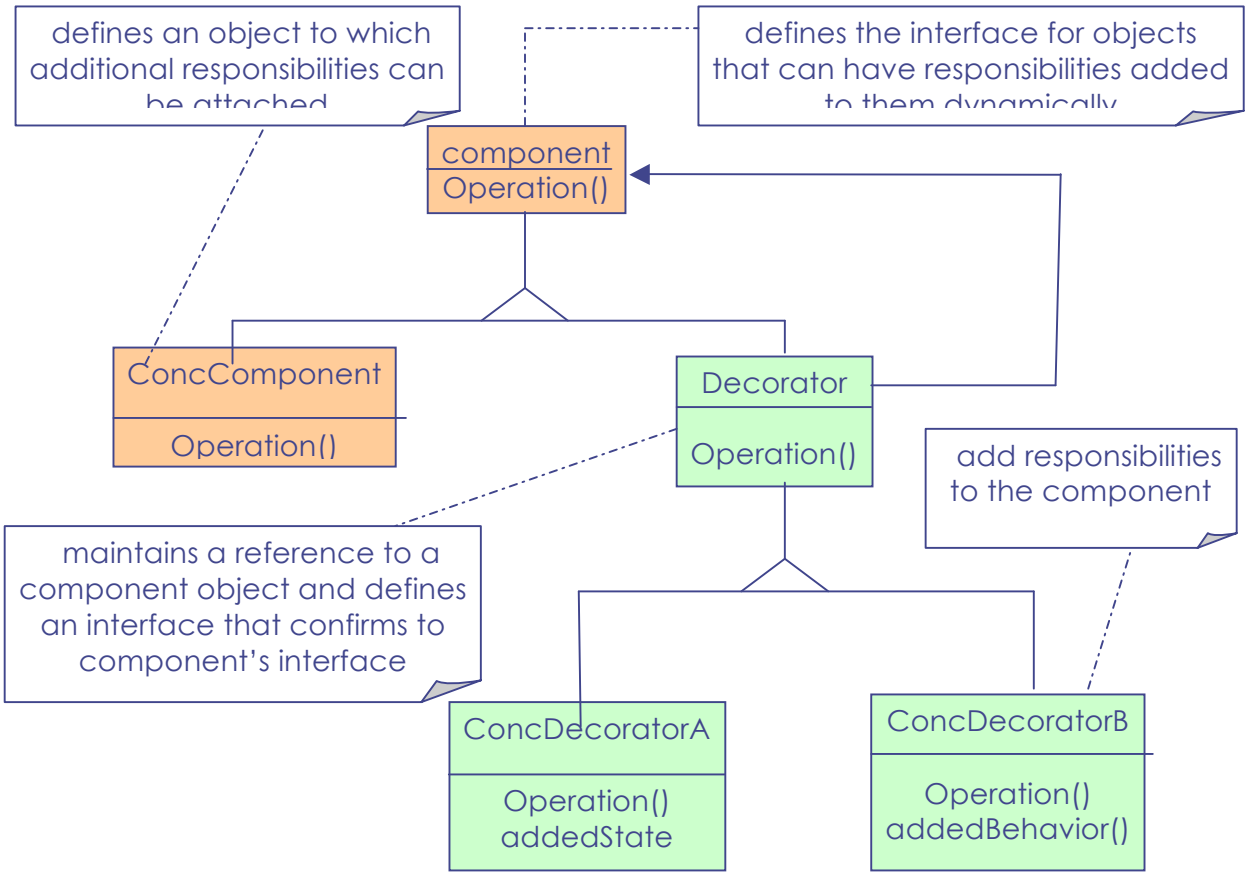

Fig 7: Decorator Pattern

In **Fig 7**, **Component** – defines the interface for objects that can have responsibilities added to then dynamically, **ConcreteComponent** – defines the object to which additional responsibilities can be attached, **Decorator** – maintains a reference to a component object and defines an interface that conforms to Component's interface, **ConcreteDecorator** – adds responsibilities to the component. We should consider using a **decorator** pattern when we want to add responsibilities to individual objects dynamically

and transparently that is without affecting other object, for responsibilities that can be withdrawn.

3.7 The Observer Pattern

In our new, more sophisticated windowing world, we often would like to display data in more than one form at the same time and have all of the displays reflect any changes in that data. For example, we might represent stock price changes both as a graph and as a table or list box. Each time the price changes, we'd expect both representations to change at once without any action on our part. The **Observer** pattern assumes that the object containing the data is separate from the objects that display the data, and that these display objects observes changes in that data. This is simple to illustrate as we see below. We should consider using a **observer** pattern when a change to one object requires changing others, when an object should be able to notify other objects without making assumptions about whom these objects are.

3.8 The Template Pattern

The **Template** pattern provides an abstract definition of an algorithm, whenever we write a parent class where we leave one or more of the methods to be implemented by derived classes, we are in essence using the **Template** pattern. The **Template** pattern formalizes the idea of defining an algorithm in a class, but leaving some of the details to be implemented in subclasses. In other words, if our base class is an abstract class, as often happens in these design patterns, we are using a simple form of the Template pattern. We should consider using a **template** pattern when to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary, We want all the derived classes to implements the algorithm that can vary depending upon the behavior of subclass.

3.9 The Strategy Pattern

Defines a family of algorithm, encapsulate each one, and make them interchangeable. It lets the algorithm vary independently from client and use it, it is also known as policy.

Here, In **Fig 8** (see below), **Strategy** (Compositor) – declares an interface common to all supported algorithms, **Context** use this interface to call the algorithm defined by a ConcreteStrategy. **ConcreteStrategy** – implements the algorithm using the Strategy interface, **Context** – is configured with a ConcreteStrategy object, maintains a reference to a Strategy object, may define an interface that lets the **Strategy** access its data. We should consider using a **strategy** pattern when many related class differ only in behavior. **Strategy** provides a way to configure a class with one of many behaviors and when an algorithm uses data that clients should not know about. Use the strategy pattern to avoid exposing complex, algorithm-specific data structure.

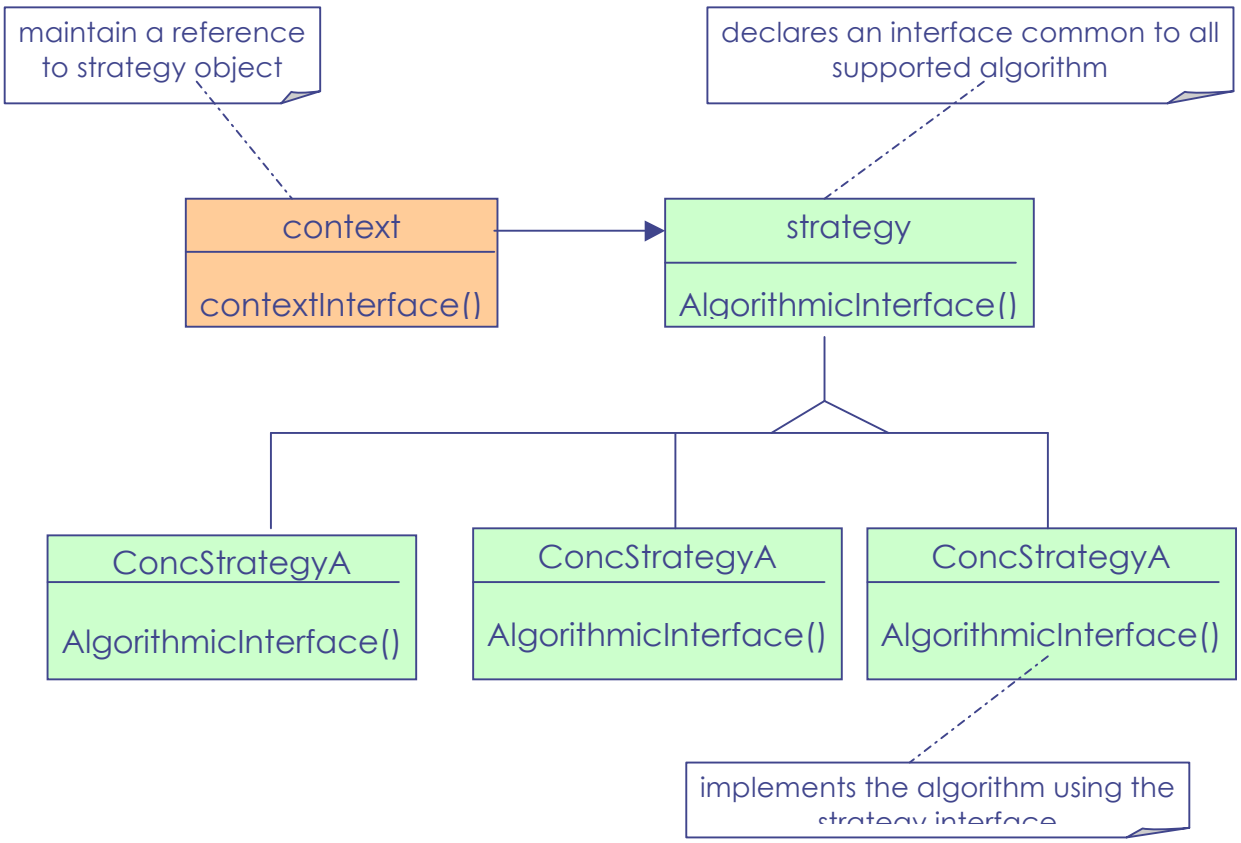

Fig 8: Strategy Pattern

4. Conclusion

There are several ways in which the design pattern can affect the way we design object-oriented software, based on our day-to-day experience with them. These design patterns can make us a better designer. They provide solution to common problems. Design patters are especially useful in turning an analysis model into an implementation model. **The Factory Pattern** is used to choose and return an instance of a class from a number of similar classes based on data you provide to the factory. **The Abstract Factory Pattern** is used to return one of several groups of classes. In some cases it actually returns a Factory for that group of classes. **The Singleton Pattern** is a pattern that insures there is one and only one instance of an object, and that it is possible to obtain global access to that one instance. The **Adapter** pattern, used to change the interface of one class to that of another one. The **Composite** pattern, a collection of objects, any one of which may be either itself a Composite, or just a primitive object. The **Decorator** pattern, a class that surrounds a given class, adds new capabilities to it. The **Template** pattern, defines a general algorithm, although the details may not be worked out completely in the base class. The **observer** pattern, Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. The **Strategy** pattern, allows selecting one of several algorithms dynamically.

References

- Design patterns, Element of Reusable OO s/w by Erirc Gamma, Richards Helm, Raplh johnson, John Vlissides

- Design Pattern, JAVA COMPANION by James W. Cooper