Mediator

Varios objetos necesitan saber de los demás, high coupling, se crea un mediator q sabe q hacer en caso de cada uno de ellos pero ellos no se conocen entre si.

Ejemplo de paginas web, q acción realizan.

Adapter

Objetos se vuelven incompatibles, se crea uno q cree un objeto del nuevo esperado tomando como base el objeto actual

Composite

Todos los objetos se toman de un mismo tipo y mediante los mismo métodos

Ejemplo vicepresitendes y divisiones, si es un objeto de tipo división imprimir imprime cada vp dentro de la división, si es un vp entonces se imprime a si mismo

Observer

Si quieres notificar a todos los observers de un cambio por medio del subject

Chain of responsability

Si quieres notificar a una serie de objetos pero puede ser q no se requiera a todos, sino a uno q pueda manejarlo y ya de ahí termina sin pasar por los demás objetos

Builder

Si quieres dar posibilidad de q un objeto se cree de distintas formas y realice distintas acciones

Factory

Si quieres dar la opción de crear distintos objetos dependiendo del input

Template

Si quieres crear una base de la cual los objetos distintos pueden overidear cierta acción, se basa en herencia

State

Si tienes objetos q pueden manejar distintos estados y dependiendo el estado realizar una acción u otra

Strategy

Utilizando composition si quieres q un objeto realice una acción u otra dependiendo el algoritmo q le sea inyectado. Como el text formatter si le mandan un formatter de bold, entonces eso hace

Command

Si quieres ejecutar varias acciones pero pueden variar

El invoker invoka el comando, el comando tiene el receiver q es el q tiene distintas operaciones, dependiendo q comando sea se utilizan las operaciones del receiver q corresponden

Decorator

Por medio de herencia puedes agregarle valores

Como las computadoras, agregas mouse pero llama el método del padre agregando un poco de información a lo q ya tenia (descripción)

The Design Patterns is the experience in designing the object oriented code.

the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

First, you need to identify the kind of design problem you are facing. A design problem can be categorized into creational, structural, or behavioral. Based to this category you can filter the patterns and selects the appropriate one.

Creational patterns

Creational design patterns are used to design the instantiation process of objects. The creational pattern uses the inheritance to vary the object creation.

First, they all encapsulate knowledge about which concrete classes the system uses. Second, they hide how instances of these classes are created and put together.

Structural patterns

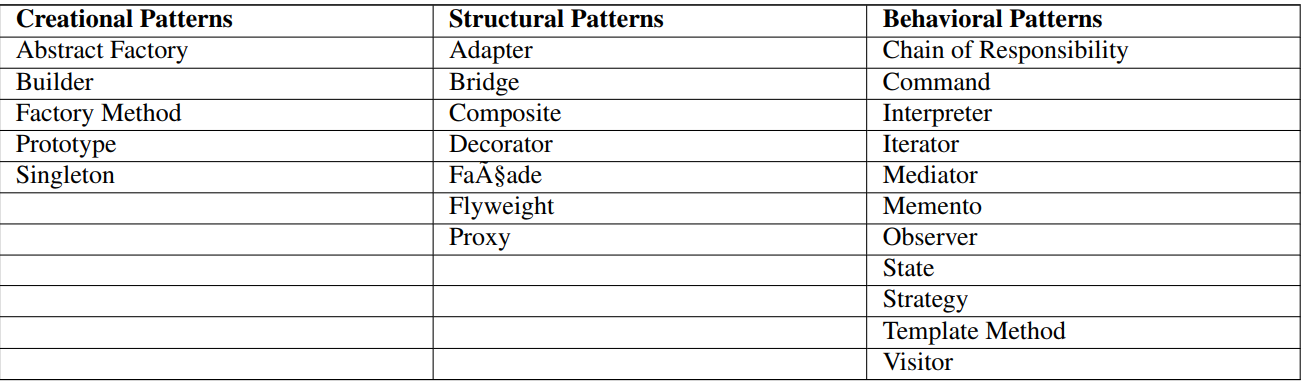
Structural class patterns use inheritance to compose interfaces or implementations.

The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition.

Behavior patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.

Behavioral object patterns use object composition rather than inheritance.



Adapter

An adapter uses composition to store the object it’s supposed to adapt

Como el ejemplo, tomas un objeto con nombre y lo adaptas a uno q tenga nombre y apellido separado

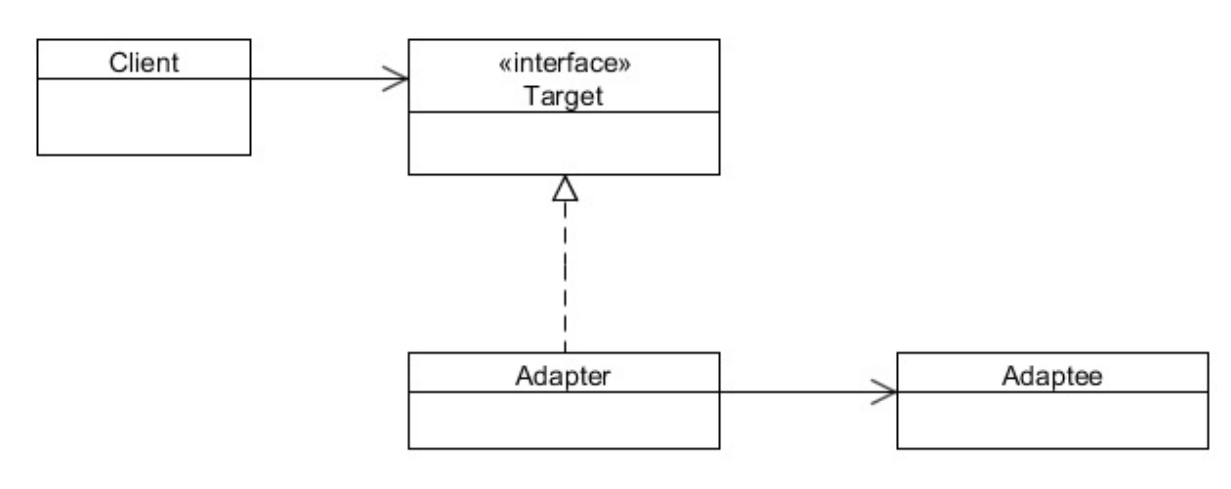
While object adapters use composition to store the object they’re adapting, class adapters are designed to use multiple inheritance to merge the adapted class and the class you’readapting it to. (not for java because the single inheritance)

The problem that arises here is that the site is attached to the Xpay payment gateway which takes an Xpay type of object. The new vendor, PayD, only allows the PayD type of objects to allow the process.

Max needs to find a way to make the code compatible with the vendor’s provided API.

What Max needs here is an Adapter which can sit in between the code and the vendor’s API, and can allow the process to flow.

You can think of an Adapter as a real world adapter which is used to connect two different pieces of equipment that cannot be connected directly.



The Adapter pattern should be used when:

• There is an existing class, and its interface does not match the one you need.

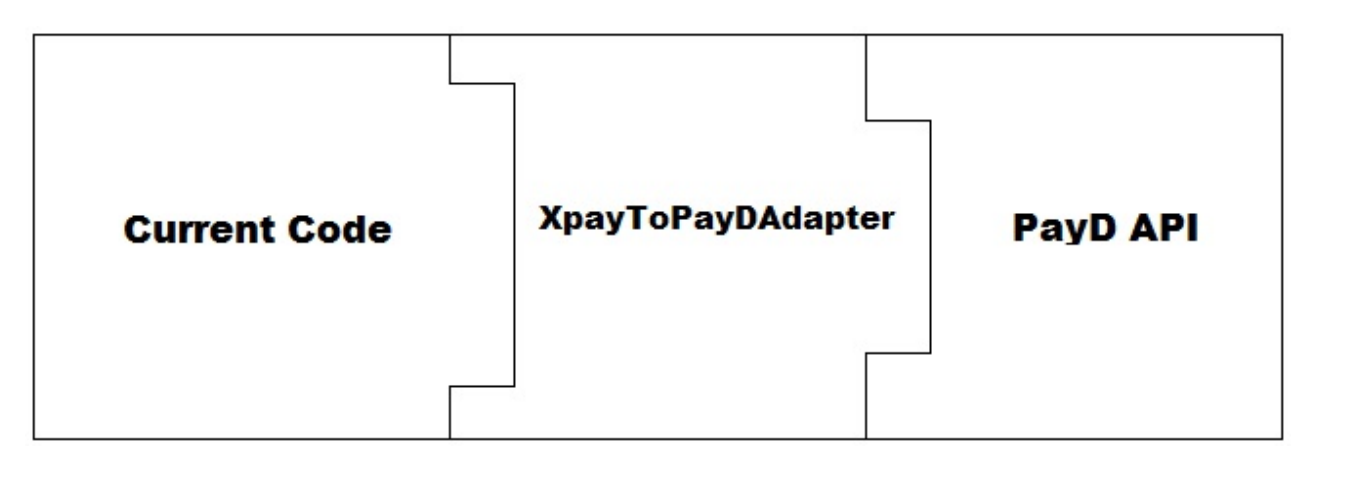
• You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don’t necessarily have compatible interfaces.

• There are several existing subclasses to be use, but it’s impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Se crea una Interfaz q implementa de la q se quiere el objeto final con composition del objeto q originalmente se tiene.

en el contructor se recibe el objeto q se tiene y de este se construye el objeto q se requiere.

En la implementación se envía este objeto requerido, cuando en realidad es un wrapper del objeto original



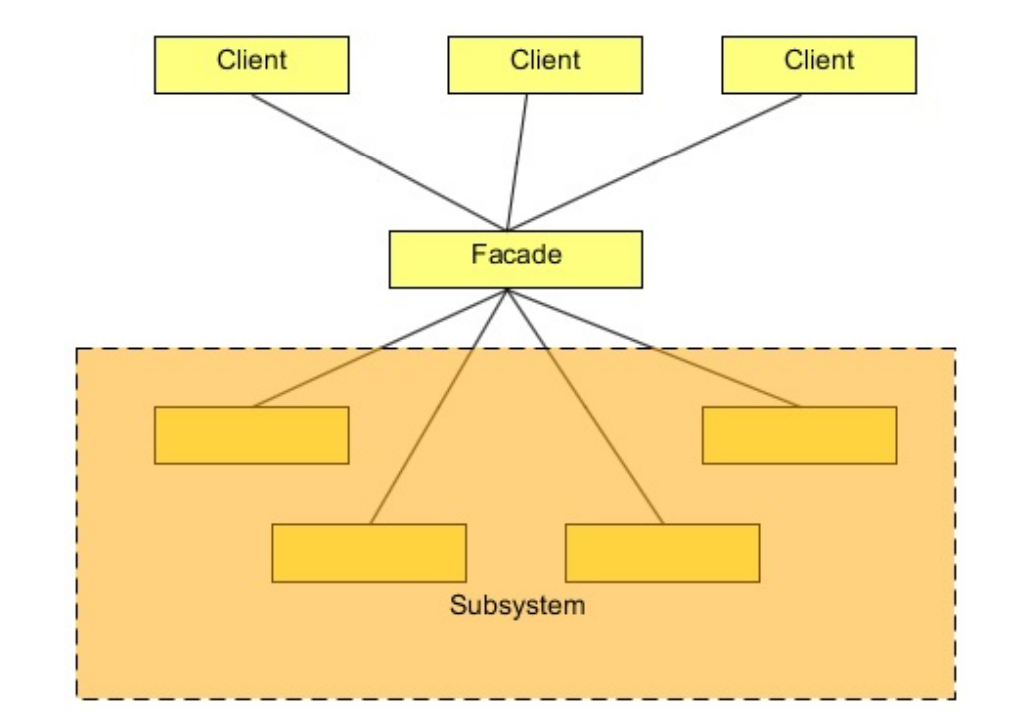
Facade Pattern

Como el de la impresora, llama 15 metodos para imprimir, con el façade creas una interfaz q se encargara de llamarlos a todos y de este solo llamas 1 metodo

A Facade Pattern can help us to resolve a design problem, makes a complex interface easier to use, using a Facade class.

provides a unified interface to a set of interface in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

The Facade do not encapsulate the subsystem classes or interfaces; it just provides a simplified interface to their functionality.

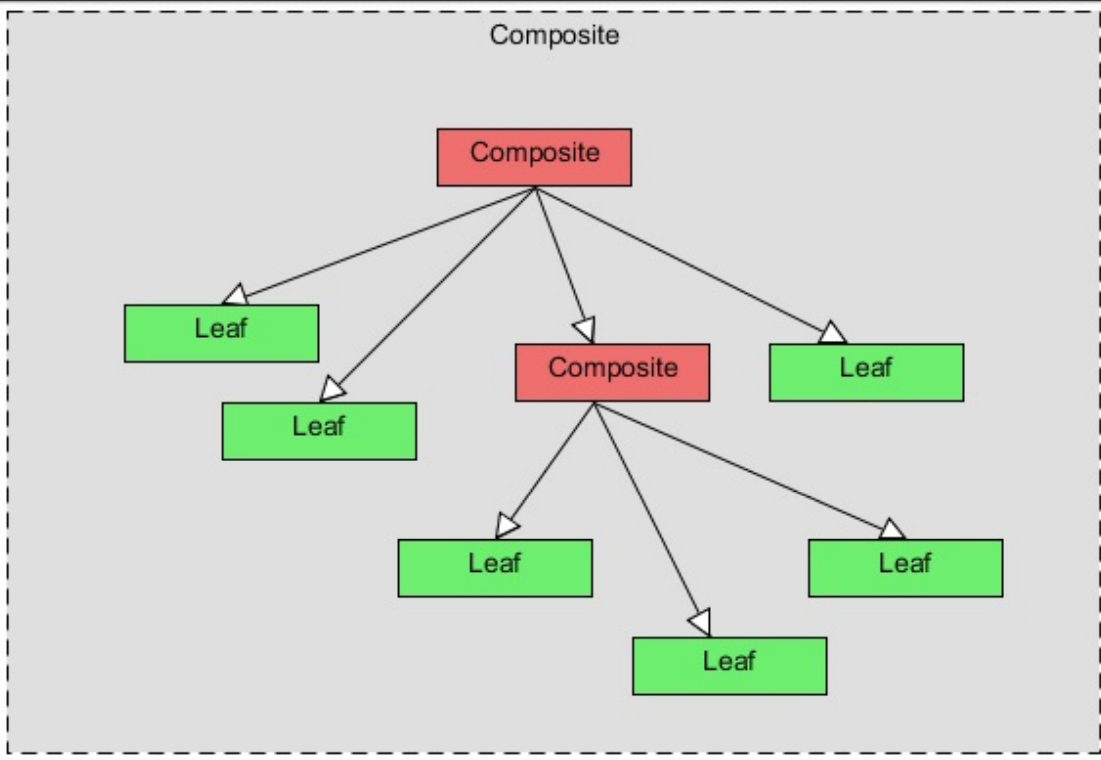


Use the Facade Pattern, when: • You want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve. Most patterns, when applied, result in more and smaller classes. This makes the subsystem more reusable and easier to customize, but it also becomes harder to use for clients that don’t need to customize it.

Clase necesita llamar 10 metodos para iniciar o para terminar y hay q llamarlos uno a uno, se prove una clase q en su lugar solo se llama un metodo y este llama los 10 metodos del subsistema

Composite Design Pattern

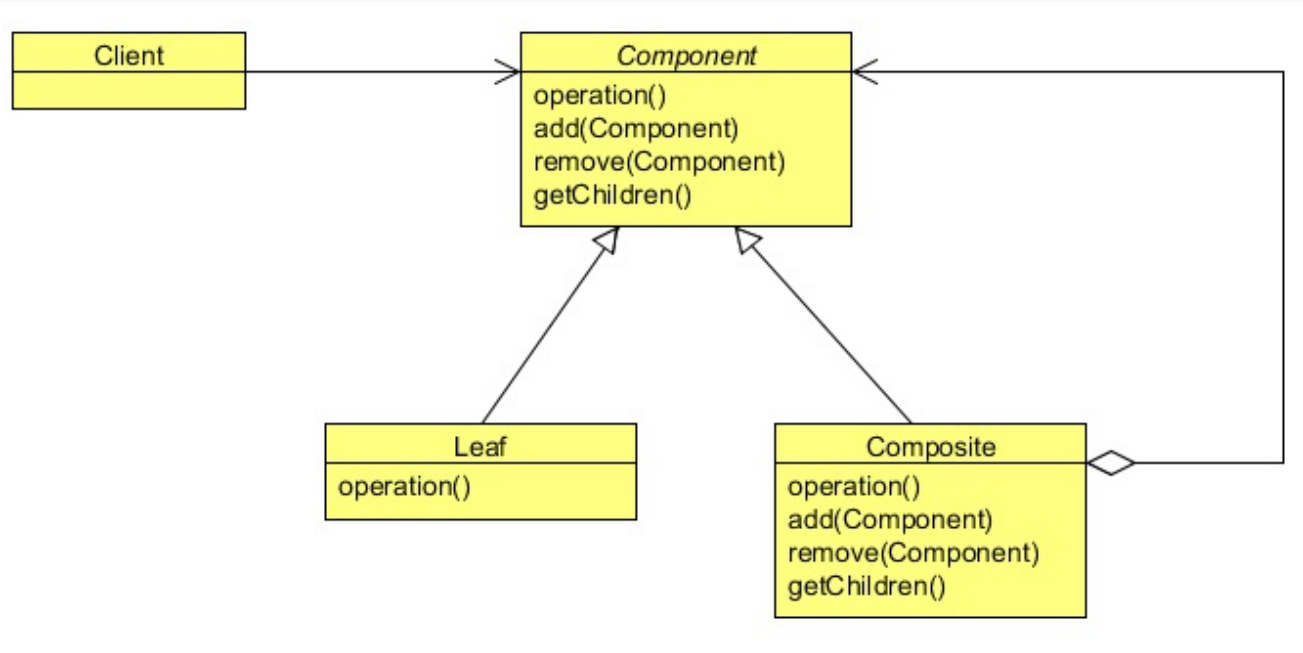
The composite means “putting together” and this is what this design pattern is all about.



Como el ejemplo de los vicepresitentes y las divisions, corporate, division y vp extienden de la clase abstracta q marca el template para utilizer sus mismos metodos, cada uno regresa un iterator q va a regresar lo necesario para poder iterar en cada branch o leaf sin problemas, si es vp se regresa a si mismo, si es division regresa cada uno con el iterate q llamara al metodo deseado

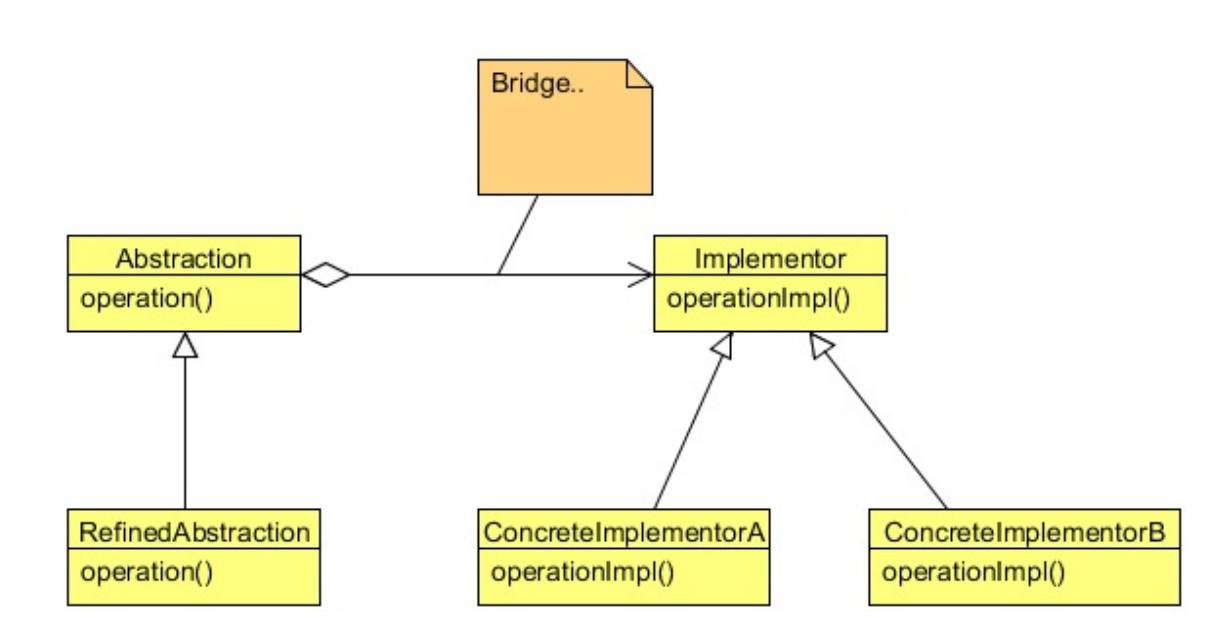
The Composite Pattern in Java can be implemented using the Component class as an abstract class or an interface. In this example, we will use an abstract class which contains all the important methods used in a composite class and a leaf class

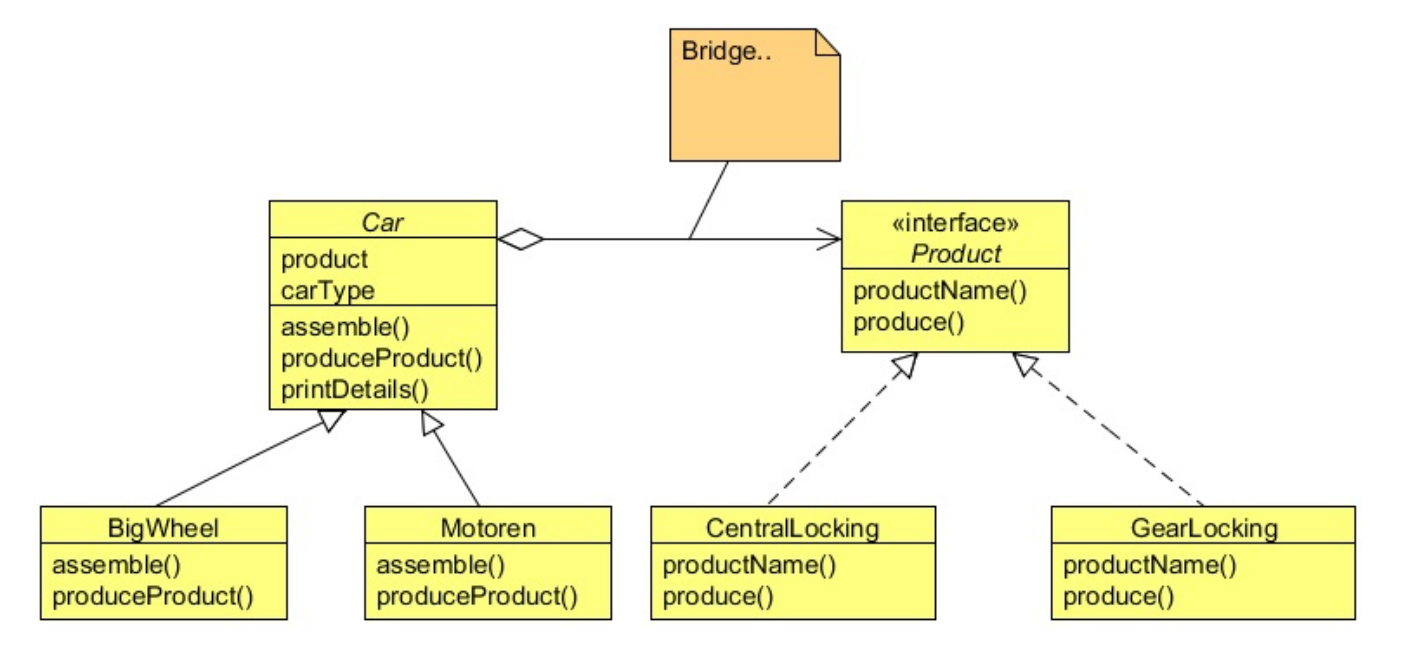
There are methods which are useful only to the composite class and are useless to the leaf class. Just provide the default implementation to these methods, throwing an exception is a good implementation of these methods to avoid any accidental call to these methods by the object which should not support them.



Bridge Pattern

The Bridge Pattern’s intent is to decouple an abstraction from its implementation so that the two can vary independently. It puts the abstraction and implementation into two different class hierarchies so that both can be extend independently.

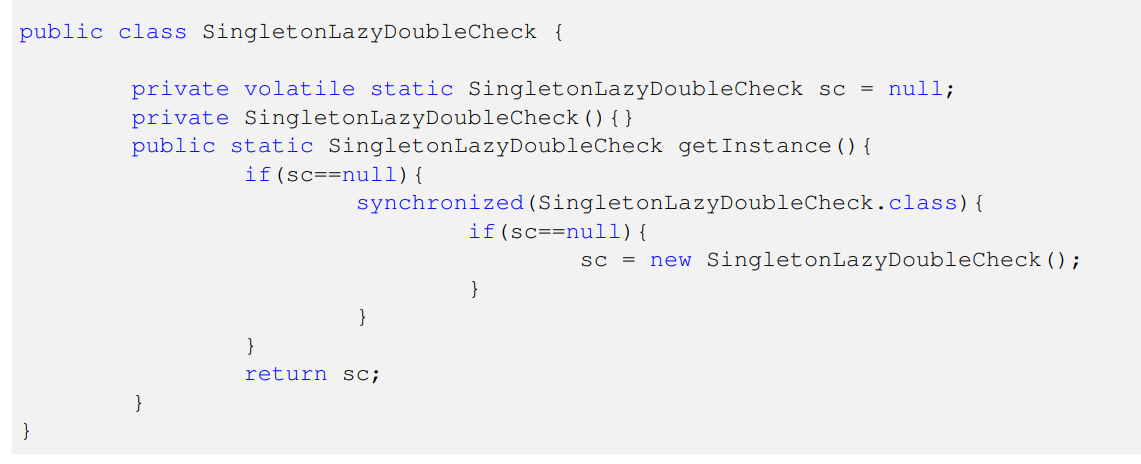




BigWheel extends Car motoren extends car each has a product which can be central locking, gearlocking, etc

Singleton

The Singleton Pattern ensures that a class has only one instance, and provides a global point of access to it.



Observer Design Pattern

The Observer design pattern lets several observer objects be notified when a subject object is changed in some way. Each observer registers with the subject, and when a change occurs, the subject notifies them all. Each of the observers is notified in parallel (that is, at the same time).

Like the edit database example, create a database (subject) which can subscribe, unsubscribe and notify observers of changes, calling each one of their update method.

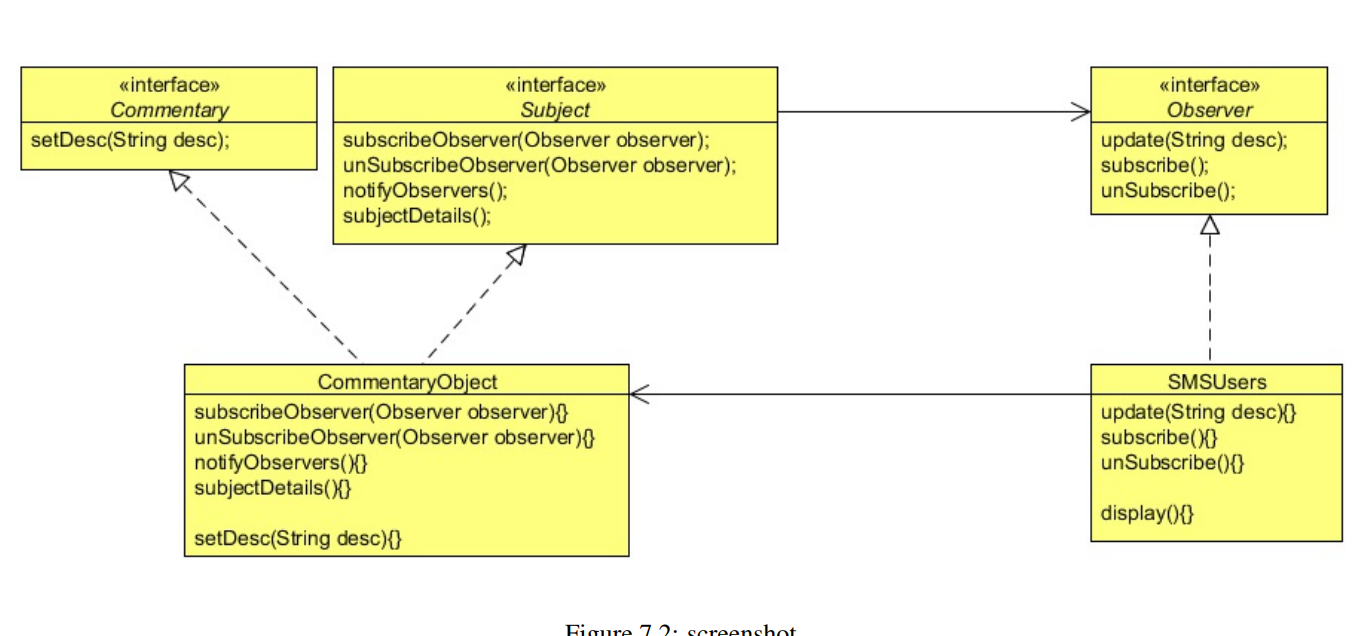
The Observer Pattern is a kind of behavior pattern which is concerned with the assignment of responsibilities between objects. The behavior patterns characterize complex control flows that are difficult to follow at run-time. They shift your focus away from the flow of control to let you concentrate just on the way objects are interconnected.

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

many dependency between objects so that when one object changes state, all its dependents

are notified and updated automatically. The Observer pattern describes these dependencies. The key objects in this pattern are subject and observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in its state.

As you can see, at first two users subscribed themselves for the soccer match and started receiving the commentary. But later one user unsubscribed it, so the user did not receive the commentary again. Then, another user subscribed and starts getting the commentary. All this happens dynamically without changing the existing code and not only this, suppose if, the company wants to broadcast the commentary on emails or any other firm wants to do collaboration with this company to broadcast the commentary



Use the Observer pattern in any of the following situations:

• When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.

• When a change to one object requires changing others, and you don’t know how many objects need to be changed.

• When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don’t want these objects tightly coupled.

Mediator Pattern

Define an object that encapsulates how a set of objects interact

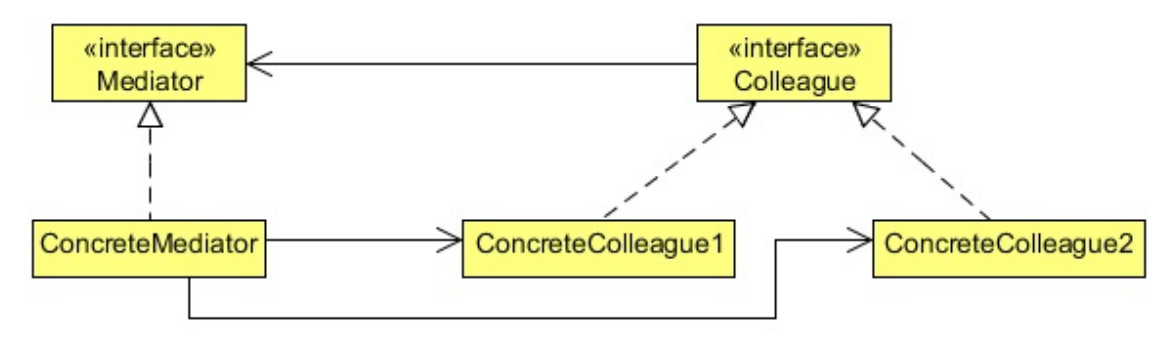
The Mediator Pattern defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Rather than interacting directly with each other, objects ask the Mediator to interact on their behalf which results in reusability and loose coupling. It encapsulates the interaction between the objects and makes them independent from each other. This allows them to vary their interaction with other objects in a totally different way by implementing a different mediator

The Mediator design pattern should be your first choice any time you have a set of objects that are tightly coupled.

Using a Mediator means the interaction code has to reside in only one place, and that makes it easier to

maintain.



When to use the Mediator Pattern

• A set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.

• Reusing an object is difficult because it refers to and communicates with many other objects.

Proxy Design Pattern

The Proxy Pattern is used to create a representative object that controls access to another object, which may be remote, expensive to create or in need of being secured.

One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it. Another reason could be to act as a local representative for an object that lives in a different JVM. The Proxy can be very useful in controlling the access to the original object, especially when objects should have different access rights

In the Proxy Pattern, a client does not directly talk to the original object, it delegates it calls to the proxy object which calls the methods of the original object.

There are three main variations to the Proxy Pattern:

• A remote proxy provides a local representative for an object in a different address space.

• A virtual proxy creates expensive objects on demand.

• A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.

The problem here is that all applications are running at their respective JVMs and the Report Checker application (which we will design soon) should run in the owner’s local system. The object required to generate the report does not exist in the owner’s system JVM and you cannot directly call on the remote object. Remote Proxy is used to solve this problem.

A remote object is an object that lives in the heap of different JVM. You call methods to the local object which forward that calls on to the remote object.

Java supports the communication between the two objects residing at two different locations (or two different JVMs) using RMI. RMI is Remote Method Invocation which is used to build the client and service helper objects, right down to creating a client helper object with the same methods as the remote service. Using RMI you don’t have to write any of the networking or I/O code yourself. With your client, you call remote methods just like normal method calls on objects running in the client’s local JVM

In conclusion, the Remote Proxy acts as a local representative for an object that lives in a different JVM. A method call on the proxy results in the call being transferred over the wire, invoked remotely, and the result being returned back to the proxy and then to the client.

Virtual Proxy

The Virtual Proxy pattern is a memory saving technique that recommends postponing an object creation until it is needed; it is used when creating an object the is expensive in terms of memory usage or processing involved.

Suppose there is a Company object in your application and this object contains a list of employees of the company in a Conta ctList object. There could be thousands of employees in a company. Loading the Company object from the database along with the list of all its employees in the ContactList object could be very time consuming. In some cases you don’t even require the list of the employees, but you are forced to wait until the company and its list of employees loaded into the memory

Protection Proxy

At times, it may be necessary to restrict the accessibility of an object only to a limited set of client objects based on their access rights.

Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:

• A remote proxy provides a local representative for an object in a different address space.

• A virtual proxy creates expensive objects on demand.

• A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.

Chain of responsibility design pattern

The Chain of Responsibility pattern is a behavior pattern in which a group of objects is chained together in a sequence and a responsibility (a request) is provided in order to be handled by the group. If an object in the group can process the particular request, it does so and returns the corresponding response. Otherwise, it forwards the request to the subsequent object in the group.

Use Chain of Responsibility when

• More than one objects may handle a request, and the handler isn’t known a priori. The handler should be ascertained automatically.

• You want to issue a request to one of several objects without specifying the receiver explicitly.

• The set of objects that can handle a request should be specified dynamically.

Flyweight Design Pattern

Too many objects might consume a large piece of memory and can slow down the application or even cause out of memory problems. As a good programmer, one should keep track of instantiated objects and control the object creation in an application. This is especially true, when we have a lot of similar objects and two objects from the pool don’t have much differences between them.

Sometimes the objects in an application might have great similarities and be of a similar kind (a similar kind here means that most of their properties have similar values and only a few of them vary in value).

The Flyweight Pattern is designed to control such kind of object creation and provides you with a basic caching mechanism. It allows you to create one object per type (the type here differs by a property of that object), and if you ask for an object with the same property (already created), it will return you the same object instead of creating a new one

The intent of the Flyweight Pattern is to use shared objects to support large numbers of fine-grained objects efficiently

For objects which can be reuse

The Flyweight pattern’s effectiveness depends heavily on how and where it’s used. Apply the Flyweight pattern when all of the following are true:

• An application uses a large number of objects.

• Storage costs are high because of the sheer quantity of objects.

• Most object state can be made extrinsic.

• Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.

• The application doesn’t depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

Builder Pattern

Como el constructor de robots, agregas metodos q permitan agregar las acciones q se van a realizar y en que orden en una lista de la cual se ejecutara conforme fueron agregados.

Una vez q se agregaron las acciones a realizar se llama el método build para obtener el objeto construido q va a ejecutar las acciones en la lista en el orden correspondiente

Using the Builder pattern, the process of constructing such an object can be designed more effectively. The Builder pattern suggests moving the construction logic out of the object class to a separate class referred to as a builder class. There can be more than one such builder classes, each with different implementations for the series of steps to construct the object. Each builder implementation results in a different representation of the object.

Use the Builder pattern when

• The algorithm for creating a complex object should be independent of the parts that make up the object and how they’re assembled.

• The construction process must allow different representations for the object that’s constructed.

Factory Method Design Pattern

Como la fabrica de computadoras, creas una clase abstracta q regresara el tipo q se desea.

Se implementa de esta y ella va a devolver el tipo de objeto q corresponda a la entrada, si recibe Oracle regresa un objeto de clase OracleConnection.

The Factory Method Pattern, suited for this situation, defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses

The Factory Method Pattern gives us a way to encapsulate the instantiations of concrete types. The Factory Method pattern encapsulates the functionality required to select and instantiate an appropriate class, inside a designated method referred to as a factory method. The Factory Method selects an appropriate class from a class hierarchy based on the application context and other influencing factors. It then instantiates the selected class and returns it as an instance of the parent class type.

Use the Factory Method pattern when

A class can’t anticipate the class of objects it must create.

• A class wants its subclasses to specify the objects it creates.

• Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Abstract Factory Method Design Pattern

The Abstract Factory (A.K.A. Kit) is a design pattern which provides an interface for creating families of related or dependent objects without specifying their concrete classes. The Abstract Factory pattern takes the concept of the Factory Method Pattern to the next level. An abstract factory is a class that provides an interface to produce a family of objects. In Java, it can be implemented using an interface or an abstract class. The Abstract Factory pattern is useful when a client object wants to create an instance of one of a suite of related, dependent classes without having to know which specific concrete class is to be instantiated. Different concrete factories implement the abstract factory interface.

Use the Abstract Factory pattern when

• A system should be independent of how its products are created, composed, and represented.

• A system should be configured with one of multiple families of products.

• A family of related product objects is designed to be used together, and you need to enforce this constraint.

• You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Prototype Design Pattern

sometimes, creating a heavy object could become costly, and if your application needs too many of that kind of objects (containing almost similar properties), it might create some performance issues.

We can use the Prototype Design Pattern to resolve this problem by creating the access control objects on all levels at once, and then provide a copy of the object to the user whenever required. In this case, data fetching from the external resources happens only once. Next time, the access control object is created by copying the existing one. The access control object is not created from scratch every time the request is sent; this approach will certainly reduce object creation time.

The Prototype design pattern is used to specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

The newly copied object may change same properties only if required.

In Java, there are certain ways to copy an object in order to create a new one. One way to achieve this is using the Cloneable interface. Java provides the clone method, which an object inherits from the Object class. You need to implement the Cloneable interface and override this `clone`method according to your needs.

Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; and

• When the classes to instantiate are specified at run-time, for example, by dynamic loading; or

• To avoid building a class hierarchy of factories that parallels the class hierarchy of products; or

• When instances of a class can have one of only 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 the appropriate state.

Memento Design Pattern

The Memento pattern can be used to accomplish this without exposing the object’s internal structure. The object whose state needs to be captured is referred to as the originator.

The Memento Pattern’s intent is, without violating encapsulation, to capture and externalize an object’s internal state so that the object can be restored to this state later.

Memento

Ideally, only the originator that produced the memento would be permitted to access the memento’s internal state.

Originator

Creates a memento containing a snapshot of its current internal state. • Uses the memento to restore its internal state.

When a client wants to save the state of the originator, it requests the current state from the originator. The originator stores all those attributes that are required for restoring its state in a separate object referred to as a Memento and returns it to the client. Thus a Memento can be viewed as an object that contains the internal state of another object, at a given point of time. A Memento object must hide the originator variable values from all objects except the originator. In other words, it should protect its internal state against access by objects other than the originator. Towards this end, a Memento should be designed to provide restricted access to other objects while the originator is allowed to access its internal state

Use the Memento Pattern in the following cases:

• A snapshot of (some portion of) an object’s state must be saved so that it can be restored to that state later, and

• A direct interface to obtaining the state would expose implementation details and break the object’s encapsulation.



Template Design Pattern

the Template Method will “Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure

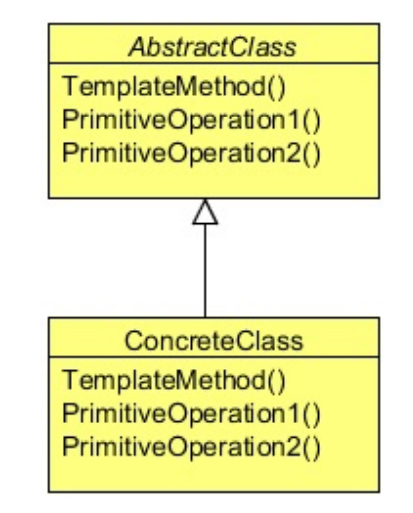
when you have

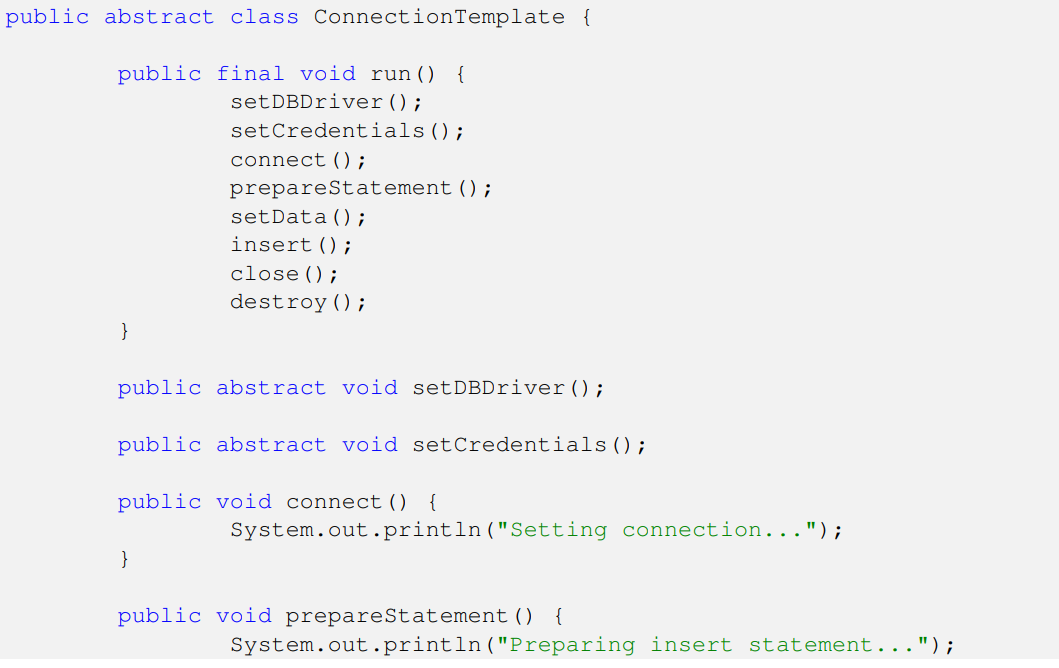
an algorithm that is made of up multiple steps, and you want to be able to customize some of those steps. Note that if you want to rewrite everything from scratch every time — if every step *has* to be customized by writing it from scratch — then you have no need of a template. Only if you have steps that are shared by various implementations of the algorithm do you need to work with a template.

Como el diseño de robots, se agrega una clase abstracta base con los metodos con un default y el metodo go q los llama a todos ellos, en las clases q extienden de esta clase se puede hacer overwrite de los metodos q no sean iguales y el resto se conserva

The Template Design Pattern is a behavior pattern and, as the name suggests, it provides a template or a structure of an algorithm which is used by users. A user provides its own implementation without changing the algorithm’s structure.

The Template Pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses to redefine certain steps of an algorithm without changing the algorithm’s structure.The Template class does not necessarily have to leave the implementation to subclasses in its entirety. Instead, as part of providing the outline of the algorithm, the Template class can also provide some amount of implementation that can be considered as invariant across different implementations. It can even provide default implementation for the variant parts, if appropriate. Only specific details will be implemented inside different subclasses.





The Template Method pattern should be used in the following cases:

• To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.

• When common behavior among subclasses should be factored and localized in a common class to avoid code duplication. You first identify the differences in the existing code and then separate the differences into new operations. Finally, you replace the differing code with a template method that calls one of these new operations.

• To control subclasses extensions. You can define a template method that calls "hook" operations (see Consequences) at specific points, thereby permitting extensions only at those points.

State Design Pattern

You might think that we can implement this without issues using if-else statements, but as a change comes the code would become more complex. The requirement clearly shows that the behavior of an object is truly based on the state of that object. We can use the State Design Pattern which encapsulates the states of the object into another individual class and keeps the context class independent of any state change.

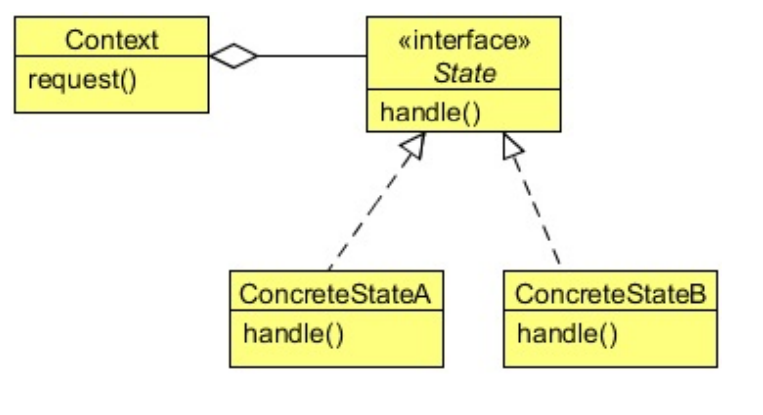
Como el ejemplo del automata q renta cuartos, se crea la interfaz para autómata y estado.

Cada estado existente implementara de este y dependiendo q estado sea se va a crear el algoritmo para cada acción q pasaría dependiendo el estado, ex dispense keys es diferente para got application q para rent aparment. El autómata crea un objeto por cada estado y cada q se va a avanzar a otro estado se cambia el estado actual a el objeto q corresponde a ese estado. El autómata tiene las operaciones q se pueden realizar y dentro de estas acciones llama al método q corresponde del object state. Cada estado recibe el autómata en el constructor, por lo q al ejecutar una operación q requiera avanzar el estado en el autómata lo hacen mediante esta referencia

The State Design Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

The State pattern is useful in designing an efficient structure for a class, a typical instance of which can exist in many different states and exhibit different behavior depending on the state it is in.

The State pattern suggests moving the state-specific behavior out of the Context class into a set of separate classes referred to as State classes. Each of the many different states that a Context object can exist in can be mapped into a separate State class



Use the State pattern in either of the following cases: • An object’s behavior depends on its state, and it must change its behavior at run-time depending on that state. • Operations have large, multipart conditional statements that depend on the object’s state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object’s state as an object in its own right that can vary independently from other objects.

Strategy Design Pattern

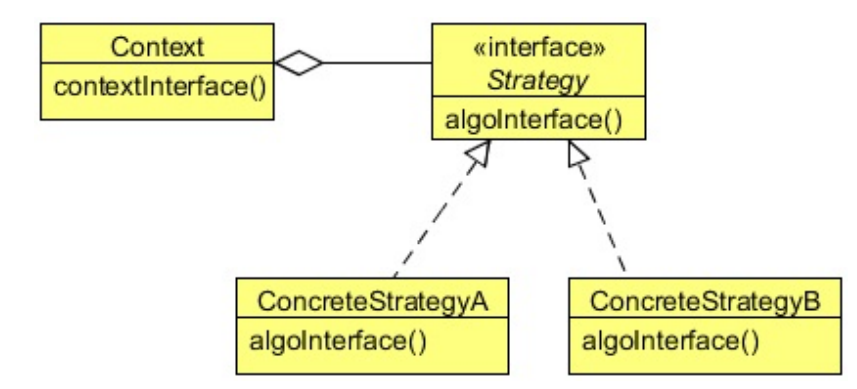
This design principle comes into play when it makes sense to extract code that handles specific tasks from your app.

Como el text tester, separas las maneras de formatear el texto en distintas clases y entonces la seteas dependiendo cual quieras y entonces se utiliza

The Strategy Design Pattern defines a family of algorithms, encapsulating each one, and making them interchangeable.

The Strategy pattern is useful when there is a set of related algorithms and a client object needs to be able to dynamically pick and choose an algorithm from this set that suits its current need. The Strategy pattern suggests keeping the implementation of each of the algorithms in a separate class. Each such algorithm encapsulated in a separate class is referred to as a strategy. An object that uses a Strategy object is often referred to as a context object.

Once the group of related algorithms is encapsulated in a set of Strategy classes in a class hierarchy, a client can choose from among these algorithms by selecting and instantiating an appropriate Strategy class.



Use the Strategy pattern when:

• Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.

• You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.

• An algorithm uses data that clients shouldn’t know about. Use the Strategy pattern to avoid exposing complex, algorithmspecific data structures.

• A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Command Design Pattern

The Command Design Pattern is a behavioral design pattern and helps to decouples the invoker from the receiver of a request.

Like the server commands example.

Generas el receiver depende el tipo, us o asia server y los commandos q se pueden ejecutar diagnose, reboot, quienes reciben el receiver y ejecutan los metodos del receiver q corresponden a el commando utilizado. El invoker solo llama el método del comando q ejecuta los métodos.

Use the Command pattern when you want to: • Parameterize objects by an action to perform. • Specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request. If the receiver of a request can be represented in an address space-independent way, then you can transfer a command object for the request to a different process and fulfill the request there. • Support undo. The Command’s Execute operation can store state for reversing its effects in the command itself. The Command interface must have an added Un-execute operation that reverses the effects of a previous call to Execute. Executed commands are stored in a history list. Unlimited-level undo and redo is achieved by traversing this list backwards and forwards calling Un-execute and Execute, respectively. • Support logging changes so that they can be reapplied in case of a system crash. By augmenting the Command interface with load and store operations, you can keep a persistent log of changes. Recovering from a crash involves reloading logged commands from disk and re-executing them with the Execute operation. • Structure a system around high-level operations built on primitives operations. Such a structure is common in information systems that support transactions. A transaction encapsulates a set of changes to data. The Command pattern offers a way to model transactions. Commands have a common interface, letting you invoke all transactions the same way. The pattern also makes it easy to extend the system with new transactions.

Interpreter Design Pattern

It’s all about putting together your own programming language, or handling an existing one, by creating an interpreter for that language.

Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees. The Interpreter pattern works best when • The grammar is simple. For complex grammars, the class hierarchy for the grammar becomes large and unmanageable. Tools such as parser generators are a better alternative in such cases. They can interpret expressions without building abstract syntax trees, which can save space and possibly time. • Efficiency is not a critical concern. The most efficient interpreters are usually not implemented by interpreting parse trees directly but by first translating them into another form. For example, regular expressions are often transformed into state machines. But even then, the translator can be implemented by the Interpreter pattern, so the pattern is still applicable.

Decorator Design Pattern

As much as possible, make your code closed for modification, but open for extension. In other words, design your core code so that it doesn’t have to be modified a lot, but may be extended as needed.

When you use wrapper code to *extend* your core functionality and you don’t need to modify that core functionality, you are essentially decorating the code.

Like in the computers description, you add a desc which is return, then you can create a sub class which on its desc method will call the method of the parent and will add some more text appart

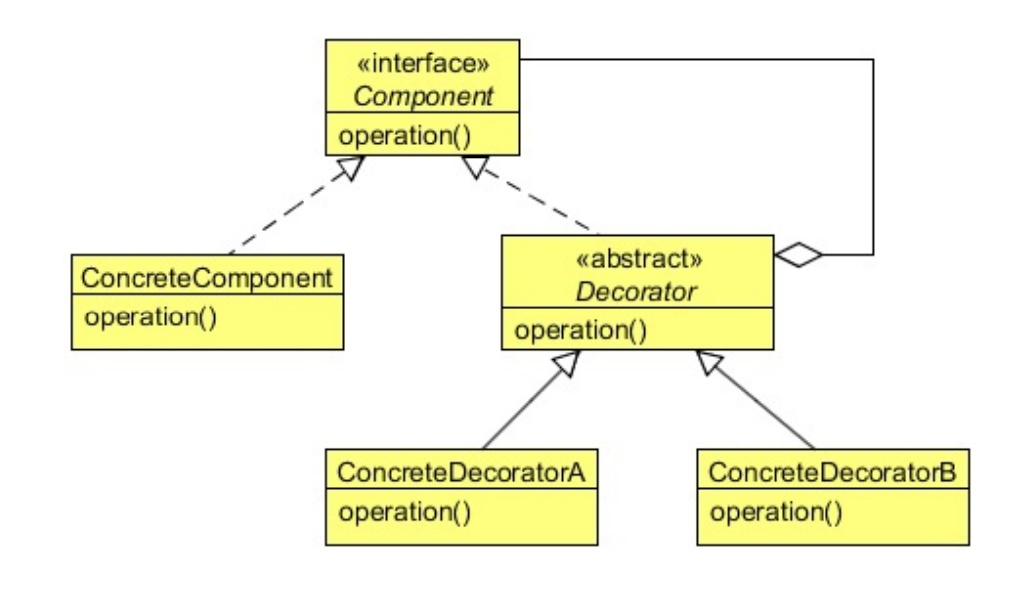
This is something like adding an extra responsibility to our pizza object at runtime and the Decorator Design Pattern is suitable for this type of requirement

The intent of the Decorator Design Pattern is to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.

The Decorator Pattern is used to extend the functionality of an object dynamically without having to change the original class source or using inheritance

This is accomplished by creating an object wrapper referred to as a Decorator around the actual object.

The Decorator object receives all requests (calls) from a client. In turn, it forwards these calls to the underlying object. The Decorator object adds some additional functionality before or after forwarding requests to the underlying object. This ensures that the additional functionality can be added to a given object externally at runtime without modifying its structure



Use the Decorator pattern in the following cases:

• To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.

• For responsibilities that can be withdrawn.

• When extension by sub-classing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for sub-classing.

Iterator Pattern

The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object. The Iterator class defines an interface for accessing the list’s elements. An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already.

The intent of the Iterator Design Pattern is to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

The Iterator pattern allows a client object to access the contents of a container in a sequential manner, without having any knowledge about the internal representation of its contents.

Use the Iterator pattern:

• To access an aggregate object’s contents without exposing its internal representation.

• To support multiple traversals of aggregate objects.

• To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration)

Dynamic binding means that issuing a request doesn't commit you to a particular

implementation until run-time.

You use composition when your object contains other objects instead of inheriting from them.

Evaluar la parte del codigo q cambia mas y aislarlo para q se pueda modificar facilmente, utilizando composition

Como el ejemplo de los vehículos q tienen la instancia de un algoritmo.

Los q extienden de vehiculo setean el tipo de algoritmo q se necesita para ese tipo de vehiculo con lo q en caso de requerir cambios solo se haría en el algoritmo sin modificar lo relacionado al algoritmo

loose coupling — connecting objects through notifications rather than hard coding a connection.