Design Patterns

**What is a design Pattern?**

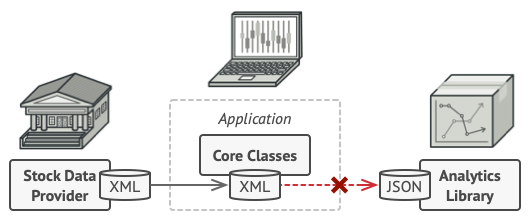
In software Industries, a design pattern is a general repeatable solution to a commonly.

The occurring problem in software design.

1. **Adapter Design Pattern:** The adapter design pattern is a structural design pattern that allows objects with incompatible interfaces to collaborate.

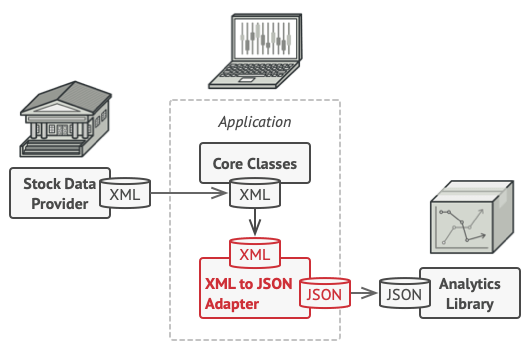
**problem**: Imagine that you’re creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.

At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there’s a catch: the analytics library only works with data in JSON format.



**Solution**: You can create an adapter. This is a special object that converts the interface of one object so that another object can understand it.

An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn’t even aware of the adapter.



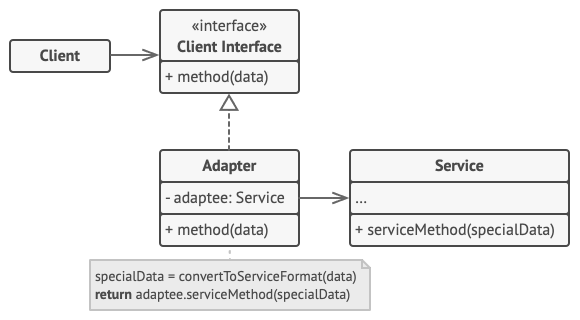
**For example,** you can wrap an object that operates in meters and kilometers with an adapter that converts all the data to imperial units such as feet and miles.

Adapters can not only convert data into various formats but can also help objects with different interfaces

collaborate. Here’s how it works:

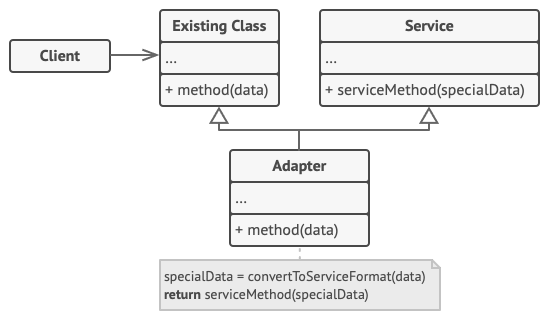
* + The adapter gets an interface, compatible with one of the existing objects.
  + Using this interface, the existing object can safely call the adapter’s methods.
  + Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

**Object adapter:** This implementation uses the object composition principle: the adapter implements the interface of one object and wraps the other. It can be implemented in all popular programming languages.



**Class Adapter:**

This implementation uses inheritance: the adapter inherits interfaces from both objects at the same time. Note that this approach can only be implemented in programming languages that support multiple inheritances, such as C++.



## **Pros and Cons**

* *Single Responsibility Principle*. You can separate the interface or data conversion code from the primary business logic of the program.
* *Open/Closed Principle*. You can introduce new types of adapters into the program without breaking the existing client code if they work with the adapters through the client interface.
* The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it’s simpler just to change the service class so that it matches the rest of your code.

**Example:**

#include <iostream>

using namespace std;

**class** MobileInterface // Interface of Adaptee

{

public:

virtual **void** Square\_pin() = 0;

};

**class** MobilePhone : public MobileInterface // Concrete Adaptee

{

public:

**void** Square\_pin() {

cout << "The phone only supports a Square pin charger" << endl;

}

};

**class** AdapterInterface // Target, the interface of Adapter which client will use

{

public:

virtual **void** Two\_pin() = 0;

};

**class** Adapter : public AdapterInterface

{

private:

MobileInterface \*mobile\_interface;

public:

Adapter(MobileInterface \*obj) {

mobile\_interface = obj;

}

**void** Two\_pin() {

mobile\_interface->Square\_pin();

}

};

int main()

{

MobilePhone \*mobilephone\_ptr = **new** MobilePhone();

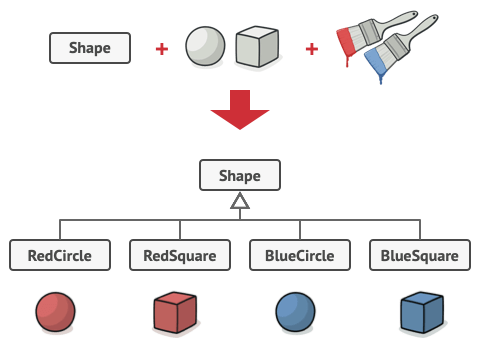
AdapterInterface \*adapter = **new** Adapter(mobilephone\_ptr);

adapter->Two\_pin();

}

1. **Bridge Design Pattern:** The bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies-abstraction-implementation which can be developed independently with each other.

## **Problem**: you have a geometric Shape class with a pair of subclasses: Circle and Square. You want to extend this class hierarchy to incorporate colors, so you plan to create Red and Blue shape subclasses. However, since you already have two subclasses, you’ll need to create four class combinations such as BlueCircle and RedSquare.

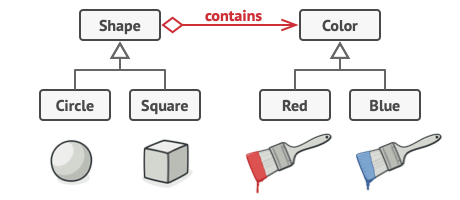


Adding new shape types and colors to the hierarchy will grow it exponentially. For example, to add a triangle shape you’d need to introduce two subclasses, one for each color. And after that, adding a new color would require creating three subclasses, one for each shape type. The further we go, the worse it becomes.

**Solution:**

This problem occurs because we’re trying to extend the shape classes in two independent dimensions: form and by color. That’s a very common issue with class inheritance.

The Bridge pattern attempts to solve this problem by switching from inheritance to object composition. What this means is that you extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all its state and behaviors within one class.



## **Structure:**

## Bridge design pattern

## **Pros and Cons**

* You can create platform-independent classes and apps.
* The client code works with high-level abstractions. It isn’t exposed to the platform details.
* *Open/Closed Principle*. You can introduce new abstractions and implementations independently from each other.
* *Single Responsibility Principle*. You can focus on high-level logic in the abstraction and on platform details in the implementation.

**Example**:

#include <iostream>

using namespace std;

class color

{

public:

virtual void fillcolor() = 0;

};

class Red : public color

{

public:

void fillcolor()

{

cout<<"Fill Red color : "<<endl;

}

};

class Blue : public color

{

public:

void fillcolor()

{

cout<<"Fill Blue color : "<<endl;

}

};

class shape

{

public:

virtual void draw() = 0;

};

class Rectangle:public shape

{

public:

color \*mcolor;

Rectangle(color \*mcolor):mcolor(mcolor)

{

}

void draw()

{

cout<<"Rectangle draw : "<<endl;

mcolor->fillcolor();

}

};

class circle : public shape

{

public:

void draw()

{

cout<<"circle draw : "<<endl;

}

};

int main()

{

color \*mcolor = new Blue();

shape \*mshape = new Rectangle(mcolor);

mshape->draw();

return 0;

}

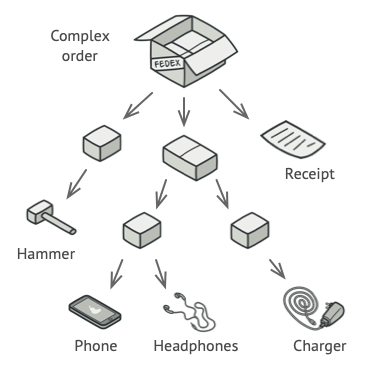
1. **Composite Design Pattern:**  The composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

## **Problem:**

Using the Composite pattern makes sense only when the core model of your app can be represented as a tree.

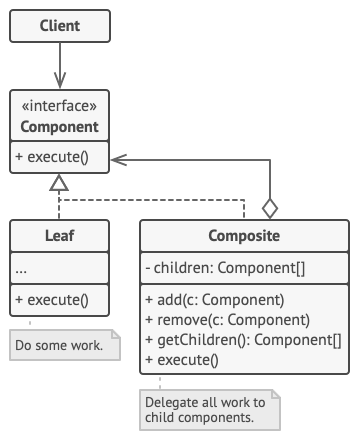
For example, imagine that you have two types of objects: Products and Boxes. A Box can contain several Products as well as several smaller boxes. These little Boxes can also hold some Products or even smaller Boxes, and so on.

Say you decide to create an ordering system that uses these classes. Orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes. How would you determine the total price of such an order?



## 

## **Structure:**



1. The **Component** interface describes operations that are common to both simple and complex elements of the tree.
2. The **Leaf** is a basic element of a tree that doesn’t have sub-elements.

Usually, leaf components end up doing most of the real work, since they don’t have anyone to delegate the work to.

1. The **Container** (aka *composite*) is an element that has sub-elements: leaves or other containers. A container doesn’t know the concrete classes of its children. It works with all sub-elements only via the component interface.

Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and then returns the

## **Pros and Cons**

* You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.
* Open/Closed Principle. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.
* It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you’d need to overgeneralize the component interface, making it harder to comprehend.

Example:

Reference link for this example:

https://aneescraftsmanship.com/composite-design-pattern/

**#include <iostream>**

**#include <list>**

**using** **namespace** std;

**class** School\_bag

{

**public**:

**virtual** **void** draw() = 0;

};

**class** Water\_bottle : **public** School\_bag

{

**public**:

**void** draw(){

cout << "Water bottle" << endl;

}

};

**class** Colour\_pencils : **public** School\_bag

{

**public**:

**void** draw(){

cout << "Colour\_pencils" << endl;

}

};

**class** Lunch\_box : **public** School\_bag

{

**public**:

**void** draw(){

cout << "Lunch\_box" << endl;

}

};

**class** Drawing\_book : **public** School\_bag

{

**public**:

**void** draw(){

cout << "Drawing\_book" << endl;

}

};

**class** Book: **public** School\_bag

{

**public**:

**void** draw(){

cout << "Books" << endl;

}

};

**class** CompositeSchool\_bag : **public** School\_bag

{

**private**:

list<School\_bag \*> child\_Schoolbag;

**public**:

**void** draw()

{

**for**(list<School\_bag \*>::iterator Schoolbag = child\_Schoolbag.begin(); Schoolbag != child\_Schoolbag.end(); ++Schoolbag){

(\*Schoolbag)->draw();

}

}

**void** add(School\_bag \*Schoolbag)

{

child\_Schoolbag.push\_back(Schoolbag);

}

**void** remove(School\_bag \*Schoolbag)

{

child\_Schoolbag.remove(Schoolbag);

}

};

**int** main()

{

Water\_bottle \*Water\_bottle1 = new Water\_bottle();

Colour\_pencils\*Colour\_pencils1 = new Colour\_pencils();

Lunch\_box\*Lunch\_box1=new Lunch\_box();

Book \*Book1= new Book();

Drawing\_book \*Drawing\_book1 = new Drawing\_book();

CompositeSchool\_bag \*Schoolbag = new CompositeSchool\_bag();

CompositeSchool\_bag \*Section1 = new CompositeSchool\_bag();

CompositeSchool\_bag \*Section2= new CompositeSchool\_bag();

CompositeSchool\_bag \*Section3= new CompositeSchool\_bag();

Schoolbag->add(Section1);

Schoolbag->add(Section2);

Section1->add(Water\_bottle1);

Section1->add(Section3);

Section2->add(Colour\_pencils1);

Section2->add(Drawing\_book1);

Section3->add(Lunch\_box1);

Section3->add(Book1);

Schoolbag->draw();

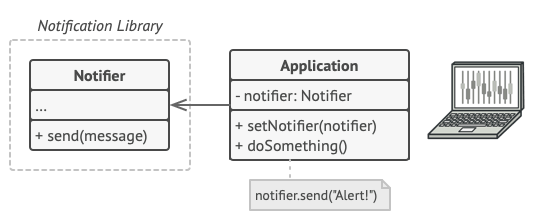
**return** 0;

}

1. **Decorator design pattern:** The Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behavior.

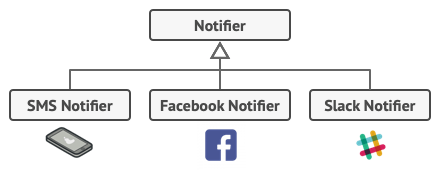
**Problem:** Imagine that you’re working on a notification library that lets other programs notify their users about important events.

The initial version of the library was based on the Notifier class that had only a few fields, a constructor, and a single send method. The method could accept a message argument from a client and send the message to a list of emails that were passed to the notifier via its constructor. A third-party app that acted as a client was supposed to create and configure the notifier object once, and then use it each time something important happened.



A program could use the notifier class to send notifications about important events to a predefined set of emails.

At some point, you realize that users of the library expect more than just email notifications. Many of them would like to receive an SMS about critical issues. Others would like to be notified on Facebook and, of course, corporate users would love to get Slack notifications.

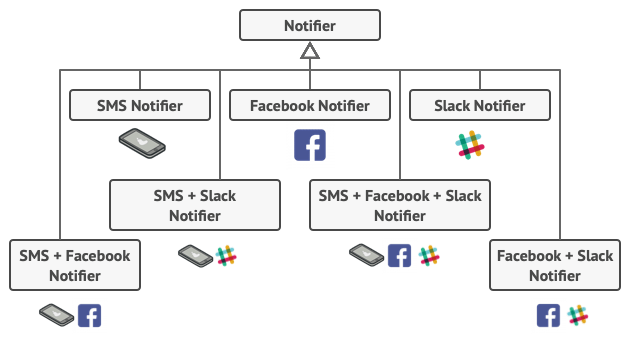


Each notification type is implemented as a notifier’s subclass.

How hard can that be? You extended the Notifier class and put the additional notification methods into new subclasses. Now the client was supposed to instantiate the desired notification class and use it for all further notifications.

But then someone reasonably asked you, “Why can’t you use several notification types at once? If your house is on fire, you’d probably want to be informed through every channel.”

You tried to address that problem by creating special subclasses which combined several notification methods within one class. However, it quickly became apparent that this approach would bloat the code immensely, not only the library code but the client code as well.



The combinatorial explosion of subclasses.

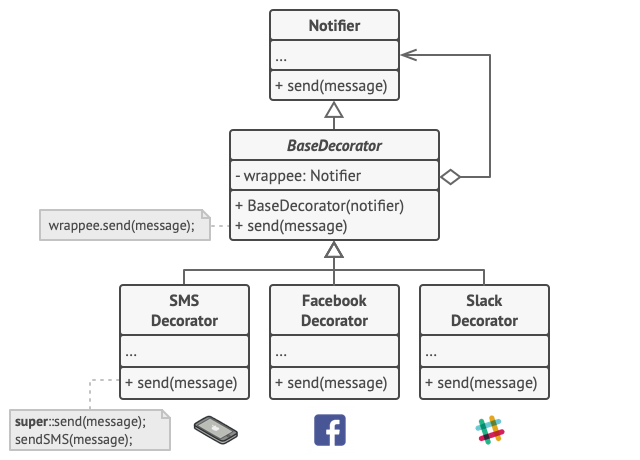
## **Solution:**

Extending a class is the first thing that comes to mind when you need to alter an object’s behavior. However, inheritance has several serious caveats that you need to be aware of.

* Inheritance is static. You can’t alter the behavior of an existing object at runtime. You can only replace the whole object with another one that’s created from a different subclass.
* Subclasses can have just one parent class. In most languages, inheritance doesn’t let a class inherit behaviors of multiple classes at the same time.

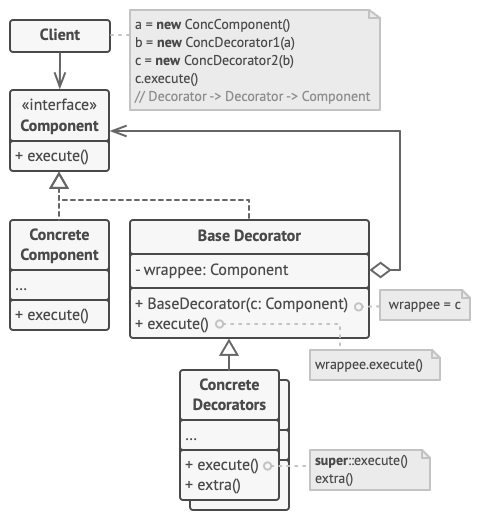
One of the ways to overcome these caveats is by using Aggregation or Composition instead of Inheritance. Both alternatives work almost the same way: one object has a reference to another and delegates it some work, whereas, with inheritance, the object itself can do that work, inheriting the behavior from its superclass.

With this new approach, you can easily substitute the linked “helper” object with another, changing the behavior of the container at runtime. An object can use the behavior of various classes, having references to multiple objects and delegating all kinds of work. Aggregation/composition is the key principle behind many design patterns, including Decorator. On that note, let’s return to the pattern discussion.



Various notification methods become decorators.

## **Structure:**



1. The **Component** declares the common interface for both wrappers and wrapped objects.
2. **Concrete Component** is a class of objects being wrapped. It defines the basic behavior, which can be altered by decorators.
3. The **Base Decorator** class has a field for referencing a wrapped object. The field’s type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.
4. **Concrete Decorators** define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behavior either before or after calling the parent method.
5. The **Client** can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.

## 

## **Pros and Cons:**

* You can extend an object’s behavior without making a new subclass.
* You can add or remove responsibilities from an object at runtime.
* You can combine several behaviors by wrapping an object into multiple decorators.
* *Single Responsibility Principle*. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.
* It’s hard to remove a specific wrapper from the wrapper stack.
* It’s hard to implement a decorator in such a way that its behavior doesn’t depend on the order in the decorators stack.
* The initial configuration code of layers might look ugly.

**Example:**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**// Component**

**class Car**

**{**

**public:**

**virtual string Fit\_parts () = 0;**

**virtual float Cost() = 0;**

**virtual ~Car(){}**

**};**

**// Concrete Component**

**class Audi : public Car**

**{**

**public:**

**string Fit\_parts ()**

**{**

**return "Audi Car";**

**}**

**float Cost()**

**{**

**return 3000;**

**}**

**};**

**class BMW : public Car**

**{**

**public:**

**string Fit\_parts ()**

**{**

**return "BMW Car";**

**}**

**float Cost()**

**{**

**return 6000;**

**}**

**};**

**// Decorator**

**class CarDecorator: public Car**

**{**

**protected:**

**Car \*m\_Car;**

**public:**

**CarDecorator(Car \*car): m\_Car(car){}**

**string Fit\_parts ()**

**{**

**return m\_Car->Fit\_parts ();**

**}**

**float Cost()**

**{**

**return m\_Car->Cost();**

**}**

**};**

**// Concrete Decorator**

**class With\_music\_system: public CarDecorator**

**{**

**public:**

**With\_music\_system(Car \*car): CarDecorator(car){}**

**string Fit\_parts ()**

**{**

**return m\_Car->Fit\_parts () + " decorated with music system ";**

**}**

**float Cost()**

**{**

**return m\_Car->Cost() + 400;**

**}**

**};**

**class With\_GPS\_navigator: public CarDecorator**

**{**

**public:**

**With\_GPS\_navigator(Car \*car): CarDecorator(car){}**

**string Fit\_parts ()**

**{**

**return m\_Car->Fit\_parts () + " decorated With GPS navigator ";**

**}**

**float Cost()**

**{**

**return m\_Car->Cost() + 800;**

**}**

**};**

**int main()**

**{**

**Car \*audi = new Audi();**

**cout << audi -> Fit\_parts () << endl;**

**cout << audi -> Cost() << endl;**

**Car \*bmw = new BMW();**

**cout << bmw -> Fit\_parts () << endl;**

**cout << bmw -> Cost() << endl;**

**Car \*decoratedCar = new With\_music\_system(bmw);**

**cout << decoratedCar -> Fit\_parts () << endl;**

**cout << decoratedCar -> Cost() << endl;**

**decoratedCar = new With\_GPS\_navigator(audi);**

**cout << decoratedCar -> Fit\_parts () << endl;**

**cout << decoratedCar -> Cost() << endl;**

**return 0;**

**}**

1. **Façade Design pattern**: The Façade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

## **Problem**:

Imagine that you must make your code work with a broad set of objects that belong to a sophisticated library or framework. Ordinarily, you’d need to initialize all those objects, keep track of dependencies, execute methods in the correct order, and so on.

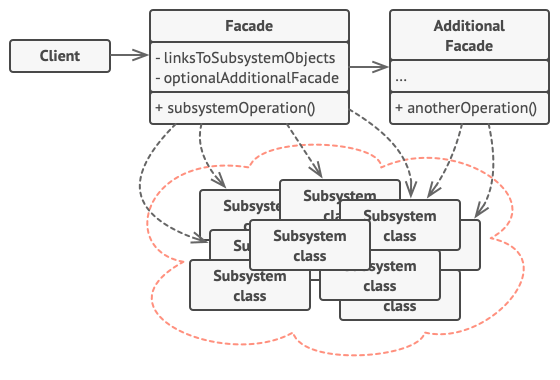
As a result, the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain.

**Solution**: A facade is a class that provides a simple interface to a complex subsystem that contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about.

Having a facade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.

For instance, an app that uploads short funny videos with cats to social media could potentially use a professional video conversion library. However, all that it really needs is a class with the single method encode(filename, format). After creating such a class and connecting it with the video conversion library, you’ll have your first facade.

**Structure:**



1. The **Facade** provides convenient access to a particular part of the subsystem’s functionality. It knows where to direct the client’s request and how to operate all the moving parts.
2. An **Additional Facade** class can be created to prevent polluting a single facade with unrelated features that might make it yet another complex structure. Additional facades can be used by both clients and other facades.
3. The **Complex Subsystem** consists of dozens of various objects. To make them all do something meaningful, you have to dive deep into the subsystem’s implementation details, such as initializing objects in the correct order and supplying them with data in the proper format.

Subsystem classes aren’t aware of the facade’s existence. They operate within the system and work with each other directly.

1. The **Client** uses the facade instead of calling the subsystem objects directly.

## 

**Example:**

**#include <iostream>**

**using** **namespace** std;

**class** Guitar\_Subsystem1

{

**public**:

**void** guitarcable() { cout << " Plug in electric guitar cable in the switch \n";}

**void** adjustment(){ cout << " String adjustment of the guitar\n";}

**void** amplifier (){ cout << " Connect amplifier for audio quality\n";}

};

**class** Vocals\_Subsystem2

{

**public**:

**void** Vocalartist (){ cout << " Vocal artist for a concert\n";}

**void** microphone() { cout << " Set microphone with stand\n";}

**void** Pluginmicrophone () { cout << " Plugin microphone in switch\n";}

};

**class** Drums\_Subsystem3

{

**public**:

**void** drumstick (){ cout << " Get two drum sticks\n";}

**void** drumsstand() { cout << " Set drums stand\n";}

**void** drummerchair () { cout << " Set chair for a drummer \n";}

};

**class** concert\_Facade

{

**private**:

Guitar\_Subsystem1 guitar;

Vocals\_Subsystem2 vocals;

Drums\_Subsystem3 drums;

**public**:

**void** Playconcert()

{

guitar.guitarcable();

guitar.adjustment();

guitar.amplifier();

vocals.Vocalartist();

vocals.microphone();

vocals.Pluginmicrophone();

drums.drumstick ();

drums.drumsstand();

drums.drummerchair();

}

};

**int** main()

{

concert\_Facade facade;

facade.Playconcert();

**return** 0;

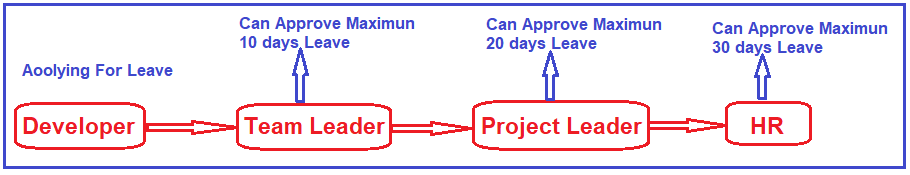
}

**Behavioral Design patterns**

**1) Chain of Responsibility design pattern:** The Chain of Responsibility is a Behavioral design pattern that lets you process the requests along a chain of Handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

**Real-Time Example:**

Let us understand this with an example. Please have a look at the following diagram which shows the reporting hierarchy in a software organization. As you can see, the Developer is reporting to Team Leader. The Team Leader is reporting Project Leader and Project Leader is reporting to HR. Again, the Team Leader can approve a maximum of 10 days’ leave. The Project Leader can approve leave for up to a maximum of 20 Days and the HR can approve leave for a maximum of 30 Days.



Suppose a developer wants to take a leave of 25 days. So, what the developer will do is he will send a request for 25 days’ leave to the Team Leader. The Team Leader will check whether he can approve the leave or not. As the leave is for 25 days and he can only approve up to 10 days, so he will forward the request to the Project Leader.

The Project Leader will check whether he can approve the leave or not. As the leave is for 25 days so, the Project Leader will not approve the leave as his capacity is up to 20 days. So, what the project leader will do is he will pass the request to HR, and HR will check and approve the leave.

On the other hand, if the developer is asking for 5 days’ leave then this can be handled by the Team Leader and once the Team Leader handles this, he will not forward this request to Project Leader.

So, the point that you need to remember is once the request is handled by any handler then it should not forward that request to the next handler.

## **Pros and Cons**

* You can control the order of request handling.
* *Single Responsibility Principle*. You can decouple classes that invoke operations from classes that perform operations.
* *Open/Closed Principle*. You can introduce new handlers into the app without breaking the existing client code.
* Some requests may end up unhandled.

**Example**:

#include <iostream>

#include <string>

using namespace std;

class Handler

{

public:

virtual Handler\* setNext(Handler \*handler) = 0;

virtual std::string Handle(int ndays) = 0;

};

class LeaveHandler : public Handler

{

Handler \*next\_handler;

public:

LeaveHandler():next\_handler(nullptr){}

Handler\* setNext(Handler \*handler) override

{

this->next\_handler = handler;

return handler;

}

std::string Handle(int days)

{

if(this->next\_handler)

{

return this->next\_handler->Handle(days);

}

return " ";

}

};

class TeamLeader: public LeaveHandler

{

public:

std::string Handle(int days)

{

if(days <= 10)

{

return "TeamLead Approved the leave request for : "+std::to\_string(days)+" days";

}

else

{

return LeaveHandler::Handle(days);

}

}

};

class Manager: public LeaveHandler

{

public:

std::string Handle(int days)

{

if(days <= 20)

{

return "Manager Approved the leave request for : "+std::to\_string(days)+" days";

}

else

{

return LeaveHandler::Handle(days);

}

}

};

class HR: public LeaveHandler

{

public:

std::string Handle(int days)

{

if(days <= 30)

{

return "HR Approved the leave request for : "+std::to\_string(days)+" days";

}

else

{

return LeaveHandler::Handle(days);

}

}

};

class LeaveRequestHandler

{

public:

void leaveRequest(Handler &handler, int ndays)

{

std::string result = handler.Handle(ndays);

if(!result.empty())

{

cout<<result<<endl;

}

else

{

cout<<"Request in invalid or rejected : "<<endl;

}

}

};

int main()

{

TeamLeader \*teamLead = new TeamLeader();

Manager \*manager = new Manager();

HR \*Hr = new HR();

teamLead->setNext(manager)->setNext(Hr);

LeaveRequestHandler obj;

obj.leaveRequest(\*teamLead,20);

return 0;

}

**2) Iterator Design Patterns:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

## **Problem**

* Need to access the elements of an aggregate object without exposing the internal details of the object.

Accessing elements of the object is required with traversal, and at the same time, it should not expose the internal structure of an aggregate object. Also, this may be applicable to different data structures, which provide different algorithms for accessing and traversing.

## **Solution**

* Provide a way to access and traverse an aggregate object by giving the responsibility of access and traversal to another object.

So, there can be a common interface, which can be applicable for different data structures to provide an interface for accessing and traversing. There will be separate objects having this interface and will have a responsibility to access the elements and traversal aggregate objects.

**Example**:

#include <iostream>

#include <list>

using namespace std;

class Product {

private:

string product\_name;

int quantity;

public:

//Constructor to initialize the Product

Product(string \_product\_name, int \_quantity){

product\_name = \_product\_name;

quantity = \_quantity;

}

//Printing any Product

string toString() {

return "Product Name: " + product\_name + ", Quantity: #" + to\_string(quantity);

}

};

template <typename Container>

class Iterator {

private:

//To point the associated cart object

Container \*containerData;

//To use as iterator

list<Product>::iterator iter;

public:

//Constructor to initialize the Iterator by providing the container data and setting the iterator to beginning of container

Iterator(Container \*\_containerData){

containerData = \_containerData;

iter = containerData->cartList.begin();

}

//Function to check whether there is some item left for the iterator or not

bool hasNext() {

return (iter != containerData->cartList.end());

}

//Move iterator to next item, it will also return the current item

list<Product>::iterator next() {

return iter++;

}

};

class Cart {

private:

//Underlying container to store the cart products

std::list<Product> cartList;

public:

//Method to add product in cart(infact a list)

void addProduct(Product a) {

cartList.push\_back(a);

}

//Method to create an Iterator for particular cart

Iterator<Cart>\* createIterator() {

return new Iterator<Cart>(this);

}

//Because we are using members of Iterator class from Cart Class that's why it is necessary to make it friend

friend class Iterator<Cart>;

};

//Follow the same sequence of defining class as given in the article, because they have interdependencies

int main() {

//Create Cart

Cart cartProducts;

//Create Products

Product a("Brush",4), b("Tooth Paste", 5), c("Chair", 8);

//Add Products

cartProducts.addProduct(a);

cartProducts.addProduct(b);

cartProducts.addProduct(c);

//Create Iterator for Cart

Iterator<Cart> \*it = cartProducts.createIterator();

//Loop, Access and Print

while(it->hasNext()){

cout << it->next()->toString() << endl;

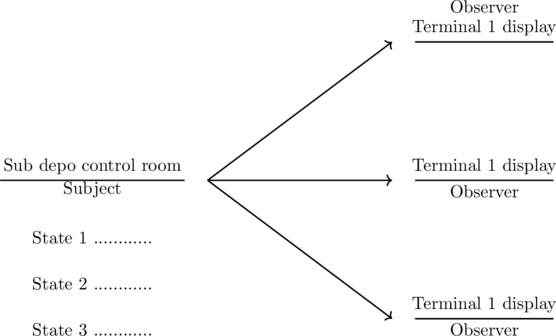
}

return 0;

}

**3) Observer Design Pattern:** The Observer is a behavioral design pattern that let you define a subscription mechanism to notify multiple objects about any event that happens to the object they’re observing.

#### Intent: There will be one too many dependencies among different objects if the subject state changes its state then all its dependents will be updated and notified



Suppose that we have the bus depo station in a city, and we consider it a subject which has three terminals to display time info of the bus leaving from station to passenger

The terminals which display the bus leaving time to passengers we consider it as an observer

Now, the problem with the bus depo station is that it constantly changes its internal info states from 1 to some state

**Why?**

Because buses will come and go to their destination and its time of leaving buses changes constantly then how do we update all the terminals accurate information of buses?

The answer is by using an observer design pattern through which we can attach and detach bus info on the terminal display and can notify

**Example**:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

class Terminal

{

public:

virtual void new\_update\_show(float ftime) = 0;

};

class BusDepo : public Terminal

{

std::string strname;

float ftime;

public:

BusDepo(std::string strname):strname(strname)

{

}

void new\_update\_show(float ftime)

{

cout << "Bus at "<< strname << " Will leave at "<< ftime << "\n";

}

};

class BusDepoOperationSubject

{

vector<BusDepo\*> veclist;

public:

void AttachInfo(BusDepo\* busdepo)

{

veclist.push\_back(busdepo);

}

void DetachInfo(BusDepo\* busdepo)

{

veclist.erase(std::remove(veclist.begin(),veclist.end(), busdepo));

}

void notifyInfo(float ftime)

{

for(vector<BusDepo\*>::iterator iter = veclist.begin(); iter != veclist.end(); ++iter)

{

(\*iter)->new\_update\_show(ftime);

}

}

};

class updateTimeinfo:public BusDepoOperationSubject

{

public:

void changeTime(float fTime)

{

notifyInfo(fTime);

}

};

int main()

{

updateTimeinfo Display;

BusDepo Terminal1("Terminal1");

BusDepo Terminal2("Terminal2");

BusDepo Terminal3("Terminal3");

Display.AttachInfo(&Terminal1);

Display.AttachInfo(&Terminal2);

Display.AttachInfo(&Terminal3);

Display.changeTime(2.45);

Display.DetachInfo(&Terminal2);

Display.changeTime(1.45);

return 0;

}

3) **Observer Design pattern:**

The Observer is a behavioral design pattern. It defines as A one-to-many dependency between objects so that when one object changes its state, all its dependents are notified and updated automatically.

4) **Mediator design pattern:** Mediator is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communication between the objects and forces them to collaborate only via a mediator.

5) **Strategy design pattern**: strategy design pattern that enables an algorithm behavior to be selected at runtime.

**Creational Design Pattern:**

**1) Singleton Design Pattern:** The main purpose of a singleton design pattern is that it will ensure a class should have only one instance of it and allow global access to it.

Example:

**#include <iostream>**

**using** **namespace** std;

**class** Singleton {

**private**:

**static** Singleton \*instance;

**int** data;

**protected**:

Singleton() {

data = 0;

}

**public**:

**static** Singleton \*getInstance() {

**if** (instance==0){instance = new Singleton;}

**return** instance;

}

**int** getData() {**return** **this** -> data;}

**void** setData(**int** data) {**this** -> data = data;}

};

Singleton \*Singleton::instance = 0;

**int** main(){

Singleton \*s = s->getInstance();

Singleton \*s1 = s1->getInstance();

Singleton \*s2 = s2->getInstance();

cout << s->getData() << endl;

s->setData(100);

s1->setData(200);

s2->setData(300);

cout << s->getData() << endl;

cout << s1->getData() << endl;

cout << s2->getData() << endl;

**return** 0;

}

2) **Factory Design Pattern:** The Factory design pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.

It creates an object for you, rather you initiate the object directly.

Define an interface or an abstract class for creating an object but let the subclasses decide which class to initiate.

Diagram

Description automatically generated

Code : <https://github.com/cppnuts-yt/designpattern>

3) **Abstract Factory Design Pattern:** The AFDP defines an abstract class for creating families of related objects but without specifying their concrete sub-class.

Diagram

Description automatically generated

4) **Prototype Design Pattern**: The Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

* Creating an object is a more expensive operation than copying an object.
* All objects’ initial state is common and takes time to build.
* Composition, Creation, and representation of objects should be decoupled from the system.
* Hide the complexity of creating a new instance from the user.
* Which classes to create are specified at runtime.

Example: In games, guns fire bullets, but before it come out from gun it needs to be prepared.

Diagram

Description automatically generated

5) **Builder Design Pattern**:

I want to create an object(plain) by composing other complex objects(body and engine) and I want to create those complex objects step by step.