# Adapter Design Pattern, An adapter helps two incompatible interfaces to work together. This is the real world definition for an adapter. Adapter design pattern is used when you want two different classes with incompatible interfaces to work together. The name says it all. Interfaces may be incompatible but the inner functionality should suit the need.

# In real world the easy and simple example that comes to mind for an adapter is the travel power adapter. American socket and plug are different from British. Their interface are not compatible with one another. British plugs are cylindrical and American plugs are recangularish. You can use an adapter in between to fit an American (rectangular) plug in British (cylindrical) socket assuming voltage requirements are met with.

**public** **interface** Sorter {

**public** **int**[] sort(**int**[] numbers);

}

**public** **class** NumberSorter {

**public** List<Integer> sort(List<Integer> numbers) {

// sort and return

**return** **new** ArrayList<Integer>();

}

}

**public** **class** SortListAdapter **implements** Sorter {

@Override

**public** **int**[] sort(**int**[] numbers) {

// convert the array to a List

List<Integer> numberList = **new** ArrayList<Integer>();

// call the adapter

NumberSorter sorter = **new** NumberSorter();

numberList = sorter.sort(numberList);

// convert the list back to an array and return

**int**[] sortedNumbers = **new** **int**[numberList.size()];

**return** sortedNumbers;

}

}

**public** **class** Client {

**public** **static** **void** main(String[] args) {

**int**[] numbers = **new** **int**[] { 34, 2, 4, 12, 1 };

Sorter sorter = **new** SortListAdapter();

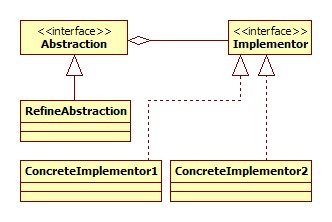
sorter.sort(numbers);

}

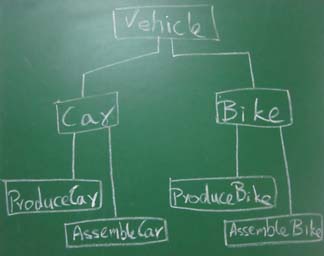
}

# Bridge Design Pattern Decouple an abstraction from its implementation so that the two can vary independently” is the intent for bridge design pattern as stated by GoF.

## **Elements of Bridge Design Pattern**

* Abstraction – core of the bridge design pattern and defines the crux. Contains a reference to the implementer.
* Refined Abstraction – Extends the abstraction takes the finer detail one level below. Hides the finer elements from implemetors.
* Implementer – This interface is the higer level than abstraction. Just defines the basic operations.
* Concrete Implementation – Implements the above implementer by providing concrete implementation

## **Before Bridge Design Pattern**



## **After Bridge Design Pattern**



**abstract** **class** Vehicle {

**protected** Workshop workShop1;

**protected** Workshop workShop2;

**protected** Vehicle(Workshop workShop1, Workshop workShop2) {

**this**.workShop1 = workShop1;

**this**.workShop2 = workShop2;

}

**abstract** **public** **void** manufacture();

}

**public** **class** Car **extends** Vehicle {

**public** Car(Workshop workShop1, Workshop workShop2) {

**super**(workShop1, workShop2);

}

@Override

**public** **void** manufacture() {

System.***out***.print("Car ");

workShop1.work();

workShop2.work();

}

}

**public** **class** Bike **extends** Vehicle {

**public** Bike(Workshop workShop1, Workshop workShop2) {

**super**(workShop1, workShop2);

}

@Override

**public** **void** manufacture() {

System.***out***.print("Bike ");

workShop1.work();

workShop2.work();

}

}

**public** **interface** Workshop {

**abstract** **public** **void** work();

}

**public** **class** Produce **implements** Workshop {

@Override

**public** **void** work() {

System.***out***.print("Produced");

}

}

**public** **class** Assemble **implements** Workshop {

@Override

**public** **void** work() {

System.***out***.println(" Assembled.");

}

}

**public** **class** BridgePattern {

**public** **static** **void** main(String[] args) {

Vehicle vehicle1 = **new** Car(**new** Produce(), **new** Assemble());

vehicle1.manufacture();

Vehicle vehicle2 = **new** Bike(**new** Produce(), **new** Assemble());

vehicle2.manufacture();

}

}

**Composite Design Pattern** when we want to represent part-whole hierarchy, use tree structure and compose objects. We know tree structure what a tree structure is and some of us don’t know what a part-whole hierarchy is. A system consists of subsystems or components. Components can further be divided into smaller components. Further smaller components can be divided into smaller elements. This is a part-whole hierarchy.

Everything around us can be a candidate for part-whole hierarchy. Human body, a car, a computer, lego structure, etc. A car is made up of engine, tyre, … Engine is made up of electrical components, valves, … Electrical components is made up of chips, transistor, … Like this a component is part of a whole system. This hierarchy can be represented as a tree structure using

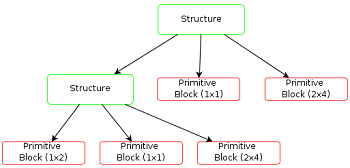
***“Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.” is the intent by GoF***

## **Important Points**

* Importance of composite pattern is, the group of objects should be treated similarly as a single object.
* Manipulating a single object should be as similar to manipulating a group of objects. In sync with our example, we join primitive blocks to create structures and similarly join structures to create house.
* Recursive formation and tree structure for composite should be noted.
* Clients access the whole hierarchy through the components and they are not aware about if they are dealing with leaf or composites.

## **Tree for Composite**

When we get a recursive structure the obvious choice for implementation is a tree. In composite design pattern, the part-whole hierarchy can be represented as a tree. Leaves (end nodes) of a tree being the primitive elements and the tree being the composite structure.



## **Uml Design for Composite Pattern**

Component: (structure)

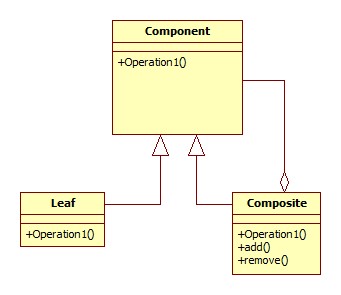
1. Component is at the top of hierarchy. It is an abstraction for the composite.
2. It declares the interface for objects in composition.
3. (optional) defines an interface for accessing a component’s parent in the recursive structure, and implements it if that’s appropriate.

Leaf: (primitive blocks)

1. The end nodes of the tree and will not have any child.
2. Defines the behaviour for single objects in the composition

Composite: (group)

1. Consists of child components and defines behaviour for them
2. Implements the child related operations.



**public** **interface** Group {

**public** **void** assemble();

}

**public** **class** Block **implements** Group {

**public** **void** assemble() {

System.***out***.println("Block");

}

}

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Structure **implements** Group {

// Collection of child groups.

**private** List<Group> groups = **new** ArrayList<>();

**public** **void** assemble() {

**for** (Group group : groups) {

group.assemble();

}

}

// Adds the group to the structure.

**public** **void** add(Group group) {

groups.add(group);

}

// Removes the group from the structure.

**public** **void** remove(Group group) {

groups.remove(group);

}

}

**public** **class** ImplementComposite {

**public** **static** **void** main(String[] args) {

// Initialize three blocks

Block block1 = **new** Block();

Block block2 = **new** Block();

Block block3 = **new** Block();

// Initialize three structure

Structure structure = **new** Structure();

Structure structure1 = **new** Structure();

Structure structure2 = **new** Structure();

// Composes the groups

structure1.add(block1);

structure1.add(block2);

structure2.add(block3);

structure.add(structure1);

structure.add(structure2);

structure.assemble();

}

}

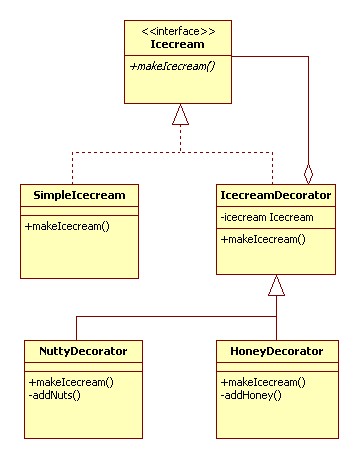
**Decorator Design Pattern**  To extend or modify the behaviour of ‘an instance’ at runtime decorator [design pattern](http://javapapers.com/design-patterns/introduction-to-design-patterns/) is used. Inheritance is used to extend the abilities of ‘a class’. Unlike inheritance, you can choose any single object of a class and [modify its behaviour](http://javapapers.com/core-java/overloading-and-overriding/) leaving the other instances unmodified.

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

In implementing the decorator pattern you construct a wrapper around an object by extending its behavior. The wrapper will do its job before or after and delegate the call to the wrapped instance.

## **Decorator Design Pattern – UML Diagram**



**public** **interface** Icecream {

**public** String makeIcecream();

}

**public** **class** SimpleIcecream **implements** Icecream {

@Override

**public** String makeIcecream() {

**return** "Base Icecream";

}

}

**abstract** **class** IcecreamDecorator **implements** Icecream {

**protected** Icecream specialIcecream;

**public** IcecreamDecorator(Icecream specialIcecream) {

**this**.specialIcecream = specialIcecream;

}

**public** String makeIcecream() {

**return** specialIcecream.makeIcecream();

}

}

**public** **class** HoneyDecorator **extends** IcecreamDecorator {

**public** HoneyDecorator(Icecream specialIcecream) {

**super**(specialIcecream);

}

**public** String makeIcecream() {

**return** specialIcecream.makeIcecream() + addHoney();

}

**private** String addHoney() {

**return** " + sweet honey";

}

}

**public** **class** NuttyDecorator **extends** IcecreamDecorator {

**public** NuttyDecorator(Icecream specialIcecream) {

**super**(specialIcecream);

}

**public** String makeIcecream() {

**return** specialIcecream.makeIcecream() + addNuts();

}

**private** String addNuts() {

**return** " + cruncy nuts";

}

}

**public** **class** TestDecorator {

**public** **static** **void** main(String args[]) {

Icecream icecream = **new** HoneyDecorator(**new** NuttyDecorator(

**new** SimpleIcecream()));

System.***out***.println(icecream.makeIcecream());

}

}

**Facade Design Pattern**, GOF definition for facade design pattern is, “Provide a unified interface to a set of interfaces in a subsystem. Facade Pattern defines a higher-level interface that makes the subsystem easier to use.”

How do we infer the above definition? Think of a component that solves a complex business problem. That component may expose lot of interfaces to interact with it. To complete a process flow we may have to interact with multiple interfaces.

**public** **class** FlightBooker {

**public** ArrayList<Flight> getFlightsFor(Date from, Date to) {

// returns flights available in the particular date range

**return** **null**;

}

}

**public** **class** HotelBooker {

**public** ArrayList<Hotel> getHotelNamesFor(Date from, Date to) {

// returns hotels available in the particular date range

**return** **null**;

}

}

**public** **class** TravelFacade {

**private** HotelBooker hotelBooker;

**private** FlightBooker flightBooker;

**public** **void** getFlightsAndHotels(Date from, Date to) {

ArrayList<Flight> flights = flightBooker.getFlightsFor(from, to);

ArrayList<Hotel> hotels = hotelBooker.getHotelNamesFor(from, to);

}

}

**public** **class** Client {

**public** **static** **void** main(String[] args) {

TravelFacade facade = **new** TravelFacade();

facade.getFlightsAndHotels(**new** Date(), **new** Date());

}

}

**Proxy Pattern*,***In proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern.

In proxy pattern, we create object having original object to interface its functionality to outer world.

# Proxy Pattern UML Diagram

**public** **interface** Image {

**void** display();

}

**public** **class** RealImage **implements** Image {

**private** String fileName;

**public** RealImage(String fileName) {

**this**.fileName = fileName;

loadFromDisk(fileName);

}

@Override

**public** **void** display() {

System.***out***.println("Displaying " + fileName);

}

**private** **void** loadFromDisk(String fileName) {

System.***out***.println("Loading " + fileName);

}

}

**public** **class** ProxyImage **implements** Image {

**private** RealImage realImage;

**private** String fileName;

**public** ProxyImage(String fileName) {

**this**.fileName = fileName;

}

@Override

**public** **void** display() {

**if** (realImage == **null**) {

realImage = **new** RealImage(fileName);

}

realImage.display();

}

}

**public** **class** ProxyPatternDemo {

**public** **static** **void** main(String[] args) {

Image image = **new** ProxyImage("test\_10mb.jpg");

// image will be loaded from disk

image.display();

System.***out***.println("");

// image will not be loaded from disk

image.display();

}

}

# Command Pattern, command design pattern is used to encapsulate a request as an object and pass to an invoker, wherein the invoker does not knows how to service the request but uses the encapsulated command to perform an action.

# Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

# A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

# One example of the command pattern being executed in the real world is the idea of a table order at a restaurant: the waiter takes the order, which is a command from the customer.This order is then queued for the kitchen staff. The waiter tells the chef that the a new order has come in, and the chef has enough information to cook the meal.

To understand command design pattern we should understand the associated key terms like client, command, command implementation, invoker, receiver.

* Command is an interface with execute method. It is the core of contract.
* A client creates an instance of a command implementation and associates it with a receiver.
* An invoker instructs the command to perform an action.
* A Command implementation’s instance creates a binding between the receiver and an action.
* Receiver is the object that knows the actual steps to perform the action.

# Image title

//Command

**public** **interface** Command {

**public** **void** execute();

}

//Concrete Command

**public** **class** LightOffCommand **implements** Command {

// reference to the light

Light light;

**public** LightOffCommand(Light light) {

**this**.light = light;

}

**public** **void** execute() {

light.switchOff();

}

}

//Concrete Command

**public** **class** LightOnCommand **implements** Command {

// reference to the light

Light light;

**public** LightOnCommand(Light light) {

**this**.light = light;

}

**public** **void** execute() {

light.switchOn();

}

}

//Receiver

**public** **class** Light {

**private** **boolean** on;

**public** **void** switchOn() {

on = **true**;

}

**public** **void** switchOff() {

on = **false**;

}

}

//Invoker

**public** **class** RemoteControl {

**private** Command command;

**public** **void** setCommand(Command command) {

**this**.command = command;

}

**public** **void** pressButton() {

command.execute();

}

}

//Client

**public** **class** Client {

**public** **static** **void** main(String[] args) {

RemoteControl control = **new** RemoteControl();

Light light = **new** Light();

Command lightsOn = **new** LightOnCommand(light);

Command lightsOff = **new** LightOffCommand(light);

// switch on

control.setCommand(lightsOn);

control.pressButton();

// switch off

control.setCommand(lightsOff);

control.pressButton();

}

}

### Iterator Pattern,  this design pattern tutorial series we will discuss about iterator design pattern, which allows to traverse a collection without exposing its internals. Ah we all know about the Java Iterator. But it is not all about it, lets dive deeper and do a comprehensive study of iterator pattern. By the end of this tutorial we will know about internal iterator, external iterator, polymorphic iterator, robust iterator, null iterator and etc.

**public** **interface** Iterator {

**public** **boolean** hasNext();

**public** Object next();

}

**public** **interface** Container {

**public** Iterator getIterator();

}

**public** **class** NameRepository **implements** Container {

**public** String names[] = { "Robert", "John", "Julie", "Lora" };

@Override

**public** Iterator getIterator() {

**return** **new** NameIterator();

}

**private** **class** NameIterator **implements** Iterator {

**int** index;

@Override

**public** **boolean** hasNext() {

**if** (index < names.length) {

**return** **true**;

}

**return** **false**;

}

@Override

**public** Object next() {

**if** (**this**.hasNext()) {

**return** names[index++];

}

**return** **null**;

}

}

}

**public** **class** IteratorPatternDemo {

**public** **static** **void** main(String[] args) {

NameRepository namesRepository = **new** NameRepository();

**for** (Iterator iter = namesRepository.getIterator(); iter.hasNext();) {

String name = (String) iter.next();

System.***out***.println("Name : " + name);

}

}

}

**Observer Design Pattern** observer design pattern multiple observer objects registers with a subject for change notification. When the state of subject changes, it notifies the observers. Objects that listen or watch for change are called observers and the object that is being watched for is called subject.

Pattern involved is also called as publish-subscribe pattern. Model view controller (MVC) architecture’s core uses the observer design pattern.

**public** **interface** Subject {

// methods to register and unregister observers

**public** **void** register(Observer obj);

**public** **void** unregister(Observer obj);

// method to notify observers of change

**public** **void** notifyObservers();

// method to get updates from subject

**public** Object getUpdate(Observer obj);

}

**public** **interface** Observer {

// method to update the observer, used by subject

**public** **void** update();

// attach with subject to observe

**public** **void** setSubject(Subject sub);

}

**public** **class** MyTopic **implements** Subject {

**private** List<Observer> observers;

**private** String message;

**private** **boolean** changed;

**private** **final** Object MUTEX = **new** Object();

**public** MyTopic() {

**this**.observers = **new** ArrayList<>();

}

@Override

**public** **void** register(Observer obj) {

**if** (obj == **null**)

**throw** **new** NullPointerException("Null Observer");

**synchronized** (MUTEX) {

**if** (!observers.contains(obj))

observers.add(obj);

}

}

@Override

**public** **void** unregister(Observer obj) {

**synchronized** (MUTEX) {

observers.remove(obj);

}

}

@Override

**public** **void** notifyObservers() {

List<Observer> observersLocal = **null**;

// synchronization is used to make sure any observer registered after

// message is received is not notified

**synchronized** (MUTEX) {

**if** (!changed)

**return**;

observersLocal = **new** ArrayList<>(**this**.observers);

**this**.changed = **false**;

}

**for** (Observer obj : observersLocal) {

obj.update();

}

}

@Override

**public** Object getUpdate(Observer obj) {

**return** **this**.message;

}

// method to post message to the topic

**public** **void** postMessage(String msg) {

System.***out***.println("Message Posted to Topic:" + msg);

**this**.message = msg;

**this**.changed = **true**;

notifyObservers();

}

}

**public** **class** MyTopicSubscriber **implements** Observer {

**private** String name;

**private** Subject topic;

**public** MyTopicSubscriber(String nm) {

**this**.name = nm;

}

@Override

**public** **void** update() {

String msg = (String) topic.getUpdate(**this**);

**if** (msg == **null**) {

System.***out***.println(name + ":: No new message");

} **else**

System.***out***.println(name + ":: Consuming message::" + msg);

}

@Override

**public** **void** setSubject(Subject sub) {

**this**.topic = sub;

}

}

**public** **class** ObserverPatternTest {

**public** **static** **void** main(String[] args) {

// create subject

MyTopic topic = **new** MyTopic();

// create observers

Observer obj1 = **new** MyTopicSubscriber("Obj1");

Observer obj2 = **new** MyTopicSubscriber("Obj2");

Observer obj3 = **new** MyTopicSubscriber("Obj3");

// register observers to the subject

topic.register(obj1);

topic.register(obj2);

topic.register(obj3);

// attach observer to subject

obj1.setSubject(topic);

obj2.setSubject(topic);

obj3.setSubject(topic);

// check if any update is available

obj1.update();

// now send message to subject

topic.postMessage("New Message");

}

}

**Strategy pattern or Policy Pattern,** we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

**public** **interface** Strategy {

**public** **int** doOperation(**int** num1, **int** num2);

}

**public** **class** OperationAdd **implements** Strategy {

@Override

**public** **int** doOperation(**int** num1, **int** num2) {

**return** num1 + num2;

}

}

**public** **class** OperationMultiply **implements** Strategy {

@Override

**public** **int** doOperation(**int** num1, **int** num2) {

**return** num1 \* num2;

}

}

**public** **class** OperationSubstract **implements** Strategy {

@Override

**public** **int** doOperation(**int** num1, **int** num2) {

**return** num1 - num2;

}

}

**public** **class** Context {

**private** Strategy strategy;

**public** Context(Strategy strategy) {

**this**.strategy = strategy;

}

**public** **int** executeStrategy(**int** num1, **int** num2) {

**return** strategy.doOperation(num1, num2);

}

}

**public** **class** StrategyPatternDemo {

**public** **static** **void** main(String[] args) {

Context context = **new** Context(**new** OperationAdd());

System.***out***.println("10 + 5 = " + context.executeStrategy(10, 5));

context = **new** Context(**new** OperationSubstract());

System.***out***.println("10 - 5 = " + context.executeStrategy(10, 5));

context = **new** Context(**new** OperationMultiply());

System.***out***.println("10 \* 5 = " + context.executeStrategy(10, 5));

}

}

**State pattern,** we create objects which represent various states and a context object whose behavior varies as its state object changes.

**public** **interface** State {

**public** **void** doAction(Context context);

}

**public** **class** StartState **implements** State {

**public** **void** doAction(Context context) {

System.***out***.println("Player is in start state");

context.setState(**this**);

}

**public** String toString() {

**return** "Start State";

}

}

**public** **class** StopState **implements** State {

**public** **void** doAction(Context context) {

System.***out***.println("Player is in stop state");

context.setState(**this**);

}

**public** String toString() {

**return** "Stop State";

}

}

**public** **class** Context {

**private** State state;

**public** Context() {

state = **null**;

}

**public** **void** setState(State state) {

**this**.state = state;

}

**public** State getState() {

**return** state;

}

}

**public** **class** StatePatternDemo {

**public** **static** **void** main(String[] args) {

Context context = **new** Context();

StartState startState = **new** StartState();

startState.doAction(context);

System.***out***.println(context.getState().toString());

StopState stopState = **new** StopState();

stopState.doAction(context);

System.***out***.println(context.getState().toString());

}

}