

Structural Patterns

- Help identify and describe relationship between entities
- Address how classes and objects are composed to form large structures
- Class oriented pattern use inheritance to compose interfaces and implementation
- Object Oriented Patterns describe ways to compose objects to realize new functionality , possibly by changing the composition at runtime.

These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

1. **Adapter:** Match interfaces of different classes. Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
2. **Bridge:** Separates an object's interface from its implementation. Decouple an abstraction from its implementation so that the two can vary independently.
3. **Composite:** A tree structure of simple and composite objects. Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
4. **Decorator:** Add responsibilities to objects dynamically. Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
5. **Facade:** A single class that represents an entire subsystem. Provide a unified interface to a set of interfaces in a system. Facade defines a higher-level interface that makes the subsystem easier to use.
6. **Flyweight:** A fine-grained instance used for efficient sharing. Use sharing to support large numbers of fine-grained objects efficiently. A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context — it's indistinguishable from an instance of the object that's not shared.
7. **Proxy:** An object representing another object. Provide a surrogate or placeholder for another object to control access to it.

BRIDGE

Name:	Bridge
Problem:	To decouple the implementation from its abstraction
Solution: (advice)	Decouple an abstraction from its implementation so that the two can vary independently

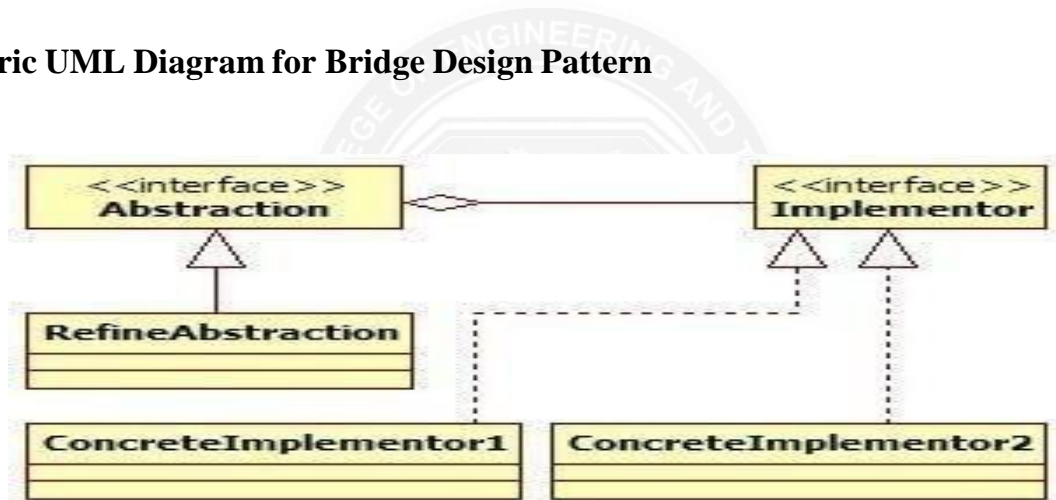
The bridge uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes. Bridge design pattern is a modified version of the notion of “prefer composition over inheritance”.

- Creates two different hierarchies. One for abstraction and another for implementation.
- Avoids permanent binding by removing the dependency between abstraction and implementation.
- We create a bridge that coordinates between abstraction and implementation.
- Abstraction and implementation can be extended separately.
- Should be used when we have need to switch implementation at runtime.
- Client should not be impacted if there is modification in implementation of abstraction.
 - Best used when you have multiple implementations.

Elements of Bridge Design Pattern:

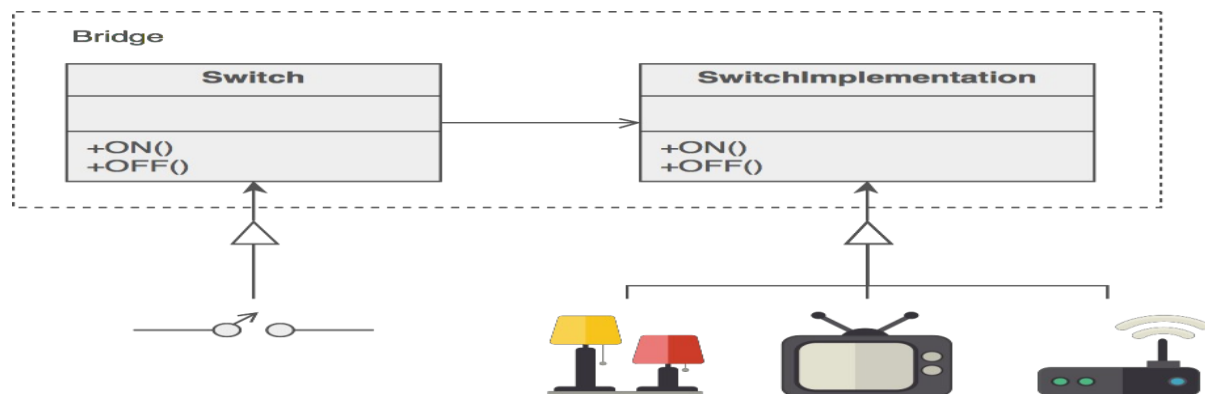
- **Abstraction** – core of the bridge design pattern and defines the crux (Create , Retrieve, Update , Delete) Contains a reference to the implementer.
- **Refined Abstraction** – Extends the abstraction takes the finer detail one level below. Hides the finer elements from implementers.
- **Implementer** – This interface is the higher level than abstraction. Just defines the basic operations.
- **Concrete Implementation** – Implements the above implementer by providing concrete implementation.

Generic UML Diagram for Bridge Design Pattern

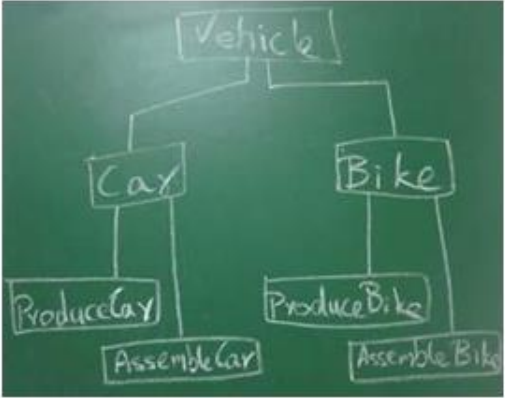
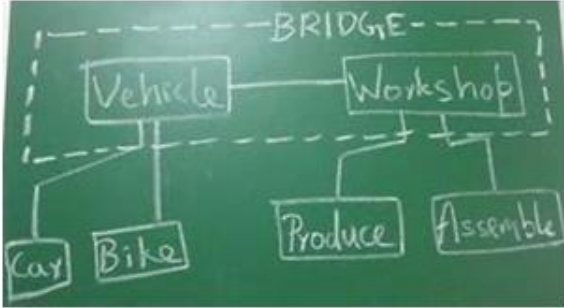


Example :

- A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches.



Need for Bridge Design Pattern

Without Bridge Pattern	With Bridge Pattern
	
<p>When there are inheritance hierarchies creating concrete implementation, you lose flexibility because of interdependence.</p>	<p>Decouple implementation from interface and hiding implementation details from client is the essence of bridge design pattern.</p>

Example : for core elements of Bridge Design Pattern

Vehicle ->
Abstraction
manufacture()

Car -> Refined
Abstraction 1
manufacture()

Bike -> Refined
Abstraction 2
manufacture()

Workshop ->
Implementor work()

Produce -> Concrete
Implementation 1 work()

Assemble -> Concrete
Implementation 2 work()

Example Coding :

```
// abstraction in bridge
pattern abstract class
Vehicle {
    protected Workshop workShop1;
    protected Workshop workShop2;
protected Vehicle(Workshop workShop1, Workshop
                this.workShop1 = workShop1;
                this.workShop2 = workShop2;
    }
    abstract public void manufacture();
}
// Refine abstraction 1 in bridge
pattern public class Car extends
Vehicle {
    public Car(Workshop workShop1, Workshop workShop2) {
        super(workShop1, workShop2);
    }

    public void manufacture() {
        System.out.print("C
ar ");
        workShop1.work();
        workShop2.work();
    } }

public class Bike extends Vehicle {
    public Bike(Workshop workShop1, Workshop workShop2) {
        super(workShop1, workShop2);
    }
    public void manufacture() {
        System.out.print("Bik
e ");
        workShop1.work();
        workShop2.work();
    } }
// Implementor for bridge pattern
public interface Workshop {
    abstract public void work();
}
```

```
//Concrete implementation 1 for bridge
pattern public class Produce implements
Workshop {
    public void work() {
        System.out.print("Produced"
        ); }}

public class Assemble implements Workshop {
    public void work() {
        System.out.println(" Assembled.");
    }
}

//Demonstration of bridge design pattern
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();
    }
}
```

Output

Car Produced Assembled.
Bike Produced Assembled.

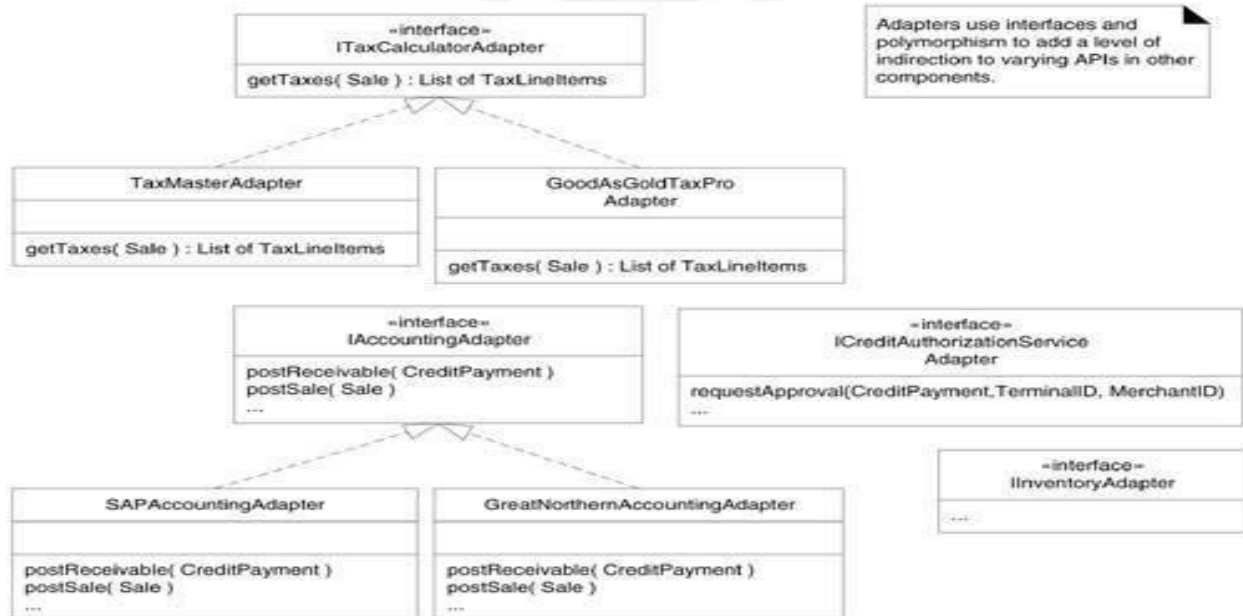
ADAPTER

Name	Adapter
Problem:	How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?
Solution: (advice)	Convert the original interface of a component into another interface, through an intermediate adapter object.

Example:

The NextGen POS system needs to supports many third party services like,

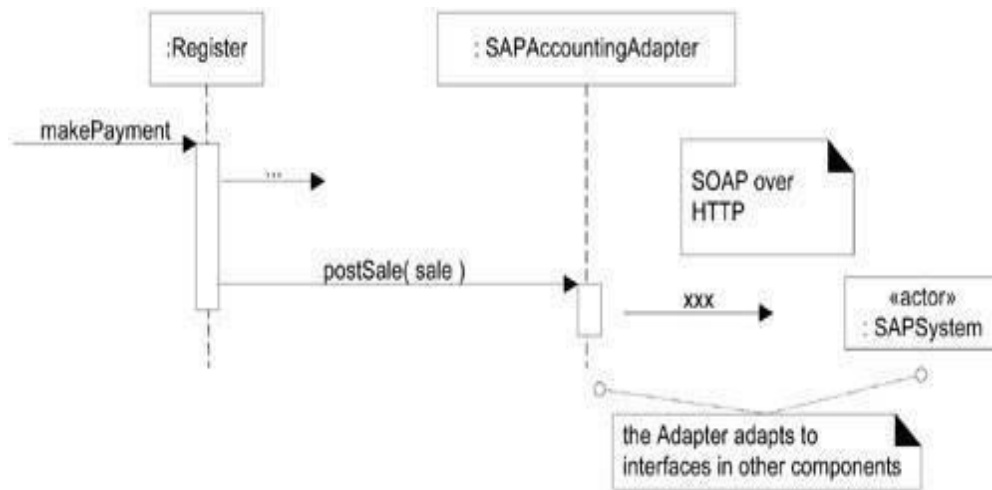
- Tax calculators
- Credit authorization
- Inventory systems
- Accounting systems etc.,



The Adapter pattern

Here a particular adapter instance will be instantiated ,
such as

- SAP for accounting, and will adapt the postSale request to the external interface.
- SOAP XML interface over HTTPS for an intranet Web services.



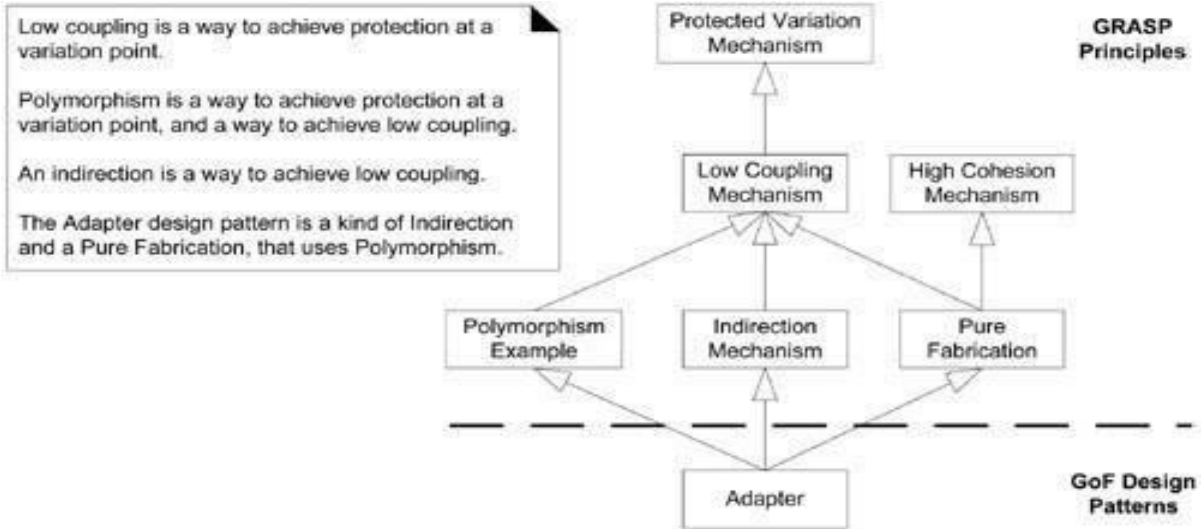
Using an Adapter

The type names include the pattern name "Adapter." This is a relatively common style and has the advantage of easily communicating to others reading the code or diagrams what design patterns are being used.

GRASP Principles as a Generalization of Other Patterns

The Adapter pattern can be viewed as a specialization of some GRASP building blocks. Adapter supports Protected Variations with respect to changing external interfaces or third- party packages through the use of an Indirection object that applies interfaces and Polymorphism.

Example: Adapter and GRASP



Relating Adapter to some core GRASP principles

