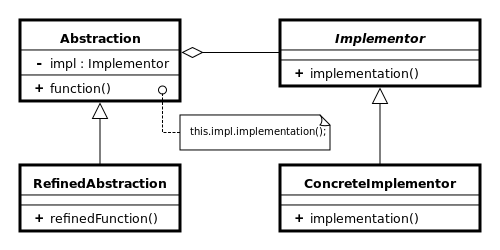
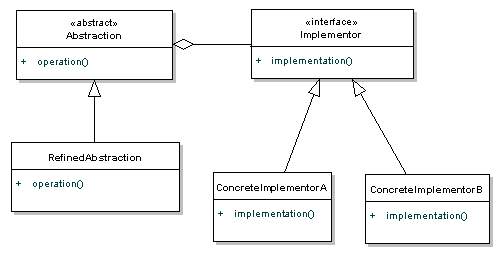
**Bridge Design (Structural) Pattern**

GOF: **Decouple abstractions from its implementations so that the two can vary independently**.

UML Class Diagram



As per DZONE



/\*\* "Abstraction" \*/

**public** **abstract** **class** AbstractShape {

**protected** IDrawingAPI drawingAPI;

**protected** AbstractShape(IDrawingAPI drawingAPI) {

**this**.drawingAPI = drawingAPI;

}

**public** **abstract** **void** draw();

}

/\*\* "Implementor" \*/

**interface** IDrawingAPI {

**public** **void** drawCircle();

}

/\*\* "ConcreteImplementor" 1 \*/

**class** ConcreteDrawingImpl **implements** IDrawingAPI {

**public** **void** drawCircle() {

System.***out***.printf("Drawing circle ... ");

}

}

/\*\* "Refined Abstraction" \*/

**class** CircleShape **extends** AbstractShape {

**public** CircleShape(IDrawingAPI drawingAPI)

{

**super**(drawingAPI);

}

// low-level i.e. Implementation specific

**public** **void** draw() {

drawingAPI.drawCircle();

}

}

**Test Java Program**

/\*\* "Client" \*/

**class** TestBridgePattern {

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

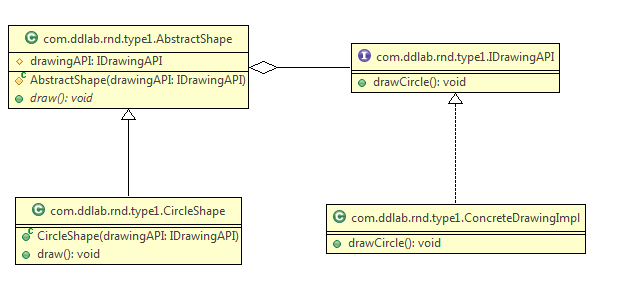
AbstractShape shape = **new** CircleShape(**new** ConcreteDrawingImpl());

shape.draw();

}

}

The class diagram is given below.



**public** **interface** ITV {

**public** **void** on();

**public** **void** off();

**public** **void** switchChannel(**int** channel);

}

**Another example on TV and Remote Control**

**public** **abstract** **class** AbstractRemoteControl {

**private** ITV tv;

**public** AbstractRemoteControl(ITV tv) {

**this**.tv = tv;

}

**public** **void** turnOn(){

tv.on();

}

**public** **void** turnOff(){

tv.off();

}

**public** **void** setChannel(**int** channel){

tv.switchChannel(channel);

}

}

**public** **class** LGTV **implements** ITV {

@Override

**public** **void** on() {

System.***out***.println("Samsung is turned on.");

}

@Override

**public** **void** off() {

System.***out***.println("Samsung is turned off.");

}

@Override

**public** **void** switchChannel(**int** channel) {

System.***out***.println("Samsung: channel - " + channel);

}

}

**public** **class** LGRemoteControl **extends** AbstractRemoteControl {

**public** LGRemoteControl(ITV tv) {

**super**(tv);

}

**public** **void** setChannel(**int** channel){

setChannel(channel);

System.***out***.println("LG uses to set channel.");

}

}

**Test program**

**public** **class** Test {

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

ITV tv = **new** SamsungTV();

LogitechRemoteControl lrc = **new** LogitechRemoteControl(tv);

lrc.setChannel(100);

}

}

In Java we have both AbstractList and List Interface.

When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance. An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways.

**Applicability**

Use the Bridge pattern when

* **You want to avoid a permanent binding between an abstraction and its implementation**. This might be the case, for example, when the implementation must be selected or switched at run-time.
* **both the abstractions and their implementations should be extensible by subclassing**. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
* changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.

The Bridge pattern is a [composite](http://today.java.net/pub/a/today/2004/10/29/patterns.html) of the [Template](http://en.wikipedia.org/wiki/Template_method_pattern) and [Strategy](http://en.wikipedia.org/wiki/Strategy_pattern) patterns.

Another good Example

public interface IServiceApi {  
 List<String> getData(); 🡸 Implementor  
}

public abstract class AbstractProcessor {  
 protected IServiceApi api;  
 public AbstractProcessor(IServiceApi api) { 🡸 Abstraction  
 this.api = api;  
 }  
  
 public abstract void process();  
}

public class ServiceNowApi implements IServiceApi {  
 private String url;  
  
 public ServiceNowApi(String url) {  
 this.url = url;  
 } 🡸 Concrete Implementation  
  
 @Override  
 public List<String> getData() {  
 return Arrays.*asList*("A","B");  
 }  
}

public class BusinessDataProcessor extends AbstractProcessor {  
  
 public DataProcessor(IServiceApi api) {  
 super(api);  
 } 🡸 Concrete Abstraction  
  
 @Override  
 public void process() {  
 List<String> dataList = api.getData();  
 System.*out*.println(dataList);  
 }  
}

**Test class**

public static void main(String[] args) {  
 IServiceApi api = new ServiceNowApi("example.com/abcd");  
 AbstractProcessor processor = new BusinessDataProcessor (api);  
 processor.process();  
}

There will implementation for RackAPI, LCS API etc, it means there can be n number of implementations.

Another Example

public abstract class AbstractValidation { 🡸 Abstraction

    protected IValidator validator;

    public AbstractValidation(IValidator validator) {

        this.validator = validator;

    }

    public abstract void performValidation();

}

public interface IValidator { 🡸 Implementator

    boolean isValid();

}

public class BusinessValidation extends AbstractValidation { 🡸 Refined Abstraction

    public BusinessValidation(IValidator validator) {

        super(validator);

    }

    @Override

    public void performValidation() {

        boolean flag = validator.isValid();

        System.out.println("Flag: "+flag);

    }

}

public class EmailValidator implements IValidator { 🡸 Contrete Implementation

    private String value;

    public StringValidator(String value) {

        this.value = value;

    }

    @Override

    public boolean isValid() {

        return this.value != null;

    }

}

**Test**

public static void main(String[] args) {

        IValidator valdtr = new StringValidator("abcd");

        AbstractValidation absValdn = new BusinessValidation(valdtr);

        absValdn.performValidation();

}