<http://www.journaldev.com/1491/bridge-pattern-in-java-example-tutorial>

**The key benefit of the**[**bridge design pattern**](http://www.devlake.com/)**is that it allows you to develop the Abstraction and the Implementation parts independently.** It also cuts down on the number of classes that you need to create to fulfill all the possible combinations of the Abstractions (user interface concepts) and the Implementations (actual actions behind the scene).

When:

A

/ \

Aa Ab

/ \ / \

Aa1 Aa2 Ab1 Ab2

Refactor to:

A N

/ \ / \

Aa(N) Ab(N) 1 2

here's a combination of [Federico's](http://stackoverflow.com/a/319757/593415) and [John's](http://stackoverflow.com/a/9406293/593415) answers.

When:

----Shape---

/ \

Rectangle Circle

/ \ / \

BlueRectangle RedRectangle BlueCircle RedCircle

Refactor to:

----Shape--- Color

/ \ / \

Rectangle(Color) Circle(Color) Blue Red

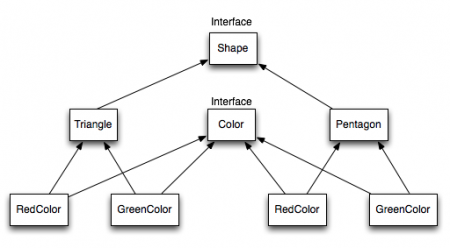
When we have interface hierarchies in both interfaces as well as implementations, then **builder design pattern** is used to decouple the interfaces from implementation and hiding the implementation details from the client programs. Like [Adapter pattern](http://www.journaldev.com/1487/adapter-design-pattern-in-java-example-tutorial), its one of the **Structural design pattern**.

According to GoF bridge design pattern is:

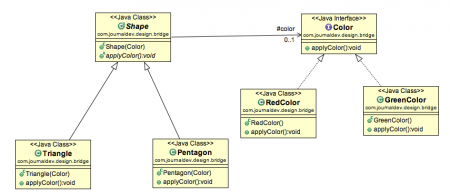
*Decouple an abstraction from its implementation so that the two can vary independently*

The implementation of bridge design pattern follows the notion to prefer [Composition](http://www.journaldev.com/1325/what-is-composition-in-java-java-composition-example) over [inheritance](http://www.journaldev.com/644/inheritance-in-java-example).

If we look into this design pattern with example, it will be easy to understand. Lets say we have an interface hierarchy in both interfaces and implementations like below image.

[](http://cdn.journaldev.com/wp-content/uploads/2013/07/Bridge-Interface-Hierarchy.png)

Now we will use bridge design pattern to decouple the interfaces from implementation and the UML diagram for the classes and interfaces after applying bridge pattern will look like below image.

[](http://cdn1.journaldev.com/wp-content/uploads/2013/07/bridge-design-pattern.png)

Notice the bridge between Shape and Color interfaces and use of composition in implementing the bridge pattern.

Here is the java code for Shape and Color interfaces.

|  |  |
| --- | --- |
| Color.java | |
| 1  2  3  4  5  6 | package com.journaldev.design.bridge;    public interface Color {        public void applyColor();  } |
| Shape.java | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | package com.journaldev.design.bridge;    public abstract class Shape {      //Composition - implementor      protected Color color;        //constructor with implementor as input argument      public Shape(Color c){          this.color=c;      }        abstract public void applyColor();  } |

We have Triangle and Pentagon implementation classes as below.

|  |  |
| --- | --- |
| Triangle.java | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package com.journaldev.design.bridge;    public class Triangle extends Shape{        public Triangle(Color c) {          super(c);      }        @Override      public void applyColor() {          System.out.print("Triangle filled with color ");          color.applyColor();      }    } |
| Pentagon.java | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package com.journaldev.design.bridge;    public class Pentagon extends Shape{        public Pentagon(Color c) {          super(c);      }        @Override      public void applyColor() {          System.out.print("Pentagon filled with color ");          color.applyColor();      }    } |

Here are the implementation classes for RedColor and GreenColor.

|  |  |
| --- | --- |
| RedColor.java | |
| 1  2  3  4  5  6  7  8 | package com.journaldev.design.bridge;    public class RedColor implements Color{        public void applyColor(){          System.out.println("red.");      }  } |
| GreenColor.java | |
| 1  2  3  4  5  6  7  8 | package com.journaldev.design.bridge;    public class GreenColor implements Color{        public void applyColor(){          System.out.println("green.");      }  } |

Lets test our bridge pattern implementation with a test program.

|  |  |
| --- | --- |
| BridgePatternTest.java | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | package com.journaldev.design.test;    import com.journaldev.design.bridge.GreenColor;  import com.journaldev.design.bridge.Pentagon;  import com.journaldev.design.bridge.RedColor;  import com.journaldev.design.bridge.Shape;  import com.journaldev.design.bridge.Triangle;    public class BridgePatternTest {        public static void main(String[] args) {          Shape tri = new Triangle(new RedColor());          tri.applyColor();            Shape pent = new Pentagon(new GreenColor());          pent.applyColor();      }    } |

Output of above class is:

|  |  |
| --- | --- |
| 1  2 | Triangle filled with color red.  Pentagon filled with color green. |

Bridge design pattern can be used when both abstraction and implementation can have different hierarchies independently and we want to hide the implementation from the client application.

**Bridge** – Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. Examples -

* JDBC-ODBC Bridge

**Difference between Bridge pattern and Adapter pattern**

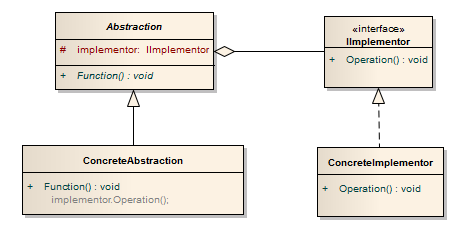
Adapter makes things work after they're designed; Bridge makes them work before they are. [GoF, p219]"

Effectively, the Adapter pattern is useful when you have existing code, be it third party, or in-house, but out of your control, or otherwise not changeable to quite meet the interface you need it to. For instance, we have a SuperWeaponsArray which can control a fine array of doomsday devices.

BEST CODE EXAMPLE

<http://www.codeproject.com/Articles/183655/Bridge-Design-Pattern>

The [**bridge design pattern**](http://www.devlake.com/) allows you to separate the abstraction from the implementation. In the [**bridge pattern**](http://www.devlake.com/), there are 2 parts - the first part is the Abstraction, and the second part is the Implementation. The [**bridge pattern**](http://www.devlake.com/) allows the Abstraction and the Implementation to be developed independently, and the client code can access only the Abstraction part without being concerned about the Implementation part.  
  
Let's look at an example to see the concept behind the [**bridge pattern**](http://www.devlake.com/). For example, inside a house, there are appliances that you can turn on or off, such as the floor lamp, the TV, and the vacuum cleaner. There are different ways to turn the appliance on or off, such as using the on/off switch, the pull switch, or using a remote control. The concept of turning the appliance on or off is the Abstraction part in the [**bridge pattern**](http://www.devlake.com/), and the user only needs to know the Abstraction part. This is the first part of the [**bridge pattern**](http://www.devlake.com/).

[](http://www.devlake.com/)

The left side is the Abstraction part with the following classes:

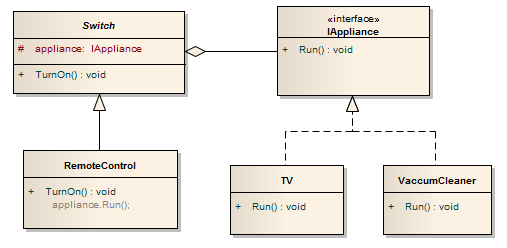
* Abstractionis the abstract parent class of the ConcreteAbstractionclass. It defines the Functionmethod for the client code to call. The protected implementorvariable holds the reference to the object that performs the implementation.
* ConcreteAbstractionis the concrete class that is inherited from the Abstractionclass.

The right side is the Implementation part with the following classes:

* IImplementoris the interface that all the implementation classes must implement.
* ConcreteImplementoris the concrete class that performs the implementation.

Applying this [**UML**](http://www.devlake.com/UML-Quick-Reference) to our example means that the left side represents controls that can turn appliances on or off, such as the on/off switch, the pull switch, or the remote control, while the right side represents the actual appliances that performs the action, such as the TV or the VacuumCleaner.

Therefore, the [**UML**](http://www.devlake.com/UML-Quick-Reference) of our example will be:

[](http://www.devlake.com/)

Another example of the [**bridge pattern**](http://www.devlake.com/) is the copy and paste function in many applications. The copy and paste function is the abstraction, where the user only needs to know how to use it, and the actual action of transferring the information into the memory and transferring the information onto the application is the implementation.

**The key benefit of the**[**bridge design pattern**](http://www.devlake.com/)**is that it allows you to develop the Abstraction and the Implementation parts independently.** It also cuts down on the number of classes that you need to create to fulfill all the possible combinations of the Abstractions (user interface concepts) and the Implementations (actual actions behind the scene).

Below are the implementation code and the output using the example given. Notice that the client code uses only the Abstraction part to perform the actions, and you can develop the Abstraction and the Implementation parts independently: