

**Advantages**

1. Bridge pattern decouple an abstraction from its implementation so that the two can vary independently.
2. It is used mainly for implementing platform independence feature.
3. It adds one more method level redirection to achieve the objective.
4. Publish abstraction interface in separate inheritance hierarchy, and put implementation in its own inheritance hierarchy.
5. Use bridge pattern to run-time binding of the implementation.
6. Use bridge pattern to map orthogonal class hierarchies
7. Bridge is designed up-front to let the abstraction and the implementation vary independently.

Bridge Design Pattern Explained

By [Pankaj K](http://ikeptwalking.com/author/o2m4d1k0c1c1amdkdk/) | September 1, 2015

[0 Comment](http://ikeptwalking.com/bridge-design-pattern-explained/#respond)

This tutorial is a C++ implementation of the Bridge design pattern. This is one of the most confusing design pattern. This and the other patterns, namely [Adapter](http://ikeptwalking.com/adapter-design-pattern-explained/), [Proxy](http://ikeptwalking.com/proxy-design-pattern-explained/) and [Decorator](http://ikeptwalking.com/decorator-pattern-explained/) design pattern have too much in common to confuse a simple soul. As usual I will first discuss the WHY and then later on the HOW.  
  
**WHY**  
If you have a background in Mathematics then you must have heard of Permutations and Combinations. I would have loved to explain it here but the fact is that I am bad at Maths. So I will limit myself to saying the Permutations is all about multiplying and Combinations is all about adding. (Yes that is the level of my Maths)

6 \* 6 = 36 permutations.  
6 + 6 = 12 combinations.

If the count increases even by a one:

6 \* 7 = 42 permutations  
6 + 7 = 13 combinations.

Permutations increase explosively (what’s the maths term for it? Exponential? Won’t risk using it. Do not know much). Combinations increase at a much more manageable pace.

Suppose you are a car maker. And you have 5 different models of cars. And you make them in 5 different colours. Now go about writing classes for each type car you make. Immediately you will realize that you need 5 \* 5 = 25 such classes !!! Yikes !!  
Like this:

Model\_A\_Color\_red, Model\_A\_Color\_blue, Model\_A\_Color\_yellow, Model\_A\_Color\_black, Model\_A\_Color\_pink  
Model\_B\_Color\_red, Model\_B\_Color\_blue, Model\_B\_Color\_yellow, Model\_B\_Color\_black, Model\_B\_Color\_pink  
Model\_C\_Color\_red, Model\_C\_Color\_blue, Model\_C\_Color\_yellow, Model\_C\_Color\_black, Model\_C\_Color\_pink  
Model\_D\_Color\_red, Model\_D\_Color\_blue, Model\_D\_Color\_yellow, Model\_D\_Color\_black, Model\_D\_Color\_pink  
Model\_E\_Color\_red, Model\_E\_Color\_blue, Model\_E\_Color\_yellow, Model\_E\_Color\_black, Model\_E\_Color\_pink

And suppose you introduce one more model of car, Model\_F. Now you have 30 classes !!!  
One more colour into the mix and you have 36 classes to code !!!

Bridge pattern breaks these Permutations into Combinations.

So instead of 5 \* 5 = 25 classes, you have 5 + 5 = 10 classes.  
Add another model and you have 6 + 5 = 11 classes.  
Add another colour and you have 6 + 6 = 12 classes.

This is the gist of Bridge pattern. Take two independently varying items (in our case: car model and car colour) and separate them. Then “COMBINE” them to get what you want.

**HOW**  
We will continue with the same example.  
For the colour we will define an Interface called IColor. It will have a method called Color which will return the color. For the car model, we will define ICarModel. It will have a method WhatIsMyType() which will return the information about the car.

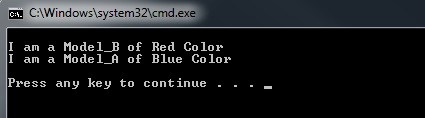
Here is the code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72 | #include<iostream>  #include<string>    using namespace std;    class IColor  {  public:  virtual string Color() = 0;  };    class RedColor : public IColor  {  public:  string Color()  {  return "of Red Color";  }  };    class BlueColor : public IColor  {  public:  string Color()  {  return "of Blue Color";  }  };    class ICarModel  {  public:  virtual string WhatIsMyType() = 0;  };    class Model\_A : public ICarModel  {  IColor\* \_myColor;  public:  Model\_A(IColor \*obj) :\_myColor(obj){}  string WhatIsMyType()  {  return "I am a Model\_A " + \_myColor->Color();  }  };    class Model\_B : public ICarModel  {  IColor\* \_myColor;  public:  Model\_B(IColor \*obj) :\_myColor(obj){}  string WhatIsMyType()  {  return "I am a Model\_B " + \_myColor->Color();;  }  };    int main()  {  IColor\* red = new RedColor();  IColor\* blue = new BlueColor();    ICarModel\* modelA = new Model\_B(red);  ICarModel\* modelB = new Model\_A(blue);    cout << "\n" << modelA->WhatIsMyType();  cout << "\n" << modelB->WhatIsMyType() << "\n\n";    delete red;  delete blue;  return 1;  } |

Here are the important lines:

* **Line 06 :** We declare the interface IColor. **This is one of the things which changes.**
* **Line 09 :** The method which will be returning the color in the implementing classes.
* **Line 12 :** The RedColor class implementing the interface.
* **Line 21 :** The BlueColor class implementing the interface.
* **Line 30 :** The interface for the Car models. **This is the other thing which changes.**
* **Line 33 :** The method which will return the type of the model.
* **Line 36 :** The Model\_A class implementing the interface.
* **Line 38 : A pointer to IColor is present in the class. This will get initialized in the constructor.**
* **Line 40 : The constructor where the initialization takes place. The object of this class will be created using an instance of type IColor.**
* **Line 47 :** The Model\_B class implementing the interface.
* **Line 60 – 64 :** The client declaring and initializing an instances of the classes.
* **Line 66-67 :** The method gets called.

And this is the output:

[](https://i2.wp.com/ikeptwalking.com/wp-content/uploads/2015/09/BridgeDesignPatternDemo.jpg)

I will sign off with providing a [link to the stackoverflow](http://stackoverflow.com/questions/350404/how-do-the-proxy-decorator-adapter-and-bridge-patterns-differ) answer regarding difference between the Proxy, Adapter, Decorator and Bridge pattern.