Bridge pattern

From Wikipedia, the free encyclopedia

The **bridge pattern** is a design pattern used in software engineering which is meant to *"decouple an abstraction from its implementation so that the two can vary independently"*.[1] The *bridge* uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.

When a class varies often, the features of object-oriented programming become very useful because changes to a program's code can be made easily with minimal prior knowledge about the program. The bridge pattern is useful when both the class and what it does vary often. The class itself can be thought of as the *abstraction* and what the class can do as the *implementation*. The bridge pattern can also be thought of as two layers of abstraction.

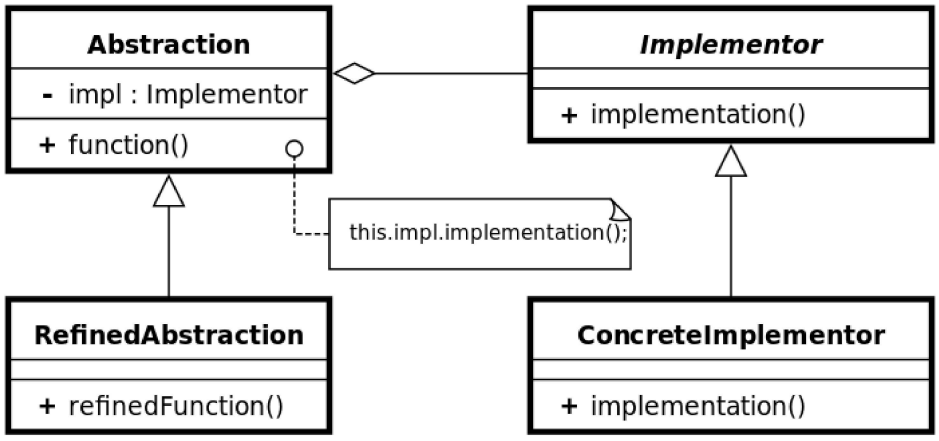
When there is only one fixed implementation, this pattern is known as the Pimpl idiom in the C++ world.

The **bridge pattern** is often confused with the adapter pattern. In fact, the **bridge pattern** is often implemented using the class **adapter pattern**, e.g. in the Java code below.

Variant: The implementation can be decoupled even more by deferring the presence of the implementation to the point where the abstraction is utilized.

Contents

Structure



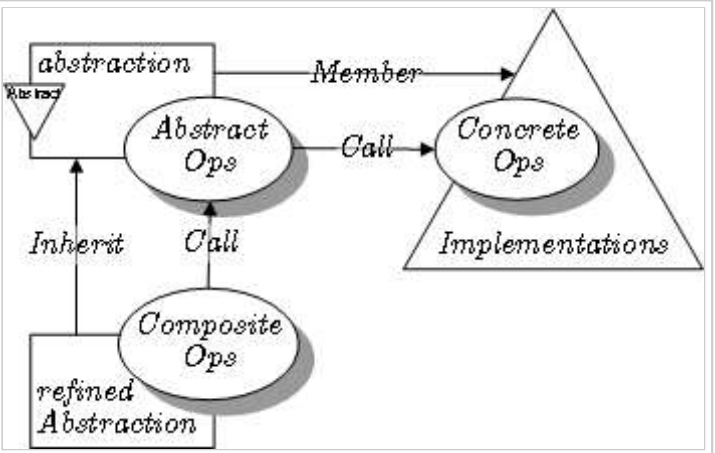
Abstraction (abstract class)

defines the abstract interface
maintains the Implementor reference.

RefinedAbstraction (normal class)
extends the interface defined by Abstraction

Implementor (interface)
defines the interface for implementation classes

ConcreteImplementor (normal class)
implements the Implementor interface



Bridge in LePUS3 (legend (http://lepus.org.uk/ref/legend/legend.xml))

Example

C#

Bridge pattern compose objects in tree structure. It decouples abstraction from implementation. Here abstraction represents the client from which the objects will be called. An example to implement in c# is given below

```
// Helps in providing truly decoupled architecture
public interface IBridge
{
    void Function1();
    void Function2();
}

public class Bridge1 : IBridge
{
    public void Function1()
    {
        throw new NotImplementedException();
    }

    public void Function2()
    {
        throw new NotImplementedException();
    }
}

public class Bridge2 : IBridge
{
    public void Function1()
    {
        throw new NotImplementedException();
    }

    public void Function2()
    {
```

```
        throw new NotImplementedException();
    }
}

public interface IAbstractBridge
{
    void CallMethod1();
    void CallMethod2();
}

public class AbstractBridge : IAbstractBridge
{
    public IBridge bridge;

    public AbstractBridge(IBridge bridge)
    {
        this.bridge = bridge;
    }

    public void CallMethod1()
    {
        this.bridge.Function1();
    }

    public void CallMethod2()
    {
        this.bridge.Function2();
    }
}
```

As you can see the Bridge classes are the Implementation, which uses the same interface oriented architecture to create objects. On the other hand, the abstraction takes an object of the implementation phase and runs its method. Thus makes it completely decoupled with one another.

Java

The following Java (SE 6) program illustrates a 'shape'.

```
/** "Implementor" */
interface DrawingAPI {
    public void drawCircle(double x, double y, double radius);
}

/** "ConcreteImplementor"  1/2 */
class DrawingAPI1 implements DrawingAPI {
    public void drawCircle(double x, double y, double radius) {
        System.out.printf("API1.circle at %f:%f radius %f\n", x, y, radius);
    }
}

/** "ConcreteImplementor" 2/2 */
class DrawingAPI2 implements DrawingAPI {
    public void drawCircle(double x, double y, double radius) {
        System.out.printf("API2.circle at %f:%f radius %f\n", x, y, radius);
    }
}

/** "Abstraction" */
abstract class Shape {
    protected DrawingAPI drawingAPI;

    protected Shape(DrawingAPI drawingAPI){
        this.drawingAPI = drawingAPI;
    }

    public abstract void draw();                                // low-level
    public abstract void resizeByPercentage(double pct);     // high-level
}
```

```
/** "Refined Abstraction" */
class CircleShape extends Shape {
    private double x, y, radius;
    public CircleShape(double x, double y, double radius, DrawingAPI drawingAPI) {
        super(drawingAPI);
        this.x = x;  this.y = y;  this.radius = radius;
    }

    // low-level i.e. Implementation specific
    public void draw() {
        drawingAPI.drawCircle(x, y, radius);
    }
    // high-level i.e. Abstraction specific
    public void resizeByPercentage(double pct) {
        radius *= (1.0 + pct/100.0);
    }
}

/** "Client" */
class BridgePattern {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[] {
            new CircleShape(1, 2, 3, new DrawingAPI1()),
            new CircleShape(5, 7, 11, new DrawingAPI2())
        };

        for (Shape shape : shapes) {
            shape.resizeByPercentage(2.5);
            shape.draw();
        }
    }
}
```

It will output:

```
API1.circle at 1.000000:2.000000 radius 3.075000
API2.circle at 5.000000:7.000000 radius 11.275000
```

PHP

```
interface DrawingAPI {
    function drawCircle($x, $y, $radius);
}

class DrawingAPI1 implements DrawingAPI {
    public function drawCircle($x, $y, $radius) {
        echo "API1.circle at $x:$y radius $radius.\n";
    }
}

class DrawingAPI2 implements DrawingAPI {
    public function drawCircle($x, $y, $radius) {
        echo "API2.circle at $x:$y radius $radius.\n";
    }
}

abstract class Shape {
    protected $drawingAPI;

    public abstract function draw();
    public abstract function resizeByPercentage($pct);

    protected function __construct(DrawingAPI $drawingAPI) {
        $this->drawingAPI = $drawingAPI;
    }
}

class CircleShape extends Shape {
    private $x;
```

```
    private $y;
    private $radius;

    public function __construct($x, $y, $radius, DrawingAPI $drawingAPI) {
        parent::__construct($drawingAPI);
        $this->x = $x;
        $this->y = $y;
        $this->radius = $radius;
    }

    public function draw() {
        $this->drawingAPI->drawCircle($this->x, $this->y, $this->radius);
    }

    public function resizeByPercentage($pct) {
        $this->radius *= $pct;
    }
}

class Tester {
    public static function main()  {
        $shapes = array(
            new CircleShape(1, 3, 7,  new DrawingAPI1()),
            new CircleShape(5, 7, 11, new DrawingAPI2()),
        );

        foreach ($shapes as $shape) {
            $shape->resizeByPercentage(2.5);
            $shape->draw();
        }
    }
}

Tester::main();
```

Output:

```
API1.circle at 1:3 radius 17.5
API2.circle at 5:7 radius 27.5
```

Scala

```
trait DrawingAPI {
  def drawCircle(x: Double, y: Double, radius: Double)
}

class DrawingAPI1 extends DrawingAPI {
  def drawCircle(x: Double, y: Double, radius: Double) = println(s"API #1 $x $y $radius")
}

class DrawingAPI2 extends DrawingAPI {
  def drawCircle(x: Double, y: Double, radius: Double) = println(s"API #2 $x $y $radius")
}

abstract class Shape(drawingAPI: DrawingAPI) {
  def draw()
  def resizePercentage(pct: Double)
}

class CircleShape(x: Double, y: Double, var radius: Double, drawingAPI: DrawingAPI)
    extends Shape(drawingAPI: DrawingAPI) {

  def draw() = drawingAPI.drawCircle(x, y, radius)

  def resizePercentage(pct: Double) { radius *= pct }
}

object BridgePattern {
  def main(args: Array[String]) {
```

```
    Seq (
    new CircleShape(1, 3, 5, new DrawingAPI1),
    new CircleShape(4, 5, 6, new DrawingAPI2)
    ) foreach { x =>
        x.resizePercentage(3)
        x.draw()
      }
  }
}
```

See also

- Template method pattern
- Strategy pattern
- Adapter pattern

References

1. Gamma, E, Helm, R, Johnson, R, Vlissides, J: *Design Patterns*, page 151. Addison-Wesley, 1995

External links

- Bridge in UML and in LePUS3 (http://www.lepus.org.uk/ref/companion/Bridge.xml) (a formal modelling language)
- "C# Design Patterns: The Bridge Pattern". *Sample Chapter*. From: James W. Cooper. *C# Design Patterns: A Tutorial*. Addison-Wesley. ISBN 0-201-84453-2.

The Wikibook *Computer Science/Design Patterns* has a page on the topic of: ***Bridge pattern implementations in various languages***

Retrieved from "https://en.wikipedia.org/w/index.php?title=Bridge_pattern&oldid=715450549"

Categories: Software design patterns

- This page was last modified on 15 April 2016, at 21:42.
- Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.