When do you use the Bridge Pattern? How is it different from Adapter pattern?

140

Has anyone ever used the Bridge Pattern in a real world application? If so, how did you use it? Is it me, or is it just the Adaptor Pattern with a little dependency injection thrown into the mix? Does it really deserve its own pattern?

90

design-patterns adapter bridge

edited Jun 18 '18 at 20:21

lmichelbacher
2,878 1 19 34

asked Nov 26 '08 at 4:12

Charles Graham
12.9k 13 38 55

11 Answers

75

A classic example of the Bridge pattern is used in the definition of shapes in an UI environment (see the Bridge

pattern Wikipedia entry). The Bridge pattern is a composite of the Template and Strategy patterns.

It is a common view some aspects of the Adapter pattern in the Bridge pattern. However, to quote from this article:

> At first sight, the Bridge pattern looks a lot like the Adapter pattern in that a class is used to convert one kind of interface to another. However, the intent of the Adapter pattern is to make one or more classes' interfaces look the same as that of a particular class. The Bridge pattern is designed to separate a class's interface from its implementation so you can vary or replace the implementation without changing the client code.

answered Nov 26 '08 at 4:40

shek
11.5k 2 22 23

230

There's a combination of Federico's and John's answers.

When:

```
                ----Shape---
               /            \
         Rectangle            Circle
        /         \          /      \
BlueRectangle  RedRectangle BlueCircle RedCircle
```

Refactor to:

```
          ----Shape---                        Color
         /            \                      /   \
Rectangle(Color)   Circle(Color)          Blue   Red
```

edited May 23 '17 at 12:03

Community ♦

1 1

answered May 21 '12 at 16:31

Anton Shchastnyi

3,231 3 16 21

5 Why would you do inheritance for colors? – vainolo Feb 3 '13 at 9:32

10 @vainolo because Color is an interface and Blue, Red are concrete colors – Weltschmerz Feb 28 '14 at 21:46

3 This is just a refactoring. Bridge pattern intent: "Decouple an abstraction from its implementation so that the two can vary independently." Where is the abstraction and where is the implementation here? – clapas Sep 6 '17 at 15:58

1 Is not Rectangle(Color) a more abstract one that the BlueRectangle thing? – Anton Shchastnyi Sep 7 '17 at 10:27

2 @clapas , Abstraction is "Shape.color" of property ,therefore class Red and class Blue is implementation,and Color interface is bridge. – reco Dec 29 '17 at 9:13

218

The Bridge pattern is an application of the old advice, "prefer composition over inheritance". It becomes handy when you must subclass different times in ways that are orthogonal with one another. Say you must implement a hierarchy of colored shapes. You wouldn't subclass Shape with Rectangle and Circle and then subclass Rectangle with RedRectangle, BlueRectangle and GreenRectangle and the same for Circle, would you? You would prefer to

say that each Shape *has* a Color and to implement a hierarchy of colors, and that is the Bridge Pattern. Well, I wouldn't implement a "hierarchy of colors", but you get the idea...

answered Nov 26 '08 at 4:44

Federico A. Ramponi
34.8k 23 95 128

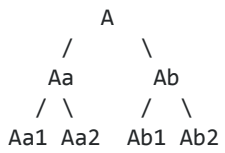
1 See also Anton Shchastnyi diagram below for a graphical illustration of this explanation. – NomadeNumerique Feb 3 '14 at 17:51

Best explanation – VorobeY1326 Feb 15 '17 at 14:54

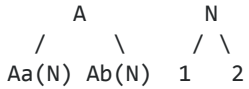
1 I do not think a color is a good example for an implementation hierarchy, it is rather confusing. There is a good example of the Bridge pattern in "Design patterns" by the GoF, where implementation is platform dependent: IBM's PM, UNIX's X etc. – clapas Sep 6 '17 at 16:12

200

When:



Refactor to:



edited Jan 30 '13 at 7:31

Alex Lockwood
75.2k 34 183 233

answered Feb 23 '12 at 2:27

John Sonmez
5,822 5 31 54

2 I think it's very pragmatic approach to patterns: 1) describe suboptimal straight-forward design 2) refactor design/code to better factored one – Alexey Jul 9 '12 at 21:44

1 Use maths concept to explain the Bridge design pattern. Very interested. – Jian Huang Feb 17 '15 at 21:34

1 This is just a refactoring. Bridge pattern intent: "Decouple an abstraction from its implementation so that the two can vary independently." Where is the abstraction and where is the implementation here? – clapas Sep 6 '17 at 15:58

John puts it nicely in a blog post. Found it to be a good read for high level overview. – Vaibhav Bhalla Jan 17 at 9:17

29

Adapter and Bridge are certainly related, and the distinction is subtle. It's likely that some people who think they are using one of these patterns are actually using the other pattern.

The explanation I've seen is that Adapter is used when you're trying to unify the interfaces of some incompatible classes that *already exist*. The Adapter functions as a kind of translator to implementations that could be considered *legacy*.

Whereas the Bridge pattern is used for code that is more likely to be greenfield. You're designing the Bridge to provide an abstract interface for an implementation that needs to vary, but you also define the interface of those implementation classes.

Device drivers is an often-cited example of Bridge, but I'd say it's a Bridge if you're defining the interface spec for

device vendors, but it's an Adapter if you're taking existing device drivers and making a wrapper-class to provide a unified interface.

So code-wise, the two patterns are very similar. Business-wise, they're different.

See also http://c2.com/cgi/wiki?BridgePattern

edited Nov 26 '08 at 18:53

answered Nov 26 '08 at 4:44

Bill Karwin
383k 64 520 678

Hey Bill. I don't understand why we necessarily use Bridge pattern in device drivers. I mean we can easily delegate implementation (of read, write, seek etc.) to the correct class via polymorphism right? Or with a Visitor perhaps? Why it's have to be Bridge? Thanks in advance. – stdout Nov 30 '16 at 23:31

1 @zgulser, yes, you do use polymorphism. The Bridge pattern describes one kind of usage of subclasses to decouple implementation from abstraction. – Bill Karwin Nov 30 '16 at 23:47

You meant decoupling Shape implementation (ie. Rectangle) from let's day Color abstraction right? And I believe you're saying there're variety of ways to do it and Bridge is just one of them. – stdout Dec 1 '16 at 7:42

Yes, subclassing has other uses. This particular way of using subclasses is what makes it the Bridge pattern. – Bill Karwin Dec 1 '16 at 8:21

And the decoupling I mean is from the abstract Shape interface to a concrete Rectangle implementation. So you can write code that needs an object of the "Shape" type, even though the concrete object is really some subclass of Shape. – Bill Karwin Dec 1 '16 at 8:23

26

In my experience, Bridge is a quite often recurring pattern, because it's the solution whenever **there are two orthogonal dimensions in the domain**. E.g. shapes and drawing methods, behaviours and platforms, file formats and serializers and so forth.

And an advice: always think of design patterns *from the conceptual perspective*, not from the implementation perspective. From the right point of view, Bridge cannot be confused with Adapter, because they solve a different problem, and composition is superior to inheritance not because of the sake of itself, but because it allows to handle orthogonal concerns separately.

edited Jun 5 '15 at 7:54

answered May 25 '10 at 9:20

thSoft

15.6k 5 75 93

18

The intent of **Bridge** and **Adapter** is different and we need both patterns separately.

Bridge pattern:

1. It is a structural pattern
2. Abstraction and implementation are not bound at compile time
3. Abstraction and implementation - both can vary without impact in client
4. Uses composition over inheritance.

Use the Bridge pattern when:

1. You want run-time binding of the implementation,
2. You have a proliferation of classes resulting from a coupled interface and numerous implementations,
3. You want to share an implementation among multiple objects,
4. You need to map orthogonal class hierarchies.

@ John Sonmez answer clearly shows effectiveness of bridge pattern in reducing class hierarchy.

You can refer to below documentation link to get better insight into bridge pattern with code example

Adapter pattern:

1. It *allows two unrelated interfaces to work together* through the different objects, possibly playing same role.
2. It modifies original interface.

Key differences:

1. *Adapter* makes things work after they're designed; *Bridge* makes them work before they are.
2. *Bridge* is designed up-front to let the *abstraction and the implementation vary independently*. *Adapter* is retrofitted to make unrelated classes work together.
3. The intent : *Adapter* allows two unrelated interfaces to work together. *Bridge* allows Abstraction and implementation to vary independently.

Related SE question with UML diagram and working code:

Difference between Bridge pattern and Adapter pattern

Useful articles:

sourcemaking bridge pattern article

sourcemaking adapter pattern article

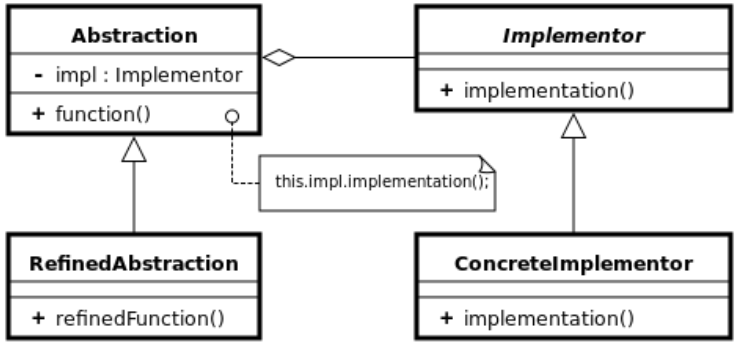
journaldev bridge pattern article

EDIT:

Bridge Pattern real world example (As per meta.stackoverflow.com suggestion, incorporated documentation site example in this post since documentation is going to sun-set)

Bridge pattern decouples abstraction from implementation so that both can vary independently. It has been achieved with composition rather than inheritance.

Bridge pattern UML from Wikipedia:



You have four components in this pattern.

`Abstraction` : It defines an interface

`RefinedAbstraction` : It implements abstraction:

`Implementor` : It defines an interface for implementation

`ConcreteImplementor` : It implements Implementor interface.

The crux of Bridge pattern : Two orthogonal class hierarchies using composition (and no inheritance). The Abstraction hierarchy and Implementation hierarchy can vary independently. Implementation never refers Abstraction. Abstraction contains Implementation interface as a member (through composition). This composition reduces one more level of inheritance hierarchy.

Real word Use case:

Enable different vehicles to have both versions of manual and auto gear system.

Example code:

```
/* Implementor interface*/
interface Gear{
    void handleGear();
}

/* Concrete Implementor - 1 */
class ManualGear implements Gear{
    public void handleGear(){
        System.out.println("Manual gear");
    }
}
/* Concrete Implementor - 2 */
class AutoGear implements Gear{
    public void handleGear(){
        System.out.println("Auto gear");
    }
}
/* Abstraction (abstract class) */
abstract class Vehicle {
    Gear gear;
    public Vehicle(Gear gear){
        this.gear = gear;
    }
    abstract void addGear();
}
/* RefinedAbstraction - 1*/
class Car extends Vehicle{
    public Car(Gear gear){
        super(gear);
        // initialize various other Car components to make
```

```
    }
    public void addGear(){
        System.out.print("Car handles ");
        gear.handleGear();
    }
}
/* RefinedAbstraction - 2 */
class Truck extends Vehicle{
    public Truck(Gear gear){
        super(gear);
        // initialize various other Truck components to ma
    }
    public void addGear(){
        System.out.print("Truck handles " );
        gear.handleGear();
    }
}
/* Client program */
public class BridgeDemo {
    public static void main(String args[]){
        Gear gear = new ManualGear();
        Vehicle vehicle = new Car(gear);
        vehicle.addGear();

        gear = new AutoGear();
        vehicle = new Car(gear);
        vehicle.addGear();

        gear = new ManualGear();
        vehicle = new Truck(gear);
        vehicle.addGear();

        gear = new AutoGear();
        vehicle = new Truck(gear);
        vehicle.addGear();
    }
}
```

output:

```
Car handles Manual gear
Car handles Auto gear
Truck handles Manual gear
Truck handles Auto gear
```

Explanation:

1. `Vehicle` is an abstraction.
2. `Car` and `Truck` are two concrete implementations of `Vehicle` .
3. `Vehicle` defines an abstract method : `addGear()` .
4. `Gear` is implementor interface
5. `ManualGear` and `AutoGear` are two implementations of `Gear`
6. `Vehicle` contains `implementor` interface rather than implementing the interface. `Compositon` of implementor interface is crux of this pattern : *It allows abstraction and implementation to vary independently.*
7. `Car` and `Truck` define implementation (redefined abstraction) for abstraction : `addGear()` : It contains `Gear` - Either `Manual` or `Auto`

Use case(s) for Bridge pattern:

1. **Abstraction** and **Implementation** can change independent each other and they are not bound at compile time
2. Map orthogonal hierarchies - One for *Abstraction* and one for *Implementation*.

edited Sep 20 '17 at 2:26

answered May 29 '16 at 20:42

Ravindra babu

30.5k 6 165 140

"Adapter makes things work after they're designed; Bridge makes them work before they are." You may want to look into the Pluggable Adapter. It's a variation of Adapter described by the GoF in the "Adapter" section of their Design Patterns book. The purpose is to create an interface for classes that do not

yet exist. A pluggable adapter is not a Bridge, so I don't feel like the first point is valid. – c1moore Nov 23 '16 at 4:18

Although manual and auto gear might require different implementation for truck and car – andigor Sep 30 '18 at 15:59

9

I have used the bridge pattern at work. I program in C++, where it is often called the PIMPL idiom (pointer to implementation). It looks like this:

```
class A
{
public:
  void foo()
  {
    pImpl->foo();
  }
private:
  Aimpl *pImpl;
};

class Aimpl
{
public:
  void foo();
  void bar();
};
```

In this example `class A` contains the interface, and `class Aimpl` contains the implementation.

One use for this pattern is to expose only some of the public members of the implementation class, but not others. In the example only `Aimpl::foo()` can be called through the public interface of `A`, but not `Aimpl::bar()`

Another advantage is that you can define `Aimpl` in a separate header file that need not be included by the users of `A`. All you have to do is use a forward declaration of `Aimpl` before `A` is defined, and move the definitions of all

the member functions referencing `pImpl` into the .cpp file. This gives you the ability to keep the `Aimpl` header private, and reduce the compile time.

edited Sep 18 '12 at 1:52

Dave M
1,216 1 16 26

answered Nov 26 '08 at 5:01

Dima
33.7k 12 60 108

2 If you use this pattern then the AImpl doesn't even need a header. I just put it inline in the implementation file for the A class – 1800 INFORMATION Nov 26 '08 at 6:05

Your implementor is private. I have a new question in regards to this, see stackoverflow.com/questions/17680762/… – Roland Jul 16 '13 at 15:28

6

To put shape example in code:

```
#include<iostream>
#include<string>
#include<cstdlib>

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 IShape
{
public:
virtual string Draw() = 0;
};

class Circle: public IShape
{
        IColor* impl;
    public:
        Circle(IColor *obj):impl(obj){}
        string Draw()
        {
            return "Drawn a Circle "+ impl->Color();
        }
};

class Square: public IShape
{
        IColor* impl;
    public:
        Square(IColor *obj):impl(obj){}
        string Draw()
        {
        return "Drawn a Square "+ impl->Color();;
        }
};

int main()
{
IColor* red = new RedColor();
IColor* blue = new BlueColor();

IShape* sq = new Square(red);
IShape* cr = new Circle(blue);
```

```
cout<<"\n"<<sq->Draw();
cout<<"\n"<<cr->Draw();

delete red;
delete blue;
return 1;
}
```

The output is:

```
Drawn a Square of Red Color
Drawn a Circle of Blue Color
```

Note the ease with which new colors and shapes can be added to the system without leading to an explosion of subclasses due to permutations.

answered Sep 14 '13 at 22:52

NotAgain
1,159 1 16 33

0

for me i think of it as a mechanism where you can swap interfaces. In the real world you might have a class that can use more then one interface, Bridge lets you swap.

answered Dec 29 '12 at 16:32

j2emanue
23.7k 26 146 261

-3

```
Bridge design pattern we can easily understand helping of s

Dao layer -> create common interface for dao layer ->
public interface Dao<T>{
void save(T t);
}
public class AccountDao<Account> implement Dao<Account>{
```

```
public void save(Account){
}
}
public LoginDao<Login> implement Dao<Login>{
public void save(Login){
}
}
Service Layer ->
1) interface
public interface BasicService<T>{
    void save(T t);
}
concrete  implementation of service -
Account service -
public class AccountService<Account> implement BasicServic
 private Dao<Account> accountDao;
 public AccountService(AccountDao dao){
   this.accountDao=dao;
   }
public void save(Account){
   accountDao.save(Account);
 }
}
login service-
public class LoginService<Login> implement BasicService<Lo
 private Dao<Login> loginDao;
 public AccountService(LoginDao dao){
   this.loginDao=dao;
   }
public void save(Login){
   loginDao.save(login);
 }
}

public class BridgePattenDemo{
public static void main(String[] str){
BasicService<Account> aService=new AccountService(new Acco
Account ac=new Account();
aService.save(ac);
}
}
}
```

answered Dec 25 '17 at 5:54

sohan kumawat

34 3

4 I downvoted because I feel this is a convoluted, poorly formatted answer. – Zimano Mar 29 '18 at 12:27

1 Totally agree, how can you post answers on this site without a minimal attention to code indentation and clearness – Massimiliano Kraus Jul 10 '18 at 11:24