HowToDoInJava

Bridge Design Pattern

Last Updated: September 1, 2021 By: Lokesh Gupta Structural Patterns Design Patterns

Bridge design pattern is used to decouple a class into two parts – abstraction and it's implementation – so that both can evolve in future without affecting each other. It increases the loose coupling between class abstraction and it's implementation.

Table of Contents

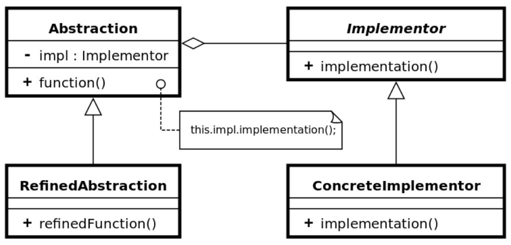
Design participants of bridge design pattern
When we need bridge design pattern
Sample problem statement
Solution using bridge design pattern
Final notes

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

Bridge is a synonym for the "**handle/body**" idiom. This is a design mechanism that encapsulates an implementation class inside of an interface class. The former is the body, and the latter is the handle. The handle is viewed by the user as the actual class, but the work is done in the body.

You get this decoupling by adding one more redirection between methods calls from abstraction to implementation.

Design participants of bridge design pattern



Bridge pattern participants

Following participants constitute the bridge design pattern.

1. **Abstraction (abstract class)**

It defined the abstract interface i.e. behavior part. It also maintains the Implementer reference.

2. **RefinedAbstraction (normal class)**

   It extends the interface defined by Abstraction.

3. **Implementer (interface)**

   It defines the interface for implementation classes. This interface does not need to correspond directly to abstraction interface and can be very different. Abstraction imp provides an implementation in terms of operations provided by Implementer interface.

4. **ConcreteImplementor (normal class)**

   It implements the Implementer interface.

When we need bridge design pattern

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.

For example, let’s say you are creating various GUI shapes with different colors. One solution could be:

Without bridge pattern

But above solution has a problem. If you want to change `Rectange` class, then you may end up changing `BlueRectangle` and `RedRectangle` as well – and even if change is color specific then you may need to change `Circle` classes as well.

You can solve above problem by decoupling the `Shape` and `Color` interfaces in below manner.

With bridge pattern

Now when you change any Shape, color would be unchanged. Similarily, vice-versa.

Sample problem statement

> Bridge design pattern is most applicable in applications where you need to provide **platform independence**.

Let's say, we are designing **an application which can be download and store files on any operating system**. I want to design the system in such a way, I should be able to add more platform support in future with minimum change. Additionally, If I want to add more support in downloader class (e.g. delete the download in windows only), then It should not affect the client code as well as linux downloader.

Solution using bridge design pattern

As this problem is classical platform independence related problem, I will use bridge pattern to solve this. I will break the downloader component into abstraction and implementer parts.

Here I am creating two interfaces, `FileDownloaderAbstraction` represents the abstraction with which client will interact; and `FileDownloadImplementor` which represents the implementation. In this way, both hierarchies can evolve separately without affecting each other.

FileDownloaderAbstraction.java

```java
public interface FileDownloaderAbstraction
{
    public Object download(String path);

    public boolean store(Object object);
}
```

FileDownloaderAbstractionImpl.java

```java
public class FileDownloaderAbstractionImpl implements FileDownloaderAbstraction {

    private FileDownloadImplementor provider = null;

    public FileDownloaderAbstractionImpl(FileDownloadImplementor provider) {
        super();
        this.provider = provider;
    }

    @Override
    public Object download(String path)
    {
        return provider.downloadFile(path);
    }

    @Override
    public boolean store(Object object)
    {
        return provider.storeFile(object);
    }
}
```

FileDownloadImplementor.java

```java
public interface FileDownloadImplementor
{
    public Object downloadFile(String path);

    public boolean storeFile(Object object);
}
```

LinuxFileDownloadImplementor.java

```java
public class LinuxFileDownloadImplementor implements FileDownloadImplementor
{
    @Override
    public Object downloadFile(String path) {
        return new Object();
    }

    @Override
    public boolean storeFile(Object object) {
        System.out.println("File downloaded successfully in LINUX !!");
        return true;
    }
}
```

WindowsFileDownloadImplementor.java

```java
public class WindowsFileDownloadImplementor implements FileDownloadImplementor
{
    @Override
    public Object downloadFile(String path) {
        return new Object();
    }

    @Override
    public boolean storeFile(Object object) {
        System.out.println("File downloaded successfully in WINDOWS !!");
        return true;
    }
}
```

Client.java

```java
public class Client
{
    public static void main(String[] args)
    {
        String os = "linux";
        FileDownloaderAbstraction downloader = null;

        switch (os)
        {
            case "windows":
                downloader = new FileDownloaderAbstractionImpl( new WindowsFileDownloadImplementor() );
                break;

            case "linux":
                downloader = new FileDownloaderAbstractionImpl( new LinuxFileDownloadImplementor() );
                break;

            default:
                System.out.println("OS not supported !!");
        }

        Object fileContent = downloader.download("some path");
        downloader.store(fileContent);
    }
}
```

```
Output:

File downloaded successfully in LINUX !!
```

Change in abstraction does not affect implementation

Now let's say you want to add one more capability (i.e. delete) at abstraction layer. It must not force a change in existing implementers and client as well.

FileDownloaderAbstraction.java

```
public interface FileDownloaderAbstraction
{
    public Object download(String path);

    public boolean store(Object object);

    public boolean delete(String object);
}
```

FileDownloaderAbstractionImpl.java

```
public class FileDownloaderAbstractionImpl implements FileDownloaderAbstraction {

    private FileDownloadImplementor provider = null;

    public FileDownloaderAbstractionImpl(FileDownloadImplementor provider) {
        super();
        this.provider = provider;
    }

    @Override
    public Object download(String path)
    {
        return provider.downloadFile(path);
    }

    @Override
    public boolean store(Object object)
    {
        return provider.storeFile(object);
    }

    @Override
    public boolean delete(String object) {
        return false;
    }
}
```

Above change does not force you to make any change in implemeters classes/interface.

Change in implementation does not affect abstraction

Let's say you want to add delete feature at implementation layer for all downloaders (an internal feature) which client should not know about.

FileDownloadImplementor.java

```
public interface FileDownloadImplementor
{
    public Object downloadFile(String path);

    public boolean storeFile(Object object);

    public boolean delete(String object);
}
```

LinuxFileDownloadImplementor.java

```
public class LinuxFileDownloadImplementor implements FileDownloadImplementor
{
    @Override
    public Object downloadFile(String path) {
        return new Object();
    }

    @Override
    public boolean storeFile(Object object) {
        System.out.println("File downloaded successfully in LINUX !!");
        return true;
    }

    @Override
    public boolean delete(String object) {
        return false;
    }
}
```

WindowsFileDownloadImplementor.java

```
public class WindowsFileDownloadImplementor implements FileDownloadImplementor
{
    @Override
    public Object downloadFile(String path) {
        return new Object();
    }

    @Override
    public boolean storeFile(Object object) {
        System.out.println("File downloaded successfully in LINUX !!");
        return true;
    }

    @Override
    public boolean delete(String object) {
        return false;
    }
}
```

Above change does not affect the abstraction layer, so client will not be impacted at all.

Final notes

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.

Happy Learning !!

Was this post helpful?

Let us know if you liked the post. That's the only way we can improve.

Yes

No

Recommended Reading:

1. Decorator Design Pattern in Java
2. Adapter Design Pattern in Java
3. Composite Design Pattern
4. Facade Design Pattern
5. Flyweight Design Pattern
6. Proxy Design Pattern
7. Prototype design pattern in Java
8. Strategy Design Pattern
9. Memento Design Pattern
0. Observer Design Pattern

4 thoughts on “Bridge Design Pattern”

Yovel

March 6, 2020 at 5:11 pm

You said that “Decouple an abstraction from its implementation so that the two can vary independently.”
However, in your example, both implementation and abstraction have the same methods in which situations do you need to add more functionality to an interface?
Because I'm still confused about the pattern

Reply

kiran

March 6, 2020 at 3:43 pm

After reading ur article i understood properly bridge pattern. I am using it everday for controller,service,repository pattern each of them evolve separately. One of the best example explained clearly.

Reply

ErwanLeroux

October 18, 2015 at 1:39 pm

In FileDownloaderAbstractionImpl.java

Isn't it better to replace
'private FileDownloadImplementor provider = null;'
by
'private final FileDownloadImplementor provider;'
?

Reply

Lokesh Gupta

October 19, 2015 at 3:10 am

You are right. We can make it final as well.

Reply

Leave a Comment

Name *

Email *

Website

☐ Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search …

Teesar British Boonie Hat with Neck Flap R…

Mil-Te
Shirt