Design Pattern Report

Bridge Pattern: (Structural Pattern)

**Motivation:**

Sometimes an abstraction can have different implementations; this problem is commonly solved by using inheritance. However, Inheritance binds an implementation to the abstraction and thus it would be difficult to modify, extend, and reuse abstraction and implementations independently.

**Intent:**

The intent of this pattern is to decouple abstraction from implementation so that the two can vary independently.

**Diagram:**

**Description:**

An Abstraction can be implemented by an abstraction implementation, and this implementation does not depend on any concrete implementers of the Implementor interface. Extending the abstraction does not affect the Implementor. Also extending the Implementor has no effect on the Abstraction.

**Advantages / Disadvantages:**

Adv:- Decoupling abstraction from implementation, Interface and implementation can be varied independently   
- Reduction in the number of sub classes. For example, Image Viewer supports 6 image formats in 3 OS, using inheritance will result in 18 classes when using bridge result in 9 classes.

- Cleaner code and Reduction in executable size

- Improved Extensibility

- Loosely coupled client code, abstraction separate client code from the implementation so the implementation can be changed with no affect on the client code.

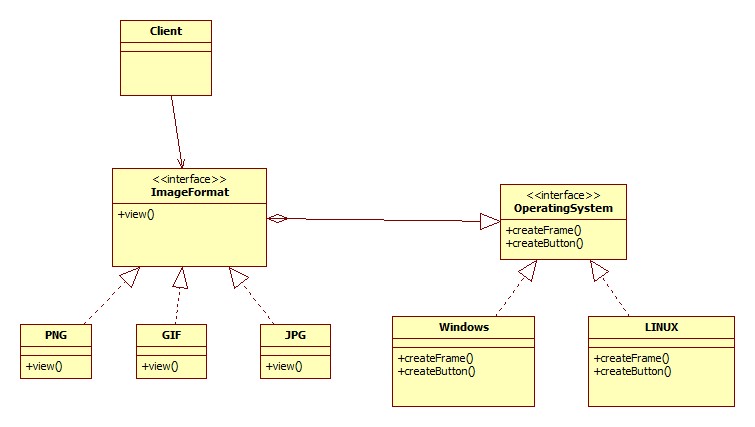
1. Decoupling abstraction from implementation - Inheritance tightly couples an abstraction with an implementation at compile time. **Bridge** **pattern** can be used to avoid the binding between abstraction and implementation and to select the implementation at run time.
2. Reduction in the number of sub classes - Sometimes, using pure inheritance will increase the number of sub classes. Let us assume that the full-blown version of our Image Viewer supports 6 image formats in 3 different operating systems. Pure inheritance would have resulted in 18 sub classes whereas applying **Bridge** **Pattern**reduces the sub class requirement only to 9.
3. Cleaner code and Reduction in executable size - In the above example, operating system specific code is encapsulated in CImageImp sub classes. This results in a cleaner code without much preprocessor statements like #ifdefs, #ifndefs. Also, it is easy to conditionally compile CImageImp sub classes for a given operating system to reduce the size of the executable.
4. Interface and implementation can be varied independently - Maintaining two different class hierarchies for interface and implementation entitles to vary one independent of the other.
5. Improved Extensibility - Abstraction and implementation can be extended independently. As mentioned earlier, the above example can easily be extended to view other image formats on Windows or view BMP images on other operating systems.
6. Loosely coupled client code - Abstraction separates the client code from the implementation. So, the implementation can be changed without affecting the client code and the client code need not be compiled when the implementation changes. (NOTE : In the above mentioned example, for the sake of simplicity, the application configures the CImage object with the right CImageImp object. However, alternate methods like Abstract Factory can be adopted to choose the CImageImp object.)

Disadv:- Performance:

Each bridge use add another function call which can negatively affect the performance.

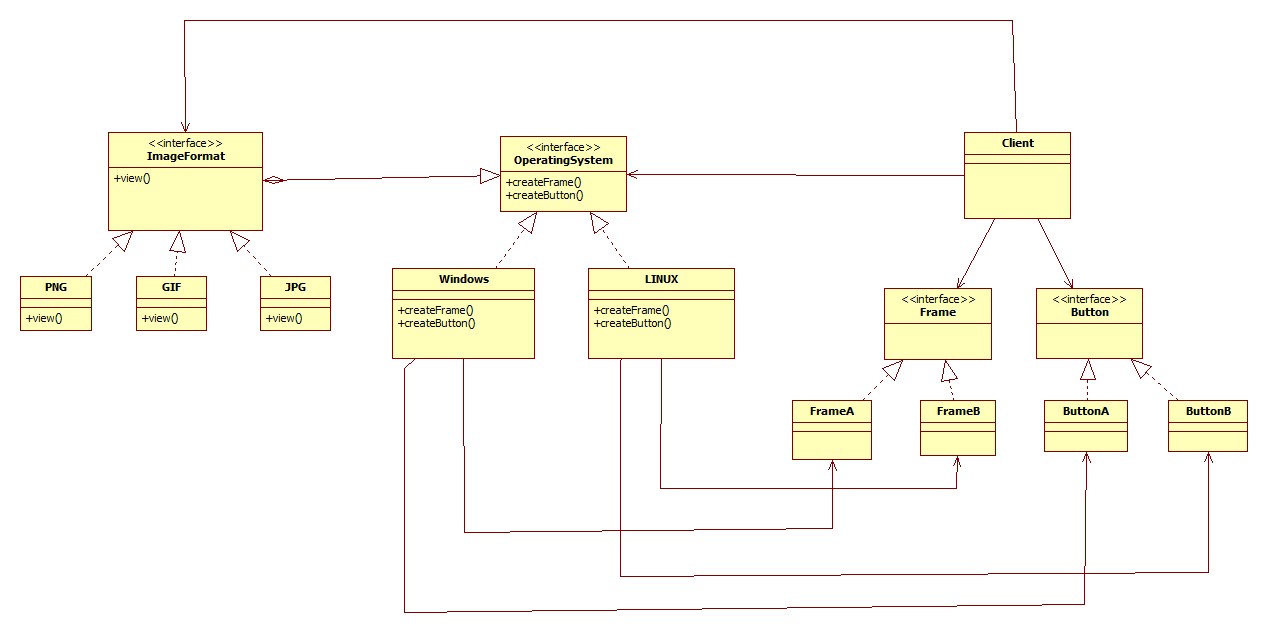
-complexity: harder to debug if a problem arise.

**Example:**

****

**Related Pattern:**

**Abstract factory:** An Abstract Factory pattern can be used create and configure a particular Bridge

****

**Alternate Pattern /why it is a better choice:**

Adapter and Bridge has some common points, both make use of indirection through another object to allow high flexibility and they use an interface to sends request to the object.

Adapter resolves incompatibility between interfaces without having to reimplement any which can cause replicating code. Whereas, Bridge create a stable interface to client despite the variation of class that implement it and it accommodates new implementations as the system evolves. In a software life-cycle, engineers use Bridge when they can foreseen that an abstraction must have many implementations and both evolves independently while Adapter is use when unexpected classes need to work together even if they are incompatible. In short, Bridge make them work when the system is designed and adapter resolve the problem when implementing the system.

**Alternate methods for achieving the same goal as the pattern and adv/disadv of using the alternate methods:**