# Why do Proxy, Decorator, Adapter, Bridge, and Facade design patterns look very similar? What are the differences?

There are often patterns that look very similar, but differ in their **intent**. Most patterns use **polymorphism** with interface inheritance. Strategy and state design patterns are very similar as well. Proxy, Decorator, Adapter, and Bridge are all variations on "**wrapping**" a class. Facade design pattern is a **container** for the classes in another sub system.

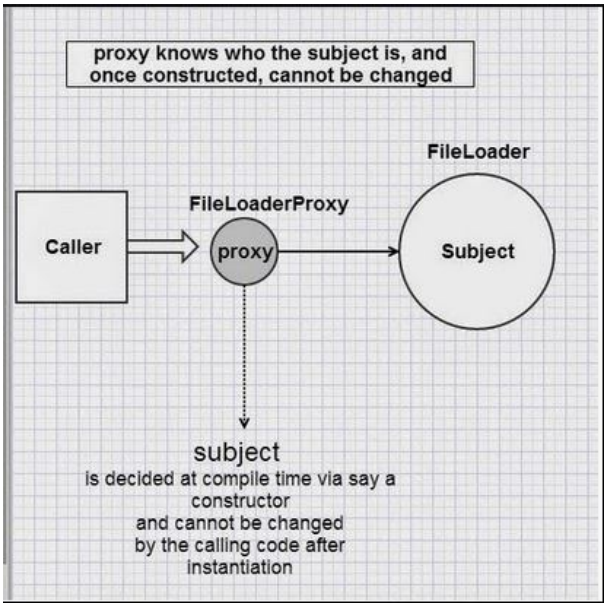
One of the reasons to use a design pattern is to talk the **common vocabulary** with the designers and the developers. The intent and naming conventions (e.g. ***FileLoaderProxy***, ***FileLoaderBridege***, etc) of these design patterns form a common vocabulary.

**Asking a few questions helps.**

- How about a facade to **simplify the subsystems** by minimizing the number of fine grained invocations from the client?
- How about a proxy to provide **access control**?
- How about adding a decorator to **enhance** current behavior? How about a decorator to fix explosive inheritance through composition at run time?
- Do we need an adapter to **talk to third-party API**?
- Do we have different **permutations**? what would our hierarchy look like? How about a bridge design pattern to re-factor this exponentially explosive inheritance
- Do we want to bind this association at compile time or run time?

**Proxy**

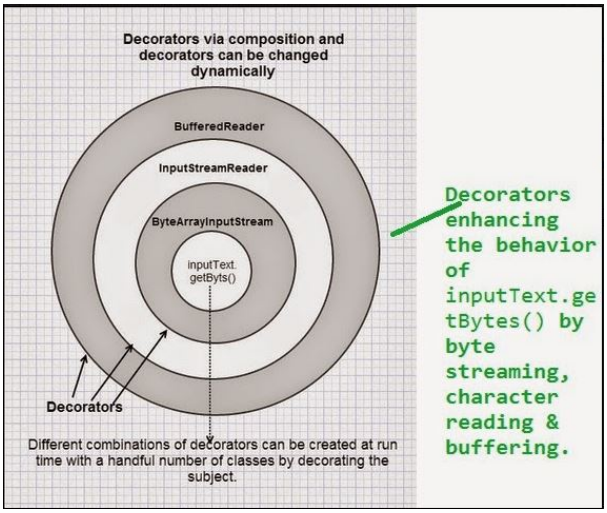
is good to act as a surrogate to provide performance optimization, synchronization, remote access, house keeping, etc. The binding of the actual **subject** to the proxy happens at **compile-time**.




Java proxy design pattern

## Decorator

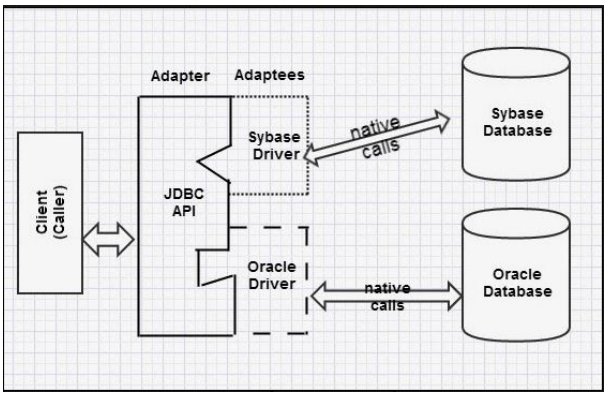
is  good to avoid out of control type hierarchies. A decorator is also known as a “smart proxy” because it enhances the behavior of a subject at **run time**. In other words binding is dynamic. The Java I/O classes are good example of a decorator design pattern. At run time different permutations can can be carried out via composition.




Java decorator design pattern

## Adapter

Adapts at **run time** like the decorator design pattern. Adapter design pattern is one of the structural design patterns and its intent is to get two unrelated interfaces work together. Think of using a laptop in UK that was bought in Japan as the sockets are different, and you need an adapter. So, the adapter's intent is to adapt between the Japanese laptop plug with UKs wall socket. The key point is that parties are different. Japanese laptop used in **third-party or external** (i.e. UK) wall socket.




Java adapter design pattern

Adapter is used when you have an abstract interface, for example a JDBC API and you want to map that interface to another object which has similar functional role, but a different interface, for example different JDBC drivers for different databases like Oracle, Sybase, DB2, SQL server, MySQL, etc. The JEE have multiple adaptors for JMS, JNDI, JDBC, JCA, etc. The drivers and **implementations are generally provided by the third party vendors.** For example, JMS implementations provided by third-party vendors and open source providers web Methods, IBM MQ Series, ActiveMQ, etc.

You can accomplish this using either inheritance or composition.

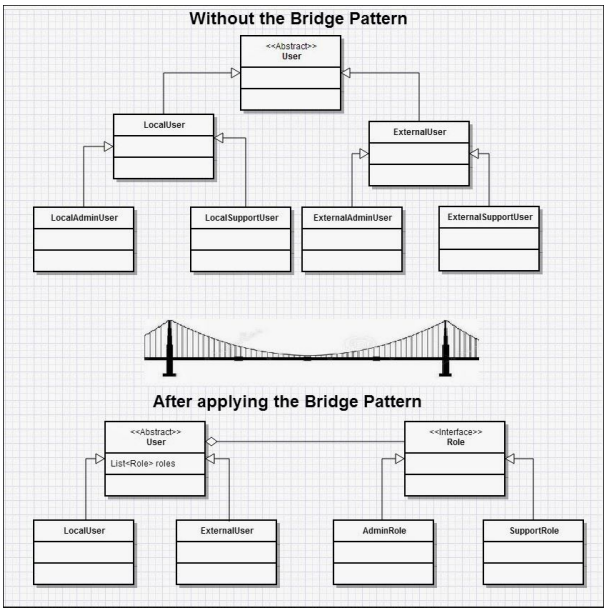
- Class Adapter – This form uses Java inheritance and extends the source interface.
- Object Adapter – This form uses Java Composition and adapter contains the source object.

## Bridge

is very similar to Adapter, but you call it a Bridge when you define both the abstract interface and the underlying implementation (**no external or third-party vendors**). Adapter makes things work **after they're designed**, but bridge makes them w**ork before they are**. Abstraction and implementation should be bound at **compile time**. You need to swap for different

implementations. For example, you are designing a graphics application that needs to swap between graphics driver and printer driver for the same draw( ) .

As shown below, you can have different permutations of  a user like local user, external user, provisional user, local admin user, local support user, and so on. re-factoring this into users and roles will prevent the explosion of class hierarchy. A decorator does this at run-time, whereas a bridge does this at compile-time.

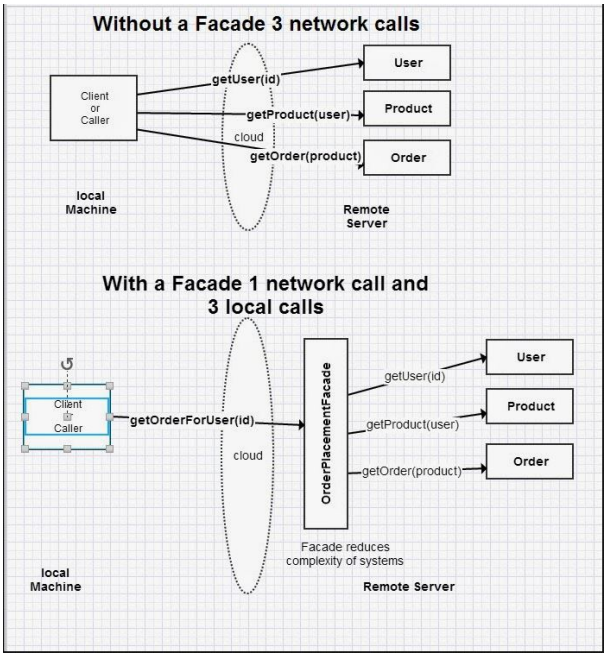



Java bridge design pattern

If you are wring unit tests, the bridge pattern is indispensable when it comes down to implementing mocks and mock services. You can use a bridge (or mock) web services in integration testing until the real services become available.

## Facade

is a higher-level interface to a subsystem of one or more classes. Think of Facade as a sort of **container** for other objects, as opposed to a **wrapper**.




Java facade design pattern

For example, when you have remote calls via EJBs, Web Services, RMI, etc, making fine grained remote calls each time can be expensive and adversely affects performance. So, in between the caller and the callee, you can define a facade that makes many fine grained local calls for a single remote call from the caller.It also simplifies the client code with just a single method call by hiding the complexity of the subsystem classes via a facade.

## Q. How does a strategy design pattern differ from a state design pattern?

The **Strategy pattern** is really about having a different implementation that accomplishes the same thing, so that one implementation can replace the other while the caller does not change. The **Strategy pattern deals with how an object performs a certain task?** — it encapsulates an algorithm or processing logic. The logic and algorithms can be improved and swapped without the caller not knowing anything about it.

For example, as a driver of a car, the steering wheel is your interface to interact, but over the years the implementations of steering mechanisms have improved like manual steering to power steering, and so on without affecting how you drive a car.

The **State pattern** is about doing different things based on the state, while leaving the caller relieved from the burden of accommodating every possible state. **The State pattern deals with**

**what state an object is in?** or **what type an object is?** — it encapsulates state-dependent behavior

For example, thread goes into different states like blocked, ready, running, etc. Placing a trade order on a stock market will have states like pending, filled, partially-filled, rejected, executed, etc.

This was intended to be very simple explanation with diagrams and examples. For more on design patterns with working code and example, search this blog or click on the “design pattern” tag cloud at the bottom.

## More Design Patterns Interview Questions & Answers:

1. GoF Design Patterns Interview questions & answers
2. JEE Patterns Interview Questions & Answers
3. Enterprise Integration Patterns Interview Questions & Answers

Arul Kumaran

Mechanical Engineer to Java contractor within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government firms. Published Java/JEE books in 2005, and sold **35K+ copies**. Books are outdated and replaced with this membership site with **1700+** registered users. Amazon.com reviews | Good reads reviews | LinkedIn Group