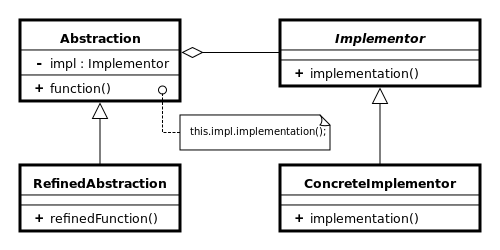
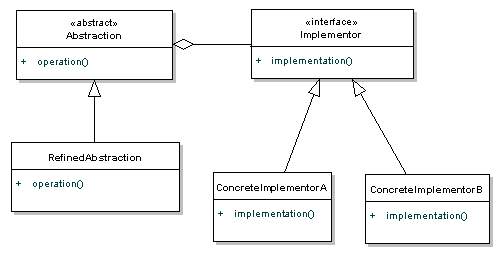
**Bridge Design (Structural) Pattern**

GOF: **Decouple abstractions from its implementations so that the two can vary independently**.

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.



As per DZONE



.

When:

Refactor to:

A N

/ \ / \

Aa(N) Ab(N) 1 2

A

/ \

Aa Ab

/ \ / \

Aa1 Aa2 Ab1 Ab2

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.

**In which scenario Bridge design pattern is used? When should we use the bridge pattern?**

**Bridge design pattern can be used when both abstraction and implementation can have different hierarchies independently** and we want to hide the implementation from the client application.

you want run-time binding of the implementation,

you have a proliferation of classes resulting from a coupled interface and numerous implementations,

you want to share an implementation among multiple objects, you need to map orthogonal class hierarchies.

The bridge pattern applies when there is a need to avoid permanent binding between an abstraction and an implementation and when the abstraction and implementation need to vary independently.

It is especially useful in situations where we need to manage complex hierarchies, anticipate changes in both Abstraction and Implementation, and maintain a separation between these two aspects.

**Advantages of the Bridge Design Pattern**

Flexibility and Extensibility: The pattern provides a high level of flexibility by allowing us to combine different Abstractions with different Implementations at runtime. This makes it easy to add new variations without altering the existing code (Open/Closed Principle)

Improved Maintainability: Since changes to the abstraction or implementation do not directly impact each other, maintenance becomes easier. We can modify one part without affecting the other, reducing the risk of unintended side effects.

**Disadvantages of the Bridge Design Pattern**

Overhead: The introduction of additional layers and abstractions can sometimes lead to a small overhead in terms of runtime performance and memory usage.

Potential Over-Abstraction: In some cases, designers might overuse the pattern, creating unnecessary layers of abstraction that complicate the system without providing tangible benefits.

Increased Code Volume: Implementing the Bridge pattern requires more code and classes, which can lead to a larger codebase.

**Real-world examples of Bridge Model Design Pattern**

Device Drivers: Operating systems use the Bridge pattern to manage device drivers. The Abstraction represents the generic device operations (e.g., read, write), while the Implementations are the similar solution structures for specific drivers in different hardware devices.

Database Systems: In database management systems, the Bridge pattern can be seen in the way database interfaces (Abstraction) are separated from the actual database engines (Implementation). Different databases can be supported by changing the Implementation while keeping the Abstraction intact.

Notification Systems: Notification systems that deliver messages through different channels (email, SMS, push notifications) use the Bridge pattern. The Abstraction represents the notification message, while Implementations handle the delivery mechanisms.

An example is given below.

**public** **class** User **implements** Resource {

**private** String name;

**private** **int** age;

**public** User(String name, **int** age) {

**super**();

**this**.name = name;

**this**.age = age;

}

// Constructor, getters, etc.

}

**public** **interface** Resource {

// Can be abstract class or interface

}

**import** com.ddlab.rnd.entity.Resource;

**public** **interface** Storage {

**void** store(Resource resource);

}

**import** com.ddlab.rnd.entity.Resource;

**public class** FileStorage **implements** Storage {

@Override

**public** **void** store(**final** Resource resource) {

// Store resource to a file on file system

System.***out***.println("File system ...");

}

}

**import** com.ddlab.rnd.entity.Resource;

**public** **class** PostgreSQLStorage **implements** Storage {

@Override

**public** **void** store(**final** Resource resource) {

System.***out***.println("DB implementation.");

// Store resource to PostgreSQL Database

}

}

**import** com.ddlab.rnd.entity.Resource;

**public class** CacheStorage **implements** Storage {

@Override

**public** **void** store(**final** Resource resource) {

// Store resource to cache

System.***out***.println("Cache system ...");

}

}

**import com.ddlab.rnd.entity.Resource;**

**import com.ddlab.rnd.storage.Storage;**

**public abstract class Repository<T extends Resource> {**

**protected Storage storage;**

**protected Repository(final Storage storage) {**

**this.storage = storage;**

**}**

**public abstract void save(final T resource);**

**}**

**import** com.ddlab.rnd.entity.User;

**import** com.ddlab.rnd.repository.Repository;

**public** **class** UserRepository **extends** Repository<User> {

**public** UserRepository(Storage storage) {

**super**(storage);

}

@Override

**public** **void** save(**final** User user) {

// Some user-specific logic

storage.store(user);

}

}

**import** com.ddlab.rnd.entity.User;

**import** com.ddlab.rnd.repository.Repository;

**import** com.ddlab.rnd.storage.PostgreSQLStorage;

**import** com.ddlab.rnd.storage.Storage;

**import** com.ddlab.rnd.storage.UserRepository;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

User user = **new** User("John Doe", 40);

Storage storage = **new** PostgreSQLStorage();

Repository<User> repository = **new** UserRepository(storage);

repository.save(user);

}

}