Bridge

Resources

- Video → 32. All Structural Design Patterns | Decorator, Proxy, Composite, Adapter, Bridge, Facade, FlyWeight
- Video → 26. Bridge Design Pattern | LLD of Bridge Pattern with Example | Low Level Design of Bridge Pattern

Definition

> ***The Bridge Design Pattern decouples an abstraction(high-level logic defines the "what") from its implementation(low-level details that define the "how") so that the two can evolve independently.***

The Problem

Let's consider an example of different living things with different respiratory mechanisms.

The naive approach of this implementation would be to introduce an abstraction for the respiratory mechanism, i.e. breathing process, and extend it to define the breathing process for each new type of living thing we introduce.

Let's look at the code example:

```java
// Approach without bridge: breathing logic hardcoded inside LivingThings
public abstract class LivingThings {
    abstract public void breathe();
}
```

```java
public class Dog extends LivingThings {

    // Breathing Process is tightly coupled to the LivingThings(abstraction)
    @Override
    public void breathe() {
        System.out.println("Dog: Breathes through its nose; Lives on land; Respiratory system: 2 lungs");
        System.out.println("Breathing Process: Inhales Oxygen from the air and Exhales Carbon Dioxide.");
    }
}
```

```java
public class Fish extends LivingThings {

    // Breathing Process is tightly coupled to the LivingThings(abstraction)
    @Override
```

```
    public void breathe() {
        System.out.println("Fish: Breathes through gills; Lives in water; Respiratory system: 2 gills");
        System.out.println("Breathing Process: Absorbs Oxygen from the water and releases Carbon Dioxide.");
    }
}
```

```
public class Tree extends LivingThings {

    // Breathing Process is tightly coupled to the LivingThings(abstraction)
    @Override
    public void breathe() {
        System.out.println("Tree: Breathes through leaves; Lives on land; Respiratory system: Leaves");
        System.out.println("Breathing Process: Inhales Carbon Dioxide and Exhales Oxygen as a result of photosynthesis.");
    }
}
```

```
public class Whale extends LivingThings {

    // Breathing Process is tightly coupled to the LivingThings(abstraction)
    @Override
    public void breathe() {
        System.out.println("Whale: Breathes through lungs; Lives in water; Respiratory system: 2 lungs");
        System.out.println("Breathing Process: Inhales Oxygen from the water and Exhales Carbon Dioxide");
    }
}
```

```
// Client
public class Demo {
    public static void main(String[] args) {
        System.out.println("======= Bridge Design Pattern - Problem Demo ======");
        LivingThings dog = new Dog();
        dog.breathe();
        LivingThings fish = new Fish();
        fish.breathe();
        LivingThings whale = new Whale();
        whale.breathe();
        LivingThings tree = new Tree();
        tree.breathe();
    }
}
```

Problems Here:

1. **Code Duplication**
   - Both `Dog` and `Whale` breathe through lungs → same logic repeated in multiple classes.
2. **Tight Coupling**
   - Breathing logic is tied to each animal class. You can't easily **reuse** breathing behavior for another animal or introduce a new type independently.
3. **Class Explosion**
   - Without Bridge, you might end up making separate classes for every combination:

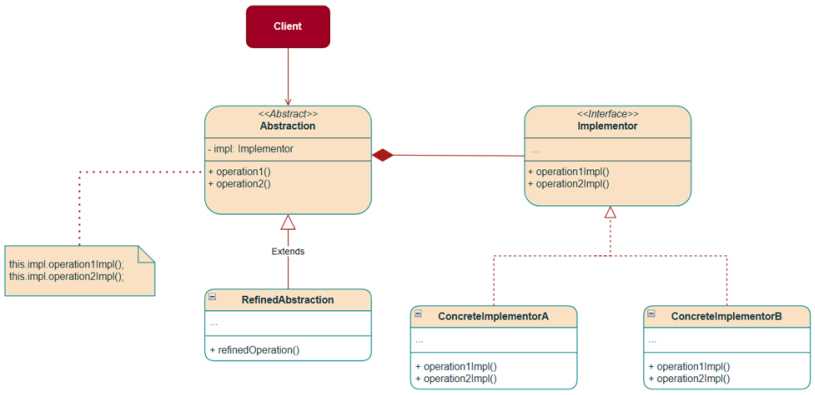
     `DogWithLungs`, `WhaleWithLungs`, `FishWithGills,` etc.

Solution: Bridge Design Pattern

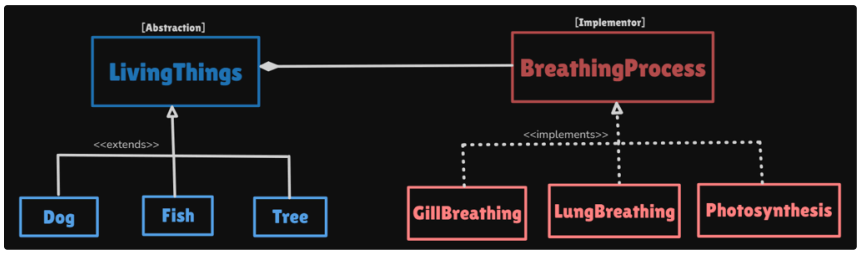
With the Bridge Pattern,

- Living Things (Dog, Fish, Whale, Tree) don't care how breathing is done.
- Breathing strategies (LungBreathing, GillBreathing, SkinBreathing) are separate.
- You can mix & match at runtime.
- Add a new breathing type? Just implement the Breathing Process.
- Add a new animal? Just extend Animal and plug in a breathing process.

Class Diagram



Structure of the Bridge Pattern



- **Abstraction** (`LivingThings`): High-level concept of a living thing and its operations.
- **Refined Abstraction** (`Dog` , `Fish` , `Whale` , `Tree`): Concrete types of living things with more refined functions.
- **Implementor** (`BreatheProcess`): Defines "how" breathing happens, the low-level details.
- **Concrete Implementors (** `LungBreathing` , `GillBreathing`): Actual breathing mechanisms defining reusable methods for respiratory mechanisms.

Implementation

```java
//Step 1: Implementor (Breathing process)
public interface BreathingProcess {
    void breathe();
}
```

```java
// Step 2: Concrete Implementor (various breathing processes)
public class GillBreathing implements BreathingProcess {
    @Override
    public void breathe() {
```

```
        System.out.println("Breathing through gills.");
    }
}
public class LungBreathing implements BreathingProcess {
    @Override
    public void breathe() {
        System.out.println("Breathing through lungs.");
    }
}
public class Photosynthesis implements BreathingProcess {
    @Override
    public void breathe() {
        System.out.println("Breathing through process of
photosynthesis. Releases Oxygen through leaves.");
    }
}
```

```
//Step 3: Abstraction for LivingThings
public abstract class LivingThings {
    // Reference to Implementor
    protected BreathingProcess breathingProcess;

    // Bridge connects Animal with Breathing
    public LivingThings(BreathingProcess breathingProcess) {
        this.breathingProcess = breathingProcess;
    }

    // Operation implemented by Implementor
    abstract public void breathe();
}
```

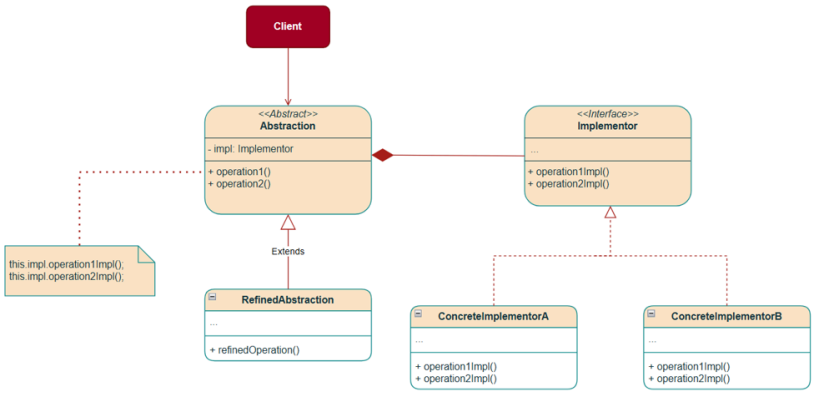
```
//Step 4: Refined Abstractions (Concrete LivingThings)
public class Dog extends LivingThings {

    public Dog(BreathingProcess breathingProcess) {
        super(breathingProcess);
    }

    @Override
    public void breathe() {
        System.out.print("Dog: ");
        breathingProcess.breathe(); // Operation implemented by
Implementor - defines the "HOW"
    }
}
public class Fish extends LivingThings {
    public Fish(BreathingProcess breathingProcess) {
        super(breathingProcess);
    }

    @Override
    public void breathe() {
        System.out.print("Fish: ");
        breathingProcess.breathe(); // Operation implemented by
Implementor - defines the "HOW"
    }
}
public class Tree extends LivingThings {
    public Tree(BreathingProcess breathingProcess) {
        super(breathingProcess);
    }

    @Override
    public void breathe() {
        System.out.print("Tree: ");
        breathingProcess.breathe(); // Operation implemented by
Implementor - defines the "HOW"
    }
}
```




```java
// Client Usage
public class Client {
    public static void main(String[] args) {
        System.out.println("======= Bridge Design Pattern - Solution Demo ======");

        LivingThings dog = new Dog(new LungBreathing());
        LivingThings fish = new Fish(new GillBreathing());
        LivingThings tree = new Tree(new Photosynthesis());

        dog.breathe();
        fish.breathe();
        tree.breathe();
    }
}
```

The Bridge Pattern allows for the independent addition of new `LivingThings` (e.g., `Frog`) or new Breathing types (e.g., `SkinBreathing` for amphibians). This enables a flexible and extensible design through mixing and matching.

Bridge Pattern vs Strategy Pattern

The patterns are mostly similar but differ in the intent:

- **Bridge**: Focuses on handling two hierarchies independently (abstraction & implementation). Like a LivingThings (abstraction) that has different Respiratory Mechanisms (implementation). LivingThings and Respiratory Mechanisms evolve independently.
- **Strategy**: Focuses on handling different behaviors dynamically. Like choosing the driving route on Google Maps, you can switch between "fastest", "shortest", or "avoid tolls" at runtime.