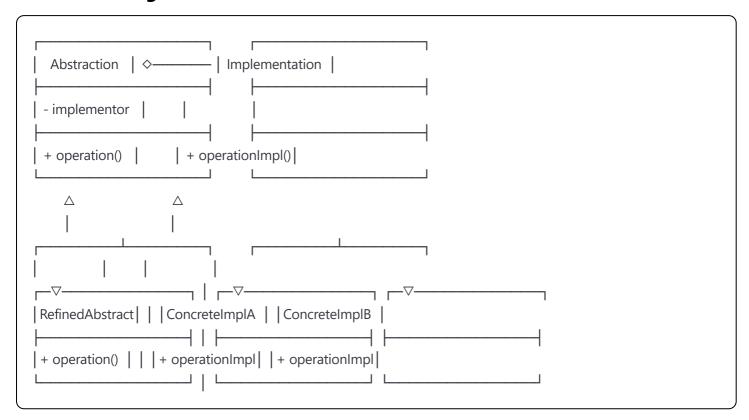
Bridge Design Pattern

Concept

The Bridge pattern separates an abstraction from its implementation so that both can vary independently. It uses composition over inheritance to achieve this separation.

UML Class Diagram



Java Implementation

Implementation Interface

```
java

// Implementation interface
interface Engine {
    void start();
    void stop();
    void accelerate();
}
```

Concrete Implementations

```
java
```

```
// Concrete Implementation A
class ElectricEngine implements Engine {
  @Override
  public void start() {
     System.out.println("Electric engine: Silent start");
  }
  @Override
  public void stop() {
     System.out.println("Electric engine: Power down");
  }
  @Override
  public void accelerate() {
     System.out.println("Electric engine: Smooth acceleration");
  }
}
// Concrete Implementation B
class PetrolEngine implements Engine {
  @Override
  public void start() {
     System.out.println("Petrol engine: Ignition start");
  }
  @Override
  public void stop() {
     System.out.println("Petrol engine: Engine off");
  }
  @Override
  public void accelerate() {
     System.out.println("Petrol engine: Revving up");
  }
}
// Concrete Implementation C
class DieselEngine implements Engine {
  @Override
  public void start() {
     System.out.println("Diesel engine: Compression start");
  }
  @Override
  public void stop() {
     System.out.println("Diesel engine: Shutdown");
```

```
@Override
public void accelerate() {
    System.out.println("Diesel engine: Turbo acceleration");
}
```

Abstraction

```
java
// Abstraction
abstract class Vehicle {
  protected Engine engine;
  public Vehicle(Engine engine) {
     this.engine = engine;
  }
  public abstract void drive();
  public abstract void park();
  // Bridge methods
  protected void startEngine() {
     engine.start();
  }
  protected void stopEngine() {
     engine.stop();
  }
  protected void accelerateEngine() {
     engine.accelerate();
  }
}
```

Refined Abstractions

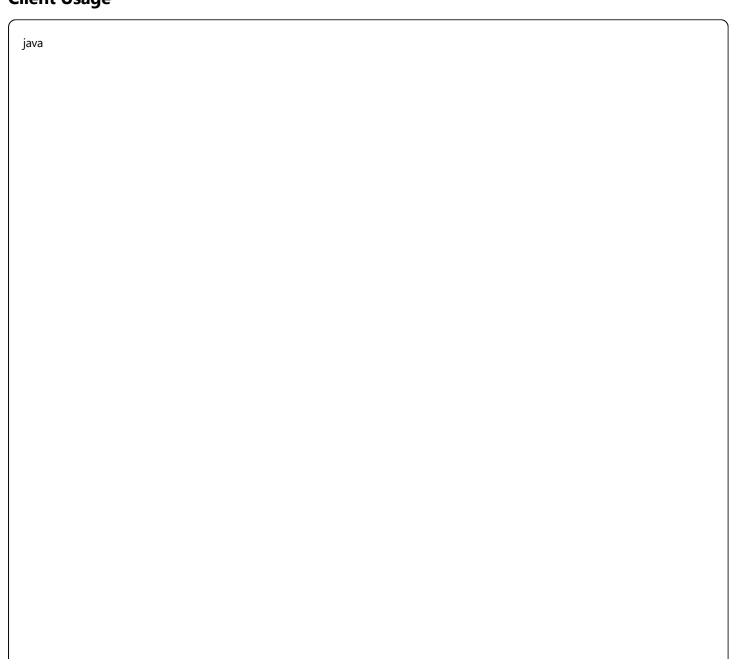
java			

```
// Refined Abstraction A
class Car extends Vehicle {
  public Car(Engine engine) {
     super(engine);
  }
  @Override
  public void drive() {
     System.out.println("Car: Starting to drive");
     startEngine();
     accelerateEngine();
     System.out.println("Car: Driving on road");
  }
  @Override
  public void park() {
     System.out.println("Car: Parking in garage");
     stopEngine();
  }
}
// Refined Abstraction B
class Truck extends Vehicle {
  public Truck(Engine engine) {
     super(engine);
  }
  @Override
  public void drive() {
     System.out.println("Truck: Starting heavy-duty drive");
     startEngine();
     accelerateEngine();
     System.out.println("Truck: Carrying heavy load");
  }
  @Override
  public void park() {
     System.out.println("Truck: Parking at loading dock");
     stopEngine();
  }
}
// Refined Abstraction C
class Motorcycle extends Vehicle {
  public Motorcycle(Engine engine) {
     super(engine);
```

```
@Override
public void drive() {
    System.out.println("Motorcycle: Quick start");
    startEngine();
    accelerateEngine();
    System.out.println("Motorcycle: Weaving through traffic");
}

@Override
public void park() {
    System.out.println("Motorcycle: Compact parking");
    stopEngine();
}
```

Client Usage



```
public class BridgePatternDemo {
  public static void main(String[] args) {
     // Different combinations of vehicles and engines
     // Electric car
     Vehicle electricCar = new Car(new ElectricEngine());
     System.out.println("=== Electric Car ===");
     electricCar.drive();
     electricCar.park();
     System.out.println();
     // Diesel truck
     Vehicle dieselTruck = new Truck(new DieselEngine());
     System.out.println("=== Diesel Truck ===");
     dieselTruck.drive();
     dieselTruck.park();
     System.out.println();
     // Petrol motorcycle
     Vehicle petrolBike = new Motorcycle(new PetrolEngine());
     System.out.println("=== Petrol Motorcycle ===");
     petrolBike.drive();
     petrolBike.park();
     System.out.println();
     // Electric truck (new combination)
     Vehicle electricTruck = new Truck(new ElectricEngine());
     System.out.println("=== Electric Truck ===");
     electricTruck.drive();
     electricTruck.park();
  }
}
```

Expected Output

=== Electric Car ===

Car: Starting to drive

Electric engine: Silent start

Electric engine: Smooth acceleration

Car: Driving on road
Car: Parking in garage

Electric engine: Power down

=== Diesel Truck ===

Truck: Starting heavy-duty drive Diesel engine: Compression start Diesel engine: Turbo acceleration

Truck: Carrying heavy load Truck: Parking at loading dock Diesel engine: Shutdown

=== Petrol Motorcycle === Motorcycle: Quick start

Petrol engine: Ignition start
Petrol engine: Revving up

Motorcycle: Weaving through traffic

Motorcycle: Compact parking
Petrol engine: Engine off

=== Electric Truck ===

Truck: Starting heavy-duty drive

Electric engine: Silent start

Electric engine: Smooth acceleration

Truck: Carrying heavy load
Truck: Parking at loading dock
Electric engine: Power down

Advantages

- Separation of Concerns: Abstraction and implementation can evolve independently
- Runtime Binding: Implementation can be selected or changed at runtime
- Extensibility: New abstractions and implementations can be added without affecting existing code
- Reduced Coupling: Client code depends only on abstraction, not implementation
- Multiple Implementations: One abstraction can work with multiple implementations

Disadvantages

- Increased Complexity: More classes and interfaces to maintain
- Indirection: Extra layer between client and actual implementation

- **Design Overhead**: May be overkill for simple scenarios
- Learning Curve: Developers need to understand the bridge relationship

Common Use Cases

- Database Drivers: JDBC drivers for different databases
- **GUI Frameworks**: Same UI components on different platforms
- **Graphics Rendering**: Different rendering engines (OpenGL, DirectX)
- **Messaging Systems**: Different protocols (HTTP, TCP, UDP)
- Payment Processing: Different payment gateways
- **Device Drivers**: Same interface for different hardware

Key Differences from Other Patterns

- vs Strategy: Bridge separates abstraction from implementation; Strategy encapsulates algorithms
- vs Adapter: Bridge is designed upfront; Adapter makes incompatible interfaces work together
- **vs State**: Bridge focuses on varying implementations; State focuses on varying behavior based on state