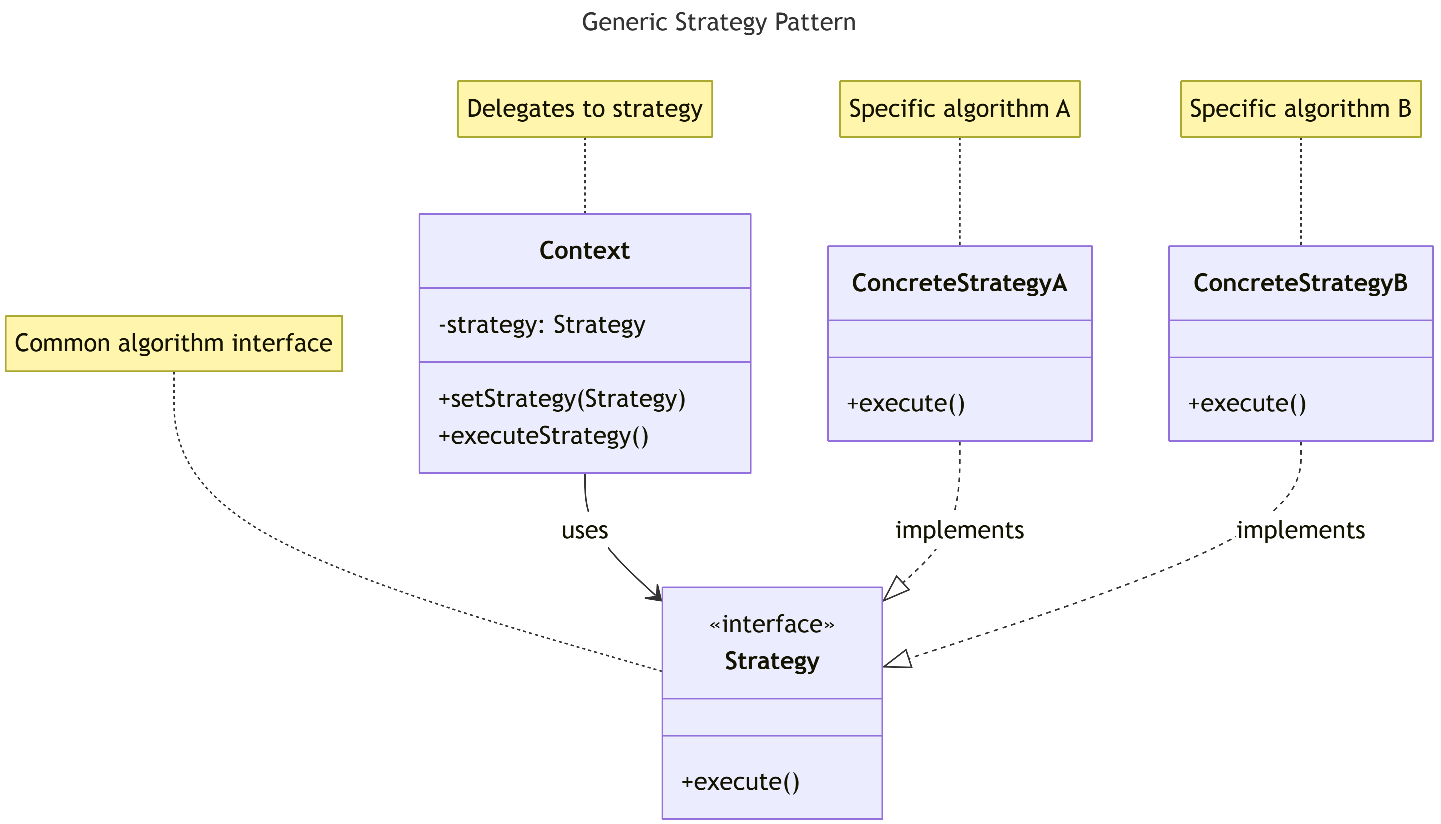
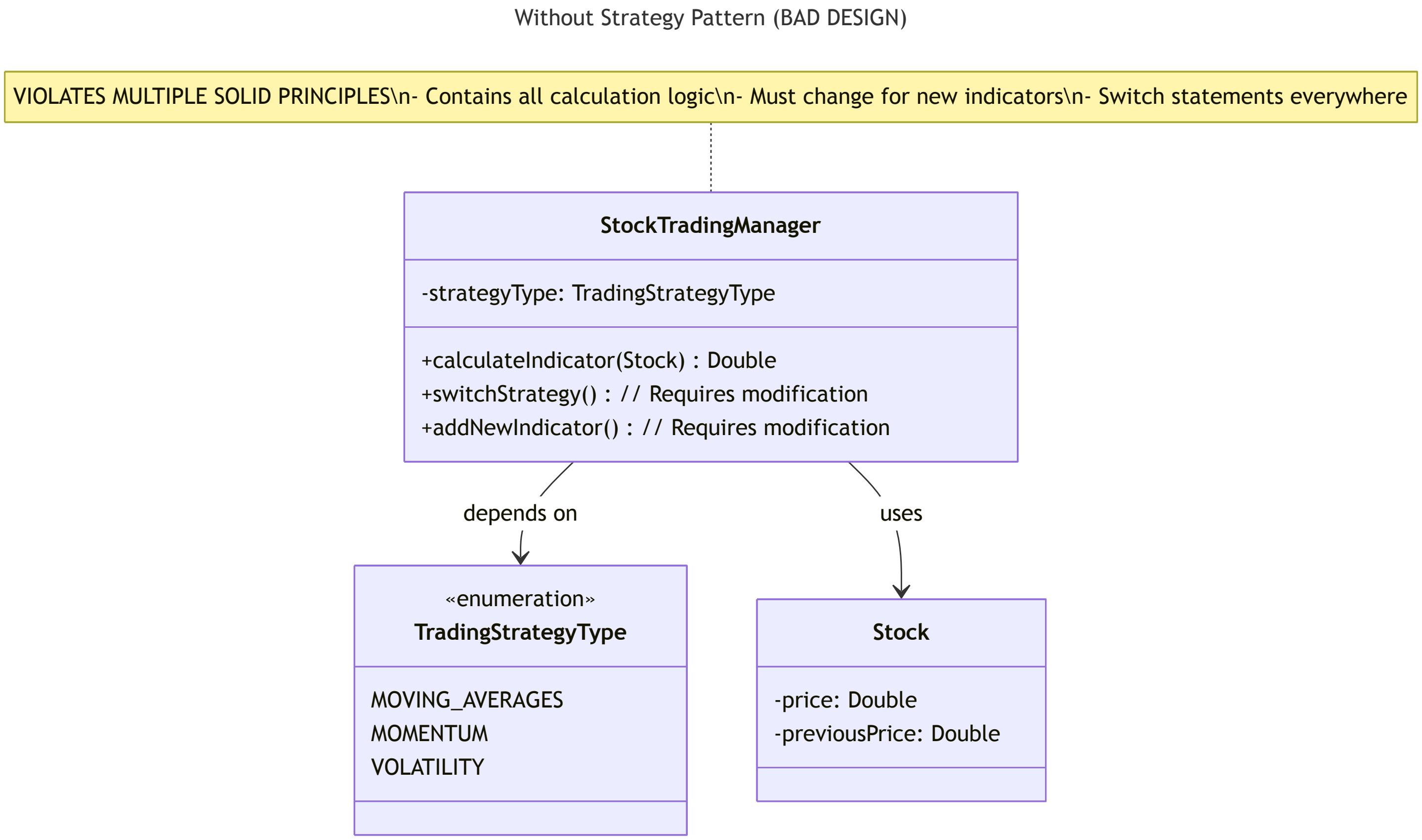
Generic Diagram



Strategy Pattern for Stock Trading Indicators

Without Strategy Pattern


Without Strategy Pattern (BAD DESIGN)
VIOLATES MULTIPLE SOLID PRINCIPLES\n- Contains all calculation logic\n- Must change for new indicators\n- Switch statements everywhere
StockTradingManager
-strategyType: TradingStrategyType
+calculateIndicator(Stock) : Double
+switchStrategy() : // Requires modification
+addNewIndicator() : // Requires modification
depends on
uses
«enumeration»
TradingStrategyType
MOVING_AVERAGES
MOMENTUM
VOLATILITY
Stock
-price: Double
-previousPrice: Double

```
// BAD - Without Strategy Pattern
public class StockTradingManager {
    private TradingStrategyType strategyType;

    public StockTradingManager(TradingStrategyType strategyType) {
        this.strategyType = strategyType;
    }

    public Double calculateIndicator(Stock stock) {
        switch (strategyType) {
            case MOVING_AVERAGES:
                return (stock.getPrice() + stock.getPreviousPrice()) / 2; // ❌ Logic m
            case MOMENTUM:
                return stock.getPrice() - stock.getPreviousPrice(); // ❌ Logic mixed
            case VOLATILITY:
                return Math.abs(stock.getPrice() - stock.getPreviousPrice()); // ❌ Log
        }
        throw new RuntimeException("Invalid strategy type"); // ❌ Runtime errors
        // Add new indicator = modify this method ❌
    }

    // Every new feature requires modifying this class ❌
}
```

SOLID Principles Violated Without Strategy Pattern

🔴 Single Responsibility Principle (SRP)

- StockTradingManager handles all calculation logic for different indicators
- One class responsible for multiple calculation algorithms

🔴 Open/Closed Principle (OCP)

- Must modify StockTradingManager to add new trading indicators
- Not open for extension, requires modification

🔴 Dependency Inversion Principle (DIP)

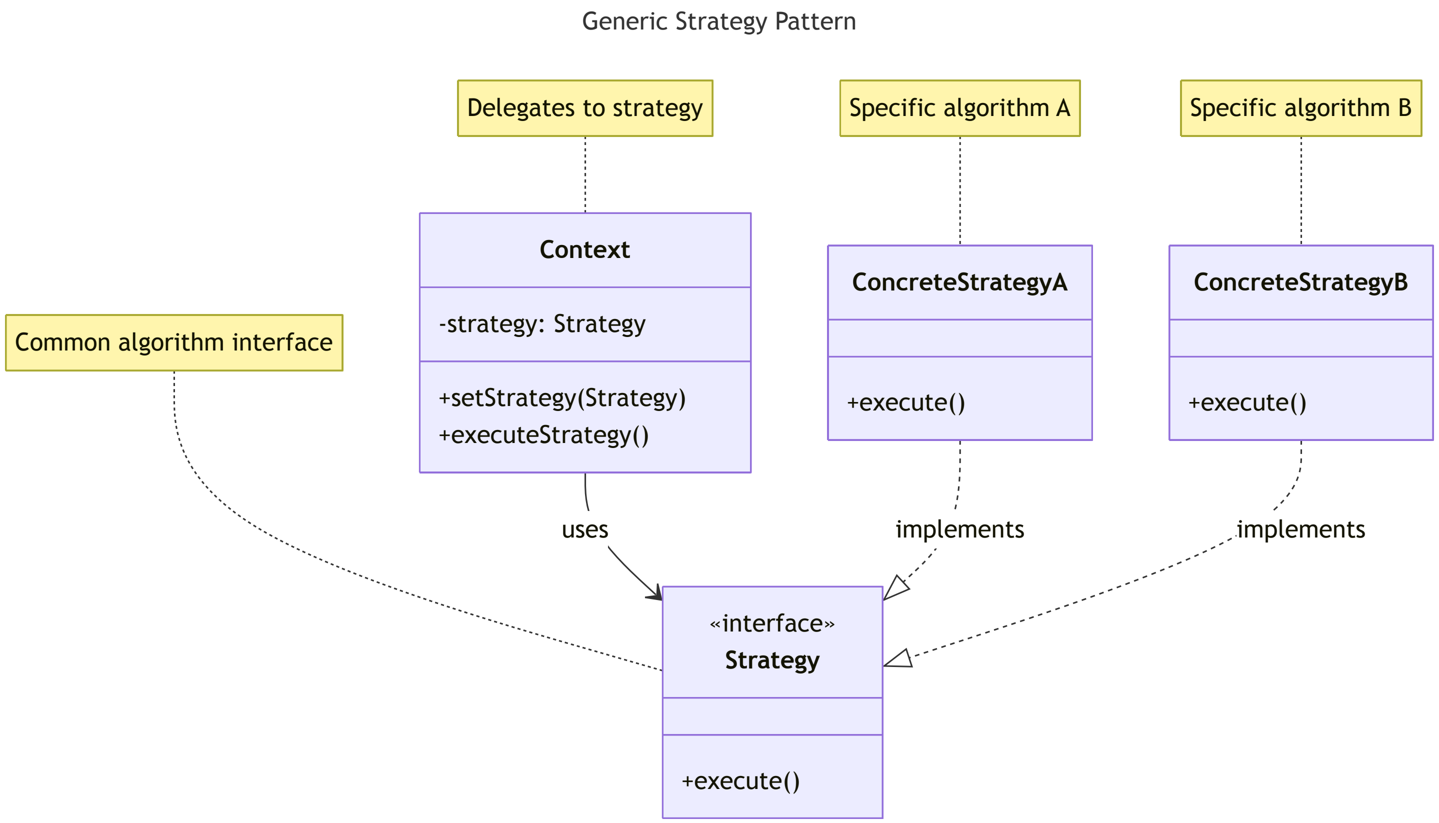
- High-level StockTradingManager depends on low-level calculation details
- No abstraction between manager and calculation logic

🔴 Interface Segregation Principle (ISP)

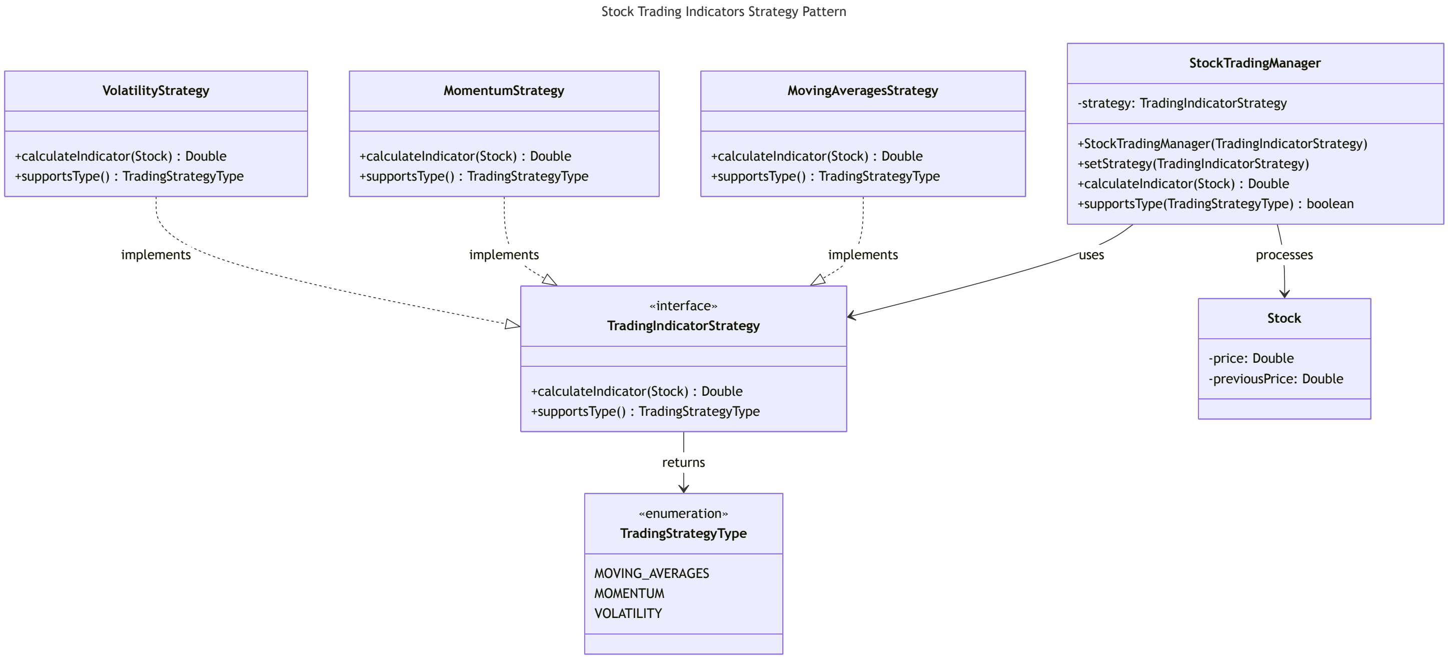
- All calculation logic mixed in single method
- No separation of different indicator concerns

With Strategy Pattern

Generic Diagram



Specific Diagram


Stock Trading Indicators Strategy Pattern
VolatilityStrategy
+calculateIndicator(Stock) : Double
+supportsType() : TradingStrategyType
MomentumStrategy
+calculateIndicator(Stock) : Double
+supportsType() : TradingStrategyType
MovingAveragesStrategy
+calculateIndicator(Stock) : Double
+supportsType() : TradingStrategyType
StockTradingManager
-strategy: TradingIndicatorStrategy
+StockTradingManager(TradingIndicatorStrategy)
+setStrategy(TradingIndicatorStrategy)
+calculateIndicator(Stock) : Double
+supportsType(TradingStrategyType) : boolean
implements
implements
implements
uses
processes
«interface»
TradingIndicatorStrategy
+calculateIndicator(Stock) : Double
+supportsType() : TradingStrategyType
Stock
-price: Double
-previousPrice: Double
returns
«enumeration»
TradingStrategyType
MOVING_AVERAGES
MOMENTUM
VOLATILITY

How Strategy Pattern Helps Here

Key Benefits:

- **Algorithm Encapsulation**: Each trading indicator encapsulated in separate strategy class
- **Runtime Strategy Switching**: Change indicators dynamically without creating new managers
- **Easy Extension**: Add new indicators without modifying existing code
- **Single Responsibility**: Each strategy handles one specific calculation
- **Testability**: Test each indicator calculation independently
- **Maintainability**: Indicator-specific logic isolated in separate classes

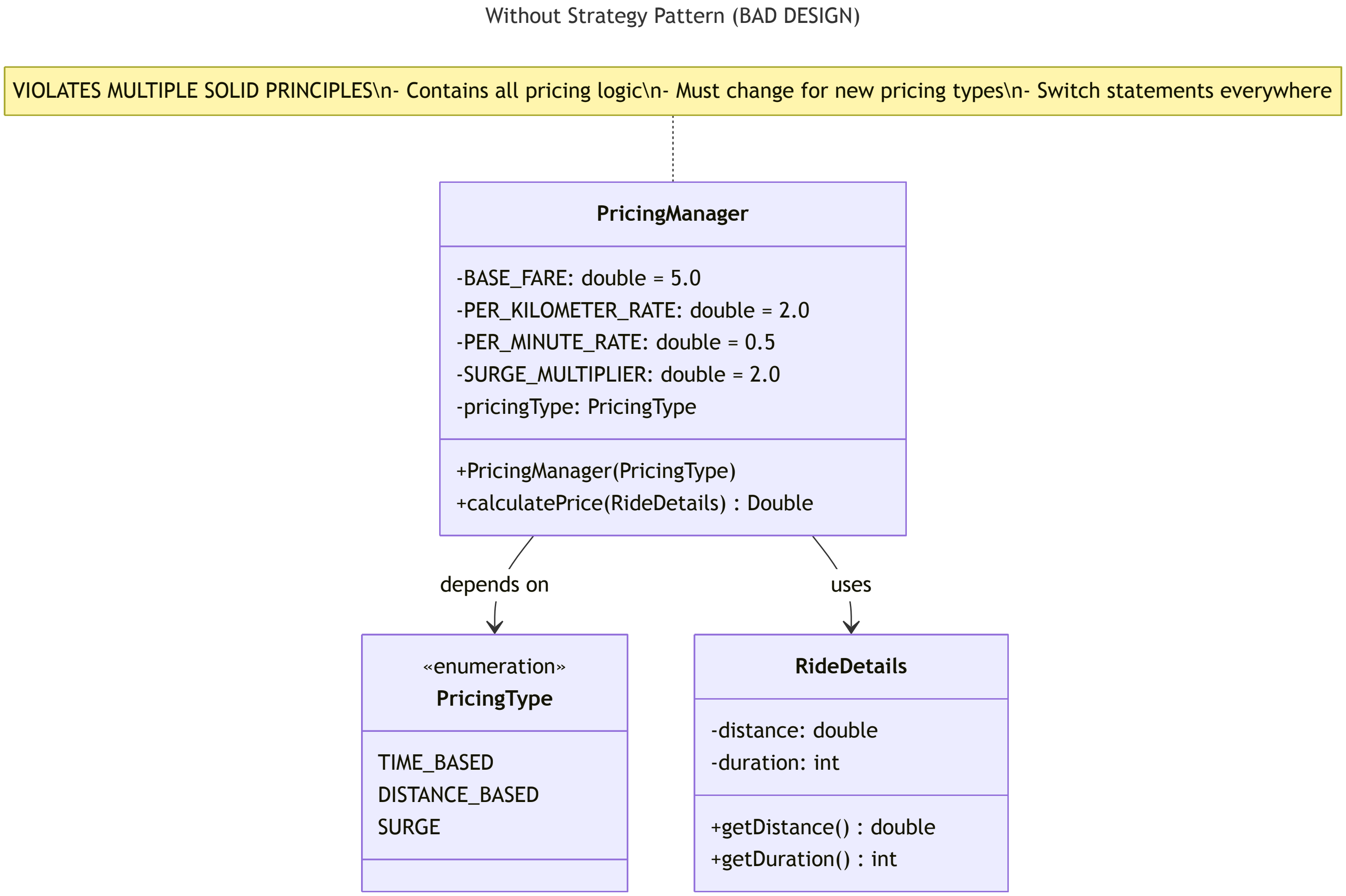
Usage Examples

```java
// Direct strategy usage ✅
StockTradingManager manager = new StockTradingManager(new MovingAveragesStrategy());
Double result = manager.calculateIndicator(stock);

// Runtime strategy switching ✅
manager.setStrategy(new MomentumStrategy());
Double momentumResult = manager.calculateIndicator(stock);

// Using with different stocks ✅
Stock appleStock = new Stock(150.0, 145.0);
Stock googleStock = new Stock(2800.0, 2750.0);

manager.setStrategy(new VolatilityStrategy());
Double appleVolatility = manager.calculateIndicator(appleStock);
Double googleVolatility = manager.calculateIndicator(googleStock);
```

The refactored design is now **extensible, maintainable, and follows SOLID principles**!

Strategy Pattern for Pricing Calculation

Without Strategy Pattern


Without Strategy Pattern (BAD DESIGN)
VIOLATES MULTIPLE SOLID PRINCIPLES\n- Contains all pricing logic\n- Must change for new pricing types\n- Switch statements everywhere
PricingManager
-BASE_FARE: double = 5.0
-PER_KILOMETER_RATE: double = 2.0
-PER_MINUTE_RATE: double = 0.5
-SURGE_MULTIPLIER: double = 2.0
-pricingType: PricingType
+PricingManager(PricingType)
+calculatePrice(RideDetails) : Double
depends on
uses
«enumeration»
PricingType
TIME_BASED
DISTANCE_BASED
SURGE
RideDetails
-distance: double
-duration: int
+getDistance() : double
+getDuration() : int

```java
// BAD - Without Strategy Pattern
public class PricingManager {

    private static final double BASE_FARE = 5.0; // Base fare amount
    private static final double PER_KILOMETER_RATE = 2.0; // Rate per kilometer
    private static final double PER_MINUTE_RATE = 0.5; // Rate per minute
    private static final double SURGE_MULTIPLIER = 2.0; // Surge pricing multiplier

    private PricingType pricingType;

    public PricingManager(PricingType pricingType) {
        this.pricingType = pricingType;
    }

    public Double calculatePrice(RideDetails rideDetails) {
        switch (pricingType) {
            case DISTANCE_BASED:
                return BASE_FARE + PER_KILOMETER_RATE * rideDetails.getDistance(); // ❌
            case TIME_BASED:
                return BASE_FARE + PER_MINUTE_RATE * rideDetails.getDuration(); // ❌ L
            case SURGE:
                return BASE_FARE * SURGE_MULTIPLIER; // ❌ Logic mixed
        }
        throw new IllegalArgumentException("Invalid pricing type"); // ❌ Runtime error
        // Add new pricing type = modify this method ❌
    }
}
```

SOLID Principles Violated Without Strategy Pattern

🔴 Single Responsibility Principle (SRP)

- PricingManager handles all pricing calculations for different strategies
- One class responsible for multiple pricing algorithms

🔴 Open/Closed Principle (OCP)

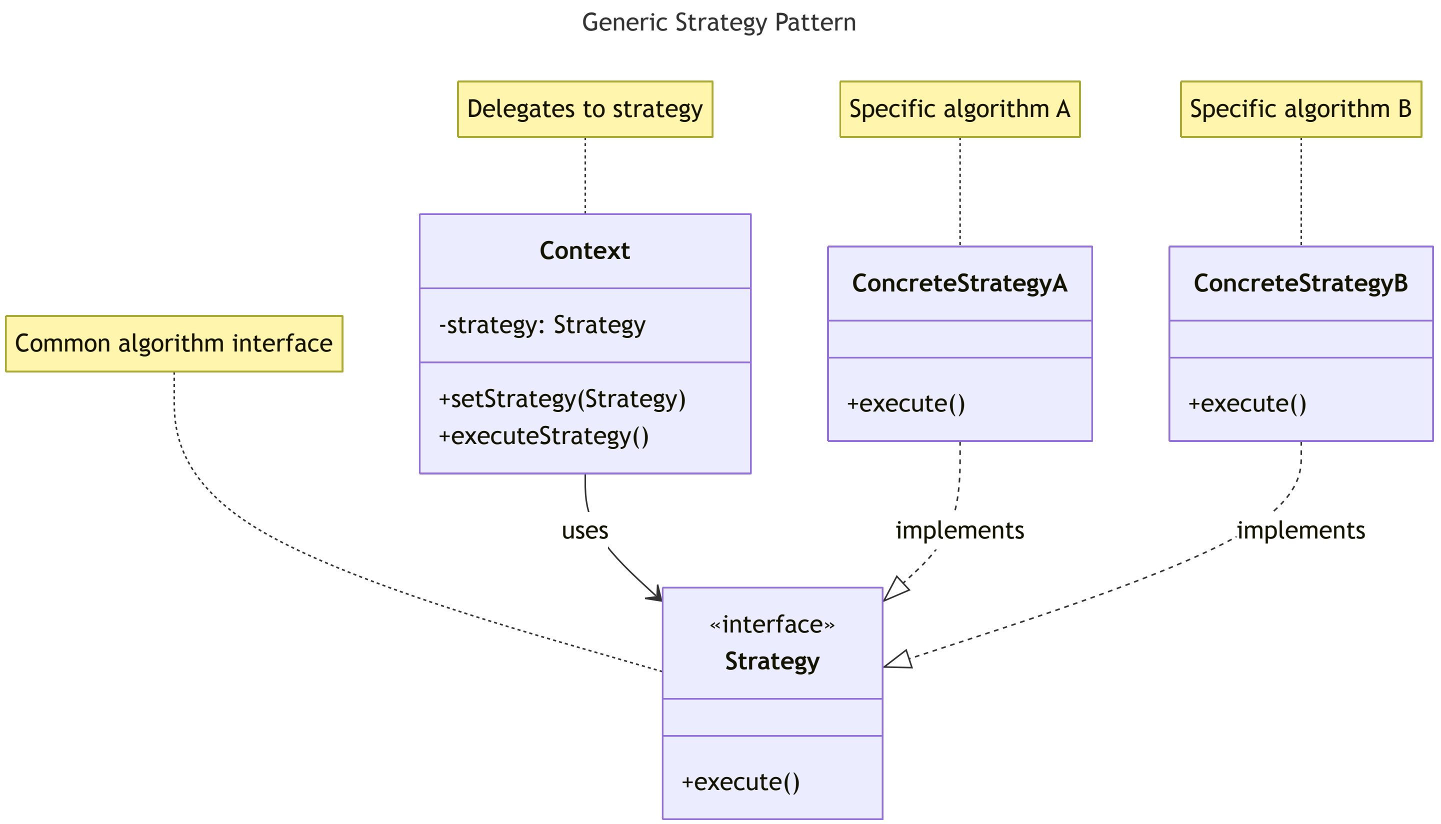
- Must modify PricingManager to add new pricing strategies
- Not open for extension, requires modification

🔴 Dependency Inversion Principle (DIP)

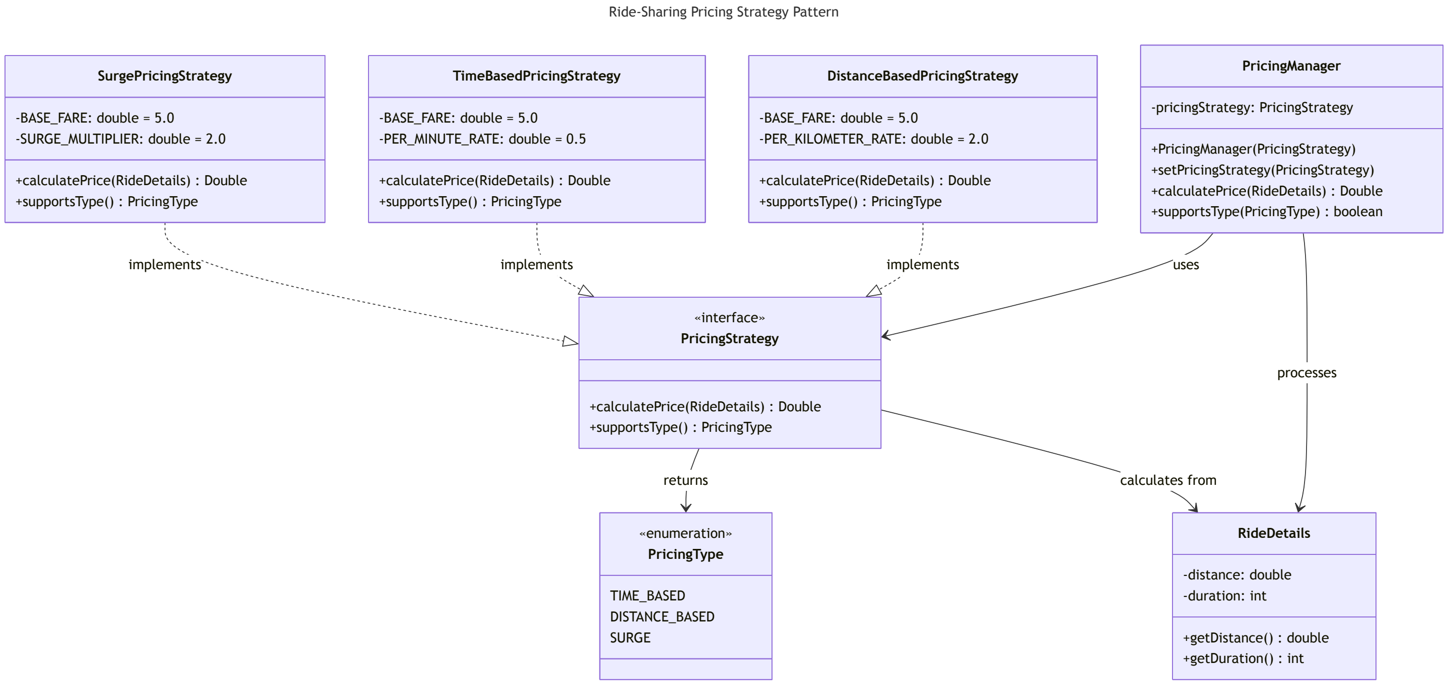
- High-level PricingManager depends on low-level calculation details
- No abstraction between manager and pricing logic

With Strategy Pattern

Generic Diagram



Specific Diagram


Ride-Sharing Pricing Strategy Pattern
SurgePricingStrategy
-BASE_FARE: double = 5.0
-SURGE_MULTIPLIER: double = 2.0
+calculatePrice(RideDetails) : Double
+supportsType() : PricingType
TimeBasedPricingStrategy
-BASE_FARE: double = 5.0
-PER_MINUTE_RATE: double = 0.5
+calculatePrice(RideDetails) : Double
+supportsType() : PricingType
DistanceBasedPricingStrategy
-BASE_FARE: double = 5.0
-PER_KILOMETER_RATE: double = 2.0
+calculatePrice(RideDetails) : Double
+supportsType() : PricingType
PricingManager
-pricingStrategy: PricingStrategy
+PricingManager(PricingStrategy)
+setPricingStrategy(PricingStrategy)
+calculatePrice(RideDetails) : Double
+supportsType(PricingType) : boolean
implements
implements
implements
uses
«interface»
PricingStrategy
+calculatePrice(RideDetails) : Double
+supportsType() : PricingType
processes
returns
calculates from
«enumeration»
PricingType
TIME_BASED
DISTANCE_BASED
SURGE
RideDetails
-distance: double
-duration: int
+getDistance() : double
+getDuration() : int

How Strategy Pattern Helps Here

Key Benefits:

- **Algorithm Encapsulation**: Each pricing strategy encapsulated in separate strategy class
- **Runtime Strategy Switching**: Change pricing models dynamically based on market conditions
- **Easy Extension**: Add new pricing models (e.g., premium, discount) without modifying existing code
- **Single Responsibility**: Each strategy handles one specific pricing calculation
- **Testability**: Test each pricing strategy independently with different ride scenarios
- **Maintainability**: Pricing-specific logic isolated in separate classes
- **Business Flexibility**: Easy to modify pricing rules per strategy for market adaptation

Key Transformations

Before (BAD) → After (GOOD)

Aspect	Without Strategy	With Strategy
Algorithm Storage	All in switch statement	Separate strategy classes

Aspect	Without Strategy	With Strategy
Adding New Pricing	Modify PricingManager	Create new strategy class
Runtime Switching	Create new manager instance	Call setPricingStrategy() method
Responsibilities	Manager handles all calculations	Each strategy handles one calculation
Testing	Test entire manager	Test each strategy independently
Constants	All constants in one class	Constants distributed to relevant strategies

SOLID Principles Now Followed ✅

- **SRP**: Each strategy class has single pricing calculation responsibility
- **OCP**: Open for extension (new strategies), closed for modification
- **DIP**: PricingManager depends on abstraction (PricingStrategy)
- **ISP**: Clean interface with only needed methods

Usage Examples

```java
// Direct strategy usage ✅
PricingManager manager = new PricingManager(new DistanceBasedPricingStrategy());
Double price = manager.calculatePrice(rideDetails);

// Runtime strategy switching ✅
manager.setPricingStrategy(new SurgePricingStrategy());
Double surgePrice = manager.calculatePrice(rideDetails);

// Different ride scenarios ✅
RideDetails shortRide = new RideDetails(2.0, 8);    // 2km, 8min
RideDetails longRide = new RideDetails(25.0, 45);   // 25km, 45min
RideDetails trafficJam = new RideDetails(5.0, 60);  // 5km, 60min

// Test all strategies with different scenarios
for (PricingType type : PricingType.values()) {
    PricingStrategy strategy = PricingStrategyFactory.createStrategy(type);
    manager.setPricingStrategy(strategy);
    System.out.println(type + ": $" + manager.calculatePrice(shortRide));
}
```

The refactored design is now **flexible, maintainable, and follows SOLID principles** - perfect for a ride-sharing app that needs dynamic pricing strategies!