Object-Oriented Software Analysis and Design

School of Computer Science
University of Windsor

Gang of Four's Pattern Catalog

Creational	Structural	Behavioral
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

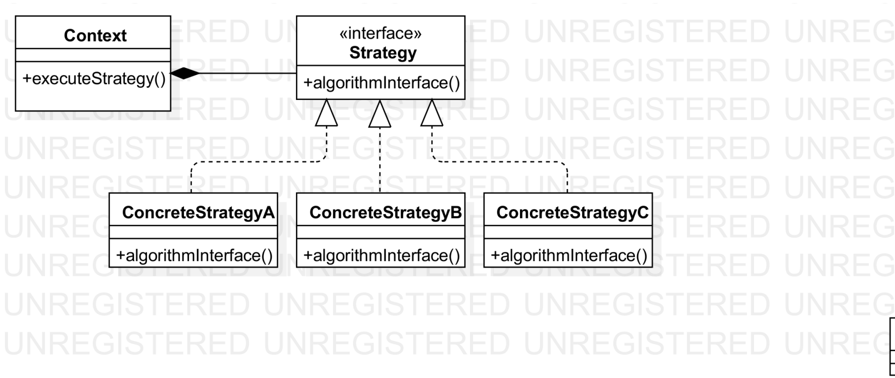
Strategy: Intent

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Also known as
  - Policy

Strategy: Applicability

- Use the Strategy pattern when
  - many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
  - you need different variants of an algorithm.
  - an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
  - a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Strategy Pattern: Structure



Strategy: Example

```
//Strategy
public interface Strategy {
   public int doOperation(int num1, int num2);
}
```

Strategy: Example (contd.)

```
//OperationAdd
public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}
```

Strategy: Example (contd.)

```java
//OperationSubtract
public class OperationSubtract implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}
```

Strategy: Example (contd.)

```
//OperationMultiply
public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}
```

Strategy: Example (contd.)

```
//Context
public class Context {
    private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

Strategy: Example (contd.)

```
public class PracticeStrategy {

    public static void main(String[] args) {
        Context context = new Context(new OperationAdd());
        System.out.println("10 + 5 = " +
            context.executeStrategy(10, 5));

        context = new Context(new OperationSubtract());
        System.out.println("10 - 5 = " +
            context.executeStrategy(10, 5));

        context = new Context(new OperationMultiply());
        System.out.println("10 * 5 = " +
            context.executeStrategy(10, 5));
    }
}
```

Strategy and Open-Closed Principle

- According to the strategy pattern, the behaviors of a class should not be inherited. Instead, they should be encapsulated using interfaces.
- This is compatible with the Open-Closed principle (OCP), which proposes that classes should be open for extension but closed for modification.
- The strategy pattern uses composition instead of inheritance. In the strategy pattern, behaviors are defined as separate interfaces and specific classes that implement these interfaces.
- This allows better **decoupling** between the behavior and the class that uses the behavior. The behavior can be changed without breaking the classes that use it, and the classes can switch between behaviors by changing the specific implementation used without requiring any significant code changes.

Strategy: Related Patterns

- Flyweight: Strategy objects often make good flyweights.