Strategy Design Pattern - Study Notes

📋 Pattern Overview

Definition: Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable at runtime.

Type: Behavioral Design Pattern

Problem Solved: Eliminates multiple conditional statements and allows dynamic behavior changes without modifying the context class.

🏗️ Pattern Structure

Core Components

1. **Strategy Interface**
   - Defines common interface for all concrete strategies
   - Declares method(s) that context uses to execute strategy
2. **Concrete Strategy Classes**
   - Implement the strategy interface
   - Contain specific algorithm implementations
   - Can be swapped interchangeably
3. **Context Class**
   - Maintains reference to strategy object
   - Delegates work to strategy instead of implementing directly
   - Provides interface for clients to configure strategies
4. **Client**
   - Creates specific strategy objects
   - Passes strategy to context

🔄 How It Works

Basic Flow

```
Client → Creates Strategy → Passes to Context → Context delegates to Strategy
```

Execution Steps

1. **Setup**: Context receives strategy object reference

2. **Execution**: Context calls strategy method when behavior needed
3. **Delegation**: Strategy object executes its specific algorithm
4. **Runtime Change**: Context can switch to different strategy anytime

---

💻 Robot Example Breakdown

Strategy Interfaces

```java
interface Talkable { void talk(); }
interface Walkable { void walk(); }
interface Flyable { void fly(); }
interface Projectable { void project(); }
```

Concrete Strategies

- **Talking**: `NormalTalk`, `NoTalk`, `MultilingualTalk`
- **Walking**: `NormalWalk`, `NoWalk`, `FastWalk`
- **Flying**: `NormalFly`, `NoFly`, `JetThrusterFly`
- **Projecting**: `NormalProject`, `NoProject`, `HDProject`

Context (Robot Class)

- Holds strategy references: `talkBehavior`, `walkBehavior`, etc.
- Delegates to strategies: `talkBehavior.talk()`
- Allows runtime changes: `setTalkBehavior(new NormalTalk())`

✅ Advantages

🎯 Flexibility

- Runtime behavior switching
- Mix and match different algorithms
- No need to modify context class for new behaviors

🧩 Extensibility

- Add new strategies without changing existing code
- Follow Open/Closed Principle (open for extension, closed for modification)

🔧 Maintainability

- Each algorithm isolated in separate class
- Easy to test individual strategies
- Clear separation of concerns

🚀 Performance

- Avoid conditional statements (`if-else`, `switch`)
- Strategies can be optimized independently

🔄 Reusability

- Same strategy can be used across different contexts
- Strategies are interchangeable

❌ Disadvantages

📚 Increased Complexity

- More classes to manage
- Additional abstraction layer

🧠 Client Knowledge

- Client must know about different strategies
- Need to understand when to use which strategy

💾 Memory Overhead

- Strategy objects consume memory
- May create objects that are rarely used

📎 Communication Overhead

- Context and strategy must share data
- May require passing parameters between them

🆚 Strategy vs Other Patterns

Strategy vs State Pattern

Strategy	State
Strategies are independent	States are aware of each other
Client chooses strategy	Context manages state transitions
Focus on algorithm variation	Focus on behavior based on state

Strategy vs Command Pattern

Strategy	Command
Encapsulates algorithms	Encapsulates requests
Multiple ways to do something	What to do and when
Runtime algorithm selection	Request queuing/logging

Strategy vs Template Method

Strategy	Template Method
Composition-based	Inheritance-based
Runtime flexibility	Compile-time structure
Whole algorithm varies	Parts of algorithm vary

🎯 When to Use Strategy Pattern

✅ Good Scenarios

- Multiple ways to perform a task
- Need runtime algorithm switching
- Want to avoid conditional statements
- Algorithm variations independent of clients
- Need to add new algorithms frequently

❌ Avoid When

- Only one algorithm needed
- Algorithms rarely change
- Simple conditional logic sufficient
- Performance is critical (overhead not acceptable)

🌟 Real-World Applications

Payment Processing

```
java

PaymentStrategy: CreditCard, PayPal, Cryptocurrency, BankTransfer
```

Sorting Algorithms

```
java

SortStrategy: BubbleSort, QuickSort, MergeSort, HeapSort
```

Compression

```
java

CompressionStrategy: ZIP, RAR, 7Z, TAR
```

Authentication

```
java

AuthStrategy: OAuth, LDAP, Database, Biometric
```

Navigation

```
java

RouteStrategy: Fastest, Shortest, Scenic, EcoFriendly
```

🎯 Strategy Pattern Core Principles

Key Design Principles

1. 🔒 **Encapsulate What Varies & Keep it Separate from What Remains Same**
   - Identify the aspects of your application that vary and separate them from what stays the same
   - Variable parts → Strategy implementations
   - Stable parts → Context class structure
2. 🚫 **Solution to Inheritance is NOT More Inheritance**
   - Don't solve inheritance problems by creating more inheritance hierarchies
   - Use composition instead of trying to fix inheritance with more inheritance
3. 🧩 **Composition Should be Favoured Over Inheritance**
   - "Has-a" relationship is more flexible than "Is-a" relationship
   - Allows runtime behavior changes

- Reduces tight coupling between classes

4. 🔌 **Code to Interface & NOT to Concretion**
   - Program against abstractions (interfaces) rather than concrete implementations
   - Makes code more flexible and extensible
   - Enables easy swapping of implementations
5. 🔄 **Do NOT Repeat Yourself (DRY)**
   - Avoid duplicating similar code across different strategy implementations
   - Extract common functionality into shared utilities or base classes
   - Each strategy should focus on its unique algorithm only

Applying These Principles to Robot Example

java

```java
// ✅ GOOD: Following the principles
interface Talkable {  // Code to interface
    void talk();
}

class Robot {
    private Talkable talkBehavior;  // Composition over inheritance

    public void setTalkBehavior(Talkable behavior) {  // Encapsulate what varies
        this.talkBehavior = behavior;
    }
}

// ❌ BAD: Violating the principles
class Robot {
    private String robotType;

    public void talk() {  // Not encapsulating what varies
        if (robotType.equals("combat")) {
            // Combat talk logic
        } else if (robotType.equals("assistant")) {
            // Assistant talk logic  (Repeating similar patterns)
        }
        // More inheritance hierarchies would make this worse
    }
}
```

🔍 Implementation Tips

Best Practices

1. **Interface Segregation**: Keep strategy interfaces focused and small
2. **Default Strategy**: Provide sensible default behavior
3. **Strategy Factory**: Use factory pattern to create strategies
4. **Null Object**: Use null object pattern for "no operation" strategies
5. **Strategy Validation**: Validate strategy compatibility with context

Common Pitfalls

1. **Over-engineering**: Don't use for simple two-option scenarios
2. **Strategy Explosion**: Too many strategies can become unwieldy
3. **Context Coupling**: Avoid tight coupling between context and strategies
4. **Data Sharing**: Minimize data passed between context and strategy

---

📝 Code Implementation Checklist

Strategy Interface

- [ ] Single responsibility method(s)
- [ ] Clear method signatures
- [ ] Appropriate parameter passing

Concrete Strategies

- [ ] Implement strategy interface
- [ ] Independent algorithm implementation
- [ ] No dependencies on other strategies

Context Class

- [ ] Strategy reference variable
- [ ] Delegation methods
- [ ] Strategy setter methods
- [ ] Constructor accepting strategy

Client Code

- [ ] Strategy object creation
- [ ] Context configuration
- [ ] Runtime strategy switching (if needed)

🎓 Study Questions

1. **Conceptual**: How does Strategy pattern promote Open/Closed Principle?
2. **Practical**: When would you choose Strategy over simple if-else statements?
3. **Design**: How would you implement a caching strategy system?
4. **Comparison**: What's the key difference between Strategy and State patterns?
5. **Implementation**: How would you handle strategy dependencies and configuration?

📚 Further Reading

- **Gang of Four Design Patterns** - Original strategy pattern documentation
- **Head First Design Patterns** - Strategy pattern chapter
- **Effective Java** - Strategy pattern best practices
- **Clean Code** - Avoiding conditionals with strategy pattern

---

*Remember: Strategy pattern is about **composition over inheritance** and **runtime flexibility over compile-time rigidity**.*