design-patterns # group-project

The Strategy Pattern

TCSS 305 Programming Practicum

> **Demo Project**
>
> The code examples in this guide come from the demo project: TCSS305-Game

The Strategy pattern solves one of the most common problems in software: selecting among multiple algorithms without scattering conditional logic throughout your code. Instead of writing if-else chains or switch statements to pick the right behavior, you encapsulate each algorithm behind a common interface and swap them in and out as needed. This guide covers the pattern's origin, its classic and modern Java implementations, and how it appears in our codebase using map-based dispatch with functional interfaces.

---

1 Brief History

The Strategy pattern was first catalogued by the "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) in their seminal 1994 book *Design Patterns: Elements of Reusable Object-Oriented Software*. However, the concept of encapsulating interchangeable algorithms predates this formalization--it is rooted in the principle of separating what varies from what stays the same, a fundamental concept in software engineering since the 1970s.

The pattern is also known as the **Policy** pattern. Over the years, its implementation idiom has evolved alongside the language:

- **Classic OOP Approach** (Pre-Java 8): Define a `Strategy` interface with a method like `execute()`. Each algorithm lives in a concrete class implementing this interface (`ConcreteStrategyA`, `ConcreteStrategyB`, etc.). A context object holds a reference to a `Strategy` and delegates algorithm execution to it. This works well but requires creating many small classes, one per algorithm.
- **Functional Approach** (Java 8+): With lambda expressions and method references, algorithms become first-class citizens. Instead of creating separate strategy classes, you

use functional interfaces (`Function`, `Supplier`, `Predicate`, etc.) and pass behavior directly as method references or lambdas. This is more concise and expressive for simple strategies.

- **Map-Based Strategy** (Modern Java): A contemporary refinement uses `Map` collections to associate keys (like enum values) with strategy implementations (method references or lambdas). This eliminates conditional logic entirely while maintaining all the benefits of the Strategy pattern. This is the approach our codebase demonstrates--using `Map<Move, BooleanSupplier>` for validation strategies and `Map<Move, Supplier<Point>>` for movement transformation strategies.

---

2 The Problem It Solves

Imagine you are building a navigation app that calculates routes from point A to point B. Different users have different preferences: some want the fastest route (even if it uses toll roads), others want the shortest distance, some want to avoid highways, and others prioritize scenic routes.

The naive approach would use conditional logic to select the routing algorithm: if the user selected "fastest," call the fastest-route method; if they selected "shortest," call the shortest-route method; and so on. As you add more routing options (avoid tolls, bicycle-friendly, wheelchair-accessible), your code becomes cluttered with if-else chains or switch statements. Every time you add a new algorithm, you must modify the routing logic, violating the Open/Closed Principle.

Worse, testing becomes difficult because the algorithm selection logic is tangled with the routing code. You cannot easily swap algorithms at runtime without complex conditional paths. The navigation class knows too much--it should not care about the details of each routing algorithm.

---

3 How the Pattern Solves It

The Strategy pattern introduces a key insight: **encapsulate each algorithm in a separate, interchangeable component**. Instead of the context object (the navigation app) containing algorithm selection logic, it delegates to a strategy object that knows how to perform the algorithm.

This pattern defines three roles:

3.1 Strategy (Interface or Functional Interface)

Declares a common interface for all supported algorithms. In the classic approach, this is an interface with an `execute()` method. In modern Java, this is often a functional interface like `Supplier<T>`, `Function<T,R>`, or `BooleanSupplier`.

3.2 Concrete Strategies (Implementations)

Each algorithm is encapsulated in a separate implementation. In the classic approach, these are classes implementing the Strategy interface. In modern Java, these are method references, lambdas, or even just methods that match the functional interface's signature.

3.3 Context

Maintains a reference to a Strategy object and delegates algorithm execution to it. The context is configured with a concrete strategy and calls its method without knowing which specific algorithm is being used.

The key insight: **replace conditional logic with polymorphism** (classic approach) or **function references** (modern approach). Rather than selecting algorithms with if-else statements, the context simply calls the strategy's method. The behavior changes based on which strategy is installed, not based on conditional branches.

---

4 Technical Implementation in Our Codebase

This project demonstrates a modern, functional take on the Strategy pattern using `Map` collections and Java's functional interfaces. Instead of creating strategy classes and using polymorphism, we use method references stored in maps for direct, efficient strategy lookup.

> **Modern Java Feature**
>
> This implementation uses **lambdas**, **method references**, and **functional interfaces** (Java 8+) to create a concise, map-based strategy implementation. For a detailed explanation of these language features, see the Functional Programming guide.

4.1 The Context: Game Class

The `Game` class is our context--it needs to validate moves and transform the game piece's location. Different move directions require different validation logic and different coordinate transformations.

Strategy Maps

Instead of using if-else or switch statements to handle different move directions, the `Game` class uses two strategy maps. These maps associate each `Move` direction with its corresponding strategy implementation:

```
public final class Game implements PropertyChangeEnabledGameControls {
    /** Map of move directions to their validation predicates. */
    private final Map<Move, BooleanSupplier> myMoveValidators;

    /** Map of move directions to their transformation functions. */
    private final Map<Move, Supplier<Point>> myMovements;

    private Game() {
        // Initialize validation strategies
        myMoveValidators = Map.of(
            Move.UP, this::isUpValid,
            Move.DOWN, this::isDownValid,
            Move.LEFT, this::isLeftValid,
            Move.RIGHT, this::isRightValid);

        // Initialize movement transformation strategies
        myMovements = Map.of(
            Move.UP, () -> myLocation.transform(0, -1),
            Move.DOWN, () -> myLocation.transform(0, 1),
            Move.LEFT, () -> myLocation.transform(-1, 0),
            Move.RIGHT, () -> myLocation.transform(1, 0));
    }
}
```

Move Validators use `BooleanSupplier`, a functional interface with a single method `boolean getAsBoolean()`. Each strategy is a method reference pointing to a validation method. For example, `Move.UP` maps to `this::isUpValid`, which checks if upward movement is valid from the current position.

Move Transformations use `Supplier<Point>`, a functional interface with a single method `T get()`. Each strategy is a lambda expression that calculates the new location based on the move direction. For example, `Move.UP` maps to `() -> myLocation.transform(0, -1)`, which creates a new point one unit above the current location.

4.2 Using the Strategies

When a move is requested, the context looks up the appropriate strategies from the maps and executes them:

```
@Override
public void move(final Move theMove) {
    // Look up and execute the validation strategy
    if (myMoveValidators.get(theMove).getAsBoolean()) {
        // Look up and execute the movement transformation strategy
        myLocation = myMovements.get(theMove).get();

        // Notify observers of successful move
        final GameEvent event = new GameEvent.MoveEvent(
            theMove,
            myLocation,
            calculateValidDirections(),
            GameEvent.now()
        );
        myPcs.firePropertyChange(event.getPropertyName(), null, event);
    } else {
        // Notify observers of invalid move attempt
        final GameEvent event = new GameEvent.InvalidMoveEvent(
            myLocation,
            theMove,
            GameEvent.now()
        );
        myPcs.firePropertyChange(event.getPropertyName(), null, event);
    }
}
```

Notice there is no conditional logic based on the move direction. The same code handles all four directions--the behavior changes based on which strategy is retrieved from the map.

4.3 The Strategy Implementations

Each validation strategy is a simple boolean method:

```
/**
 * Checks if moving up is valid from the current location.
 *
 * @return true if the piece can move up (y > 0), false otherwise
 */
private boolean isUpValid() {
    return myLocation.y() > 0;
}

/**
 * Checks if moving down is valid from the current location.
 *
 * @return true if the piece can move down (y < HEIGHT - 1), false otherwise
 */
private boolean isDownValid() {
    return myLocation.y() < HEIGHT - 1;
}

/**
 * Checks if moving left is valid from the current location.
```

```
     *
     * @return true if the piece can move left (x > 0), false otherwise
     */
    private boolean isLeftValid() {
        return myLocation.x() > 0;
    }

    /**
     * Checks if moving right is valid from the current location.
     *
     * @return true if the piece can move right (x < WIDTH - 1), false otherwise
     */
    private boolean isRightValid() {
        return myLocation.x() < WIDTH - 1;
    }
```

Each method encapsulates a specific validation algorithm. The methods are referenced, not called, when the strategy maps are initialized. They are invoked dynamically when `myMoveValidators.get(theMove).getAsBoolean()` is executed.

---

5 Contrast with the Classic Approach

In the classic OOP approach, we would define a `MoveValidator` interface and create four concrete strategy classes:

```
// The Strategy interface
public interface MoveValidator {
    boolean isValid(Point location);
}

// Concrete Strategy implementations
public class UpValidator implements MoveValidator {
    public boolean isValid(Point location) {
        return location.y() > 0;
    }
}

public class DownValidator implements MoveValidator {
    public boolean isValid(Point location) {
        return location.y() < Game.HEIGHT - 1;
    }
}

// ... and so on for LEFT and RIGHT

// Context uses the strategies
public class Game {
    private Map<Move, MoveValidator> validators;

    private Game() {
```

```
        validators = Map.of(
            Move.UP, new UpValidator(),
            Move.DOWN, new DownValidator(),
            Move.LEFT, new LeftValidator(),
            Move.RIGHT, new RightValidator()
        );
    }

    public void move(Move theMove) {
        if (validators.get(theMove).isValid(myLocation)) {
            // ... execute move
        }
    }
}
```

This works, but requires:

- Defining a `MoveValidator` interface
- Creating four separate strategy classes (one per direction)
- Instantiating four strategy objects
- More boilerplate code overall

The functional approach with method references is more concise--we get the same behavior with less ceremony. The tradeoff is that strategies must match a standard functional interface signature. For simple strategies like ours (boolean validation, coordinate transformation), this is perfect. For complex strategies that need multiple methods or maintain internal state, the classic class-based approach may be more appropriate.

---

6 Benefits and Tradeoffs

6.1 Benefits

- **Eliminates conditional logic**: No if-else chains or switch statements for algorithm selection; the map lookup handles it
- **Open/Closed Principle**: You can add new strategies without modifying existing code--just add a new entry to the strategy map
- **Runtime algorithm selection**: Strategies can be swapped at runtime by changing the map contents
- **Easy to test**: Each strategy is isolated and can be tested independently; mock strategies can be injected for testing
- **Reusable algorithms**: Strategies can be shared across different contexts

- **Cleaner code organization**: Related algorithms are grouped together; the complexity of each algorithm is hidden from the context

6.2 Tradeoffs

- **Overkill for simple cases**: If you only have one or two algorithms that never change, Strategy adds unnecessary complexity
- **Strategy proliferation**: Each algorithm requires its own implementation, which can lead to many small classes (in the classic approach) or many small methods (in the functional approach)
- **Clients must understand strategies**: Code that creates the context must know which strategy to use and when
- **Communication overhead**: If strategies need significant data from the context, you may need to pass many parameters or provide context access, creating coupling
- **Potential performance cost**: Map lookups and indirect method calls have a small overhead compared to direct calls (though this is negligible in most applications)

---

7 Common Pitfalls

⚠ Using Strategy When Simple Conditionals Suffice

Not every if-else statement needs to be replaced with Strategy. If you have two simple branches that will never change, a conditional is clearer than introducing strategy infrastructure. Strategy shines when you have multiple algorithms, they may change independently, or you need to swap them at runtime.

⚠ Stateful Strategies and Side Effects

If strategies maintain mutable state, you must be careful about reusing them across multiple contexts or concurrent executions. Our codebase avoids this by using stateless method references--each validation and transformation strategy is a pure function of its inputs.

⚠ Forgetting to Handle Missing Strategies

When using map-based strategies, what happens if you call `map.get(key)` for a key that does not exist? Our code uses `Map.of()` which creates an immutable map with all keys present, so this cannot happen. If you use a mutable map or allow dynamic strategy registration, you must handle the case where `get()` returns null.

⚠ Tight Coupling Between Context and Strategy

If a strategy needs to access many fields or methods from the context, you have created tight coupling. This often indicates the responsibility split between context and strategy is wrong. Our strategies avoid this by operating only on data passed to them (`myLocation`) or using public constants (`Game.WIDTH`, `Game.HEIGHT`).

⚠ Overusing Functional Interfaces for Complex Strategies

While method references and lambdas are concise for simple strategies, complex algorithms with multiple steps, error handling, or internal state are clearer as dedicated classes. Do not force everything into a lambda just because you can.

---

8 Related Patterns

State: Strategy and State have similar structures--both use composition to delegate behavior to interchangeable objects. The key difference is *intent*: Strategy encapsulates alternative algorithms for doing the same task (different routing algorithms all calculate routes), while State encapsulates state-specific behavior (a TCP connection behaves differently when open vs. closed). Strategy strategies are typically chosen by the client, while State transitions are often triggered by the object itself.

Observer: Our codebase combines Strategy with Observer--the `Game` class uses Strategy for move validation and transformation, then uses Observer to notify interested parties of the result. See the Observer Pattern guide. The patterns are complementary: Strategy handles *how* to perform operations, Observer handles *who* to notify about changes.

Template Method: Both patterns deal with varying algorithms, but Template Method uses inheritance (a superclass defines the algorithm skeleton, subclasses override specific steps) while Strategy uses composition (the context delegates to a strategy object). Prefer Strategy

for runtime flexibility and to avoid inheritance hierarchies; use Template Method when the algorithm structure is stable and you are varying implementation details.

Factory Method: Factories are often used to create strategy objects in the classic OOP approach. When the client needs to create a context with a specific strategy, a factory can encapsulate the creation logic, selecting the appropriate strategy based on configuration or runtime conditions.

Model-View-Controller (MVC): The Controller often uses Strategy to handle different user actions. Our `Game` class (the Model) uses Strategy internally, while the Controller uses Observer to react to game events. See the MVC Pattern guide.

> **Gen AI & Learning: Strategy Pattern and AI-Assisted Refactoring**
>
> The Strategy pattern is one of the most common refactoring targets when working with AI coding assistants. If you describe a method full of if-else branches to an AI tool, it will often suggest extracting each branch into a strategy. Understanding the pattern helps you evaluate whether that suggestion is appropriate--remember, not every conditional warrants a full Strategy refactoring. Knowing the pattern vocabulary also lets you give precise instructions: "Extract these conditionals into a map-based Strategy using method references" produces far better AI output than "clean up this code."

Summary

Concept	Key Point
Strategy Pattern	Encapsulates interchangeable algorithms behind a common interface, eliminating conditional selection logic
Three Roles	Strategy (interface), Concrete Strategies (implementations), Context (delegates to strategy)
Classic Approach	Interface + concrete classes; more boilerplate but supports complex, stateful strategies
Functional Approach	Lambdas and method references as strategies; concise for simple algorithms
Map-Based Dispatch	Associates keys (e.g., enum values) with strategy implementations in a `Map` for direct lookup

Concept	Key Point
Open/Closed Principle	Add new strategies without modifying existing code--just add a map entry
Key Tradeoff	Overkill for simple cases; best when you have multiple algorithms that may change independently
Related Patterns	State (similar structure, different intent), Observer (complementary), Template Method (inheritance alternative)

Further Reading

External Resources

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Chapter 5, pages 315--323.
- Freeman, E., & Robson, E. (2020). *Head First Design Patterns* (2nd ed.). O'Reilly. Chapter 1 provides an accessible introduction with a duck simulator example.
- Java Functional Interfaces (`java.util.function`) -- Official documentation for `BooleanSupplier`, `Supplier`, and other functional interfaces used in modern Strategy implementations.

References

Primary Texts:

- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Item 42: "Prefer lambdas to anonymous classes"--discusses using lambdas for strategy implementations.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Chapter 5: Strategy pattern.
- Freeman, E., & Robson, E. (2020). *Head First Design Patterns* (2nd ed.). O'Reilly. Chapter 1: Strategy Pattern.
- Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code* (2nd ed.). Addison-Wesley. Chapter 10: "Replace Conditional with Polymorphism."
- Martin, R. C. (2008). *Clean Code*. Prentice Hall. Chapter 6: "Objects and Data Structures."

Language Documentation:

- Oracle JDK 25: `BooleanSupplier` -- Functional interface for boolean-valued strategies
- Oracle JDK 25: `Supplier<T>` -- Functional interface for value-producing strategies
- Oracle JDK 25: `Function<T,R>` -- Functional interface for transforming strategies
- Oracle JDK 25: `Map.of()` -- Immutable map factory method used for strategy maps

---

This guide is part of TCSS 305 Programming Practicum, School of Engineering and Technology, University of Washington Tacoma.