Template Method Design Pattern - Complete Guide

🎯 Pattern Overview

The Template Method Pattern is a behavioral design pattern that defines the skeleton of an algorithm in a superclass, but lets subclasses override specific steps of the algorithm without changing its structure.

In simple terms: Define the structure of an algorithm once, and let subclasses customize specific parts!

🧩 Core Components

1. Abstract Class (Template)

- Defines the template method that outlines the algorithm structure
- Contains both concrete methods (common behavior) and abstract methods (customizable steps)
- Controls the overall flow and calls hook methods at appropriate times
- Implements the "Don't call us, we'll call you" principle (Hollywood Principle)

2. Concrete Class

- Extends the abstract class
- Implements the abstract methods to provide specific behavior
- Cannot change the overall algorithm structure
- May override hook methods to customize behavior

3. Template Method

- The main method that defines the algorithm skeleton
- Usually marked as `final` to prevent overriding
- Calls primitive operations in a specific order
- Controls the flow of execution

4. Primitive Operations

- **Abstract methods**: Must be implemented by subclasses
- **Hook methods**: Optional methods with default (often empty) implementations
- **Concrete methods**: Implemented in the abstract class, used by all subclasses

🏕️ Real-Life Example: Coffee and Tea Preparation

Both beverages follow a similar preparation process with some variations!

Coffee Preparation:
1. Boil water ✓
2. Brew coffee grounds
3. Pour in cup ✓
4. Add sugar and milk

Tea Preparation:
1. Boil water ✓
2. Steep tea bag
3. Pour in cup ✓
4. Add lemon

🔄 Working Flow

1. **Client Call**: Client calls the template method on a concrete class
2. **Algorithm Start**: Template method begins executing the predefined algorithm
3. **Concrete Steps**: Calls concrete methods implemented in the abstract class
4. **Abstract Steps**: Calls abstract methods implemented in the concrete subclass
5. **Hook Steps**: Optionally calls hook methods if overridden by subclass
6. **Algorithm End**: Template method completes the algorithm and returns result

✅ Advantages

1. Code Reuse

- Common algorithm structure is defined once in the superclass
- Eliminates code duplication across similar algorithms
- Promotes DRY (Don't Repeat Yourself) principle

2. Control Structure

- Superclass controls the overall algorithm flow
- Subclasses cannot alter the sequence of operations
- Ensures consistent behavior across implementations

3. Flexibility

- Subclasses can customize specific steps without affecting others
- Hook methods provide optional customization points
- Easy to add new variations by creating new subclasses

4. Open/Closed Principle

- Open for extension (new subclasses)
- Closed for modification (template method structure)
- Follows SOLID design principles

5. Hollywood Principle

- "Don't call us, we'll call you"
- Framework controls the flow and calls subclass methods when needed
- Reduces coupling between high-level and low-level modules

❌ Disadvantages

1. Inheritance Dependency

- Relies heavily on inheritance rather than composition
- Can lead to deep inheritance hierarchies
- Violates "favor composition over inheritance" principle

2. Rigid Structure

- Algorithm structure is fixed in the template method
- Difficult to change the sequence of operations
- May not be suitable for highly variable algorithms

3. Complexity Growth

- Can become complex with many steps and variations
- Difficult to understand the full algorithm flow
- May lead to the "yo-yo problem" (jumping between classes)

4. Limited Flexibility

- Template method usually marked as final
- Cannot dynamically change algorithm structure at runtime
- All variations must be known at compile time

🛠️ Implementation Example

```java
```

```java
import java.util.*;

// Abstract Class - Template
abstract class BeverageTemplate {

    // Template Method - defines the algorithm skeleton
    public final void prepareBeverage() {
        boilWater();
        brew();
        pourInCup();
        if (customerWantsCondiments()) {
            addCondiments();
        }
        serve();
    }

    // Concrete methods - same for all beverages
    private void boilWater() {
        System.out.println("Boiling water...");
    }

    private void pourInCup() {
        System.out.println("Pouring into cup...");
    }

    private void serve() {
        System.out.println("Your beverage is ready! Enjoy!");
        System.out.println("---");
    }

    // Abstract methods - must be implemented by subclasses
    protected abstract void brew();
    protected abstract void addCondiments();

    // Hook method - optional customization point
    protected boolean customerWantsCondiments() {
        return true; // Default behavior
    }

    // Hook method for cleanup
    protected void cleanup() {
        System.out.println("Cleaning up...");
    }
}

// Concrete Class - Coffee
```

```
class Coffee extends BeverageTemplate {
    private boolean wantsMilk;
    private boolean wantsSugar;

    public Coffee(boolean wantsMilk, boolean wantsSugar) {
        this.wantsMilk = wantsMilk;
        this.wantsSugar = wantsSugar;
    }

    @Override
    protected void brew() {
        System.out.println("Dripping coffee through filter...");
    }

    @Override
    protected void addCondiments() {
        List<String> condiments = new ArrayList<>();
        if (wantsSugar) {
            condiments.add("sugar");
        }
        if (wantsMilk) {
            condiments.add("milk");
        }

        if (!condiments.isEmpty()) {
            System.out.println("Adding " + String.join(" and ", condiments));
        }
    }

    @Override
    protected boolean customerWantsCondiments() {
        return wantsMilk || wantsSugar;
    }

    @Override
    protected void cleanup() {
        System.out.println("Cleaning coffee machine...");
    }
}

// Concrete Class - Tea
class Tea extends BeverageTemplate {
    private String teaType;
    private boolean wantsLemon;
    private boolean wantsHoney;

    public Tea(String teaType, boolean wantsLemon, boolean wantsHoney) {
```

```
        this.teaType = teaType;
        this.wantsLemon = wantsLemon;
        this.wantsHoney = wantsHoney;
    }

    @Override
    protected void brew() {
        System.out.println("Steeping " + teaType + " tea for 3-5 minutes...");
    }

    @Override
    protected void addCondiments() {
        List<String> condiments = new ArrayList<>();
        if (wantsLemon) {
            condiments.add("lemon slice");
        }
        if (wantsHoney) {
            condiments.add("honey");
        }

        if (!condiments.isEmpty()) {
            System.out.println("Adding " + String.join(" and ", condiments));
        }
    }

    @Override
    protected boolean customerWantsCondiments() {
        return wantsLemon || wantsHoney;
    }
}

// Concrete Class - Hot Chocolate
class HotChocolate extends BeverageTemplate {
    private boolean wantsMarshmallows;
    private boolean wantsWhippedCream;

    public HotChocolate(boolean wantsMarshmallows, boolean wantsWhippedCream) {
        this.wantsMarshmallows = wantsMarshmallows;
        this.wantsWhippedCream = wantsWhippedCream;
    }

    @Override
    protected void brew() {
        System.out.println("Mixing cocoa powder with hot water...");
    }

    @Override
```

```
    protected void addCondiments() {
        List<String> toppings = new ArrayList<>();
        if (wantsMarshmallows) {
            toppings.add("marshmallows");
        }
        if (wantsWhippedCream) {
            toppings.add("whipped cream");
        }

        if (!toppings.isEmpty()) {
            System.out.println("Adding " + String.join(" and ", toppings));
        }
    }

    @Override
    protected boolean customerWantsCondiments() {
        return wantsMarshmallows || wantsWhippedCream;
    }
}

// Usage Example
public class TemplateMethodDemo {
    public static void main(String[] args) {
        System.out.println("=== Template Method Pattern Demo: Beverage Preparation ===\n");

        // Prepare different beverages
        System.out.println("Preparing Coffee:");
        BeverageTemplate coffee = new Coffee(true, true);
        coffee.prepareBeverage();

        System.out.println("Preparing Tea:");
        BeverageTemplate tea = new Tea("Green", true, false);
        tea.prepareBeverage();

        System.out.println("Preparing Hot Chocolate:");
        BeverageTemplate hotChocolate = new HotChocolate(true, true);
        hotChocolate.prepareBeverage();

        // Demonstrate hook method customization
        System.out.println("Preparing Plain Coffee (no condiments):");
        BeverageTemplate plainCoffee = new Coffee(false, false);
        plainCoffee.prepareBeverage();
    }
}
```

🔧 Advanced Features

1. Multiple Template Methods

java

```
abstract class DataProcessor {

    // Main template method
    public final ProcessResult processData(String inputData) {
        try {
            validateInput(inputData);
            String processedData = transformData(inputData);
            String result = applyBusinessLogic(processedData);
            saveResult(result);
            return new ProcessResult(true, result);
        } catch (Exception e) {
            handleError(e);
            return new ProcessResult(false, e.getMessage());
        }
    }

    // Secondary template method
    public final void generateReport(ProcessResult result) {
        if (result.isSuccess()) {
            prepareReportData(result.getData());
            formatReport();
            if (shouldEmailReport()) {
                emailReport();
            }
            if (shouldSaveReport()) {
                saveReport();
            }
        }
    }

    // Abstract methods
    protected abstract void validateInput(String input);
    protected abstract String transformData(String data);
    protected abstract String applyBusinessLogic(String data);

    // Concrete methods
    protected void saveResult(String result) {
        System.out.println("Saving result to database: " + result);
    }

    protected void handleError(Exception e) {
        System.err.println("Error occurred: " + e.getMessage());
    }

    // Hook methods
    protected boolean shouldEmailReport() { return false; }
```

```
    protected boolean shouldSaveReport() { return true; }
    protected void prepareReportData(String data) { /* Default implementation */ }
    protected void formatReport() { /* Default implementation */ }
    protected void emailReport() { /* Default implementation */ }
    protected void saveReport() { /* Default implementation */ }
}

class ProcessResult {
    private boolean success;
    private String data;

    public ProcessResult(boolean success, String data) {
        this.success = success;
        this.data = data;
    }

    // Getters
    public boolean isSuccess() { return success; }
    public String getData() { return data; }
}
```

2. Strategy Pattern Integration

java

```
abstract class SortingTemplate<T> {

    public final void sortArray(T[] array) {
        if (array == null || array.length <= 1) {
            return;
        }

        preprocessArray(array);
        performSort(array);
        postprocessArray(array);

        if (shouldValidateResult()) {
            validateSortedArray(array);
        }
    }

    // Template methods with different strategies
    protected abstract void performSort(T[] array);
    protected abstract Comparator<T> getComparator();

    // Hook methods
    protected void preprocessArray(T[] array) {
        System.out.println("Preprocessing array of size: " + array.length);
    }

    protected void postprocessArray(T[] array) {
        System.out.println("Sorting completed!");
    }

    protected boolean shouldValidateResult() {
        return true;
    }

    protected void validateSortedArray(T[] array) {
        Comparator<T> comparator = getComparator();
        for (int i = 0; i < array.length - 1; i++) {
            if (comparator.compare(array[i], array[i + 1]) > 0) {
                throw new IllegalStateException("Array is not properly sorted!");
            }
        }
        System.out.println("Array sorting validation passed!");
    }
}

class QuickSort<T> extends SortingTemplate<T> {
    private Comparator<T> comparator;
```

```
    public QuickSort(Comparator<T> comparator) {
        this.comparator = comparator;
    }

    @Override
    protected void performSort(T[] array) {
        System.out.println("Performing QuickSort...");
        quickSort(array, 0, array.length - 1);
    }

    @Override
    protected Comparator<T> getComparator() {
        return comparator;
    }

    private void quickSort(T[] array, int low, int high) {
        if (low < high) {
            int pivotIndex = partition(array, low, high);
            quickSort(array, low, pivotIndex - 1);
            quickSort(array, pivotIndex + 1, high);
        }
    }

    private int partition(T[] array, int low, int high) {
        T pivot = array[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (comparator.compare(array[j], pivot) <= 0) {
                i++;
                swap(array, i, j);
            }
        }
        swap(array, i + 1, high);
        return i + 1;
    }

    private void swap(T[] array, int i, int j) {
        T temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}
```

3. Logging and Monitoring Integration

java

```
abstract class MonitoredTemplate {
    private static final Logger logger = LoggerFactory.getLogger(MonitoredTemplate.class);

    public final ExecutionResult executeWithMonitoring() {
        String operationName = getOperationName();
        long startTime = System.currentTimeMillis();

        logger.info("Starting operation: {}", operationName);

        try {
            preExecute();
            Object result = execute();
            postExecute(result);

            long duration = System.currentTimeMillis() - startTime;
            logger.info("Operation {} completed successfully in {} ms", operationName, duration);

            return ExecutionResult.success(result, duration);

        } catch (Exception e) {
            long duration = System.currentTimeMillis() - startTime;
            logger.error("Operation {} failed after {} ms", operationName, duration, e);

            handleFailure(e);
            return ExecutionResult.failure(e, duration);
        }
    }

    // Abstract methods
    protected abstract String getOperationName();
    protected abstract Object execute() throws Exception;

    // Hook methods
    protected void preExecute() {
        logger.debug("Pre-execution setup for: {}", getOperationName());
    }

    protected void postExecute(Object result) {
        logger.debug("Post-execution cleanup for: {}", getOperationName());
    }

    protected void handleFailure(Exception e) {
        logger.warn("Handling failure for operation: {}", getOperationName());
    }
}
```

```
class ExecutionResult {
    private boolean success;
    private Object result;
    private Exception error;
    private long duration;

    private ExecutionResult(boolean success, Object result, Exception error, long duration) {
        this.success = success;
        this.result = result;
        this.error = error;
        this.duration = duration;
    }

    public static ExecutionResult success(Object result, long duration) {
        return new ExecutionResult(true, result, null, duration);
    }

    public static ExecutionResult failure(Exception error, long duration) {
        return new ExecutionResult(false, null, error, duration);
    }

    // Getters...
}
```

🎯 When to Use Template Method Pattern

✅ Use When:

- Multiple classes share similar algorithms with minor variations
- You want to control the algorithm structure and prevent subclasses from changing it
- You have common behavior that should be factored out to avoid code duplication
- You need to provide hooks for subclasses to extend behavior at specific points
- You want to implement the Hollywood Principle ("Don't call us, we'll call you")
- The invariant parts of an algorithm should be implemented once

❌ Avoid When:

- The algorithm varies significantly between implementations
- You need to change the algorithm structure at runtime
- Composition would be more appropriate than inheritance
- The template method would have too many steps, making it complex
- Subclasses need to override the template method itself
- You have only one concrete implementation

Real-World Examples

1. Web Framework Request Processing

java

```
abstract class HttpRequestHandler {

    public final HttpResponse handleRequest(HttpRequest request) {
        try {
            if (!authenticate(request)) {
                return HttpResponse.unauthorized();
            }

            if (!authorize(request)) {
                return HttpResponse.forbidden();
            }

            validateRequest(request);
            Object data = processRequest(request);
            HttpResponse response = createResponse(data);

            logRequest(request, response);
            return response;

        } catch (ValidationException e) {
            return HttpResponse.badRequest(e.getMessage());
        } catch (Exception e) {
            logError(request, e);
            return HttpResponse.internalServerError();
        }
    }

    // Abstract methods
    protected abstract Object processRequest(HttpRequest request);

    // Concrete methods with default implementations
    protected boolean authenticate(HttpRequest request) {
        String token = request.getHeader("Authorization");
        return token != null && validateToken(token);
    }

    protected boolean authorize(HttpRequest request) {
        // Default: allow all authenticated requests
        return true;
    }

    protected void validateRequest(HttpRequest request) {
        // Default: basic validation
        if (request.getPath() == null) {
            throw new ValidationException("Path is required");
        }
```

```
    }

    protected HttpResponse createResponse(Object data) {
        return HttpResponse.ok(data);
    }

    protected void logRequest(HttpRequest request, HttpResponse response) {
        System.out.printf("Request: %s %s -> Status: %d%n",
            request.getMethod(), request.getPath(), response.getStatus());
    }

    protected void logError(HttpRequest request, Exception e) {
        System.err.printf("Error processing %s %s: %s%n",
            request.getMethod(), request.getPath(), e.getMessage());
    }

    private boolean validateToken(String token) {
        // Token validation logic
        return token.startsWith("Bearer ");
    }
}

class UserController extends HttpRequestHandler {
    private UserService userService = new UserService();

    @Override
    protected Object processRequest(HttpRequest request) {
        String userId = request.getPathParameter("id");

        switch (request.getMethod()) {
            case "GET":
                return userService.getUser(userId);
            case "POST":
                User user = request.getBody(User.class);
                return userService.createUser(user);
            case "PUT":
                User updateUser = request.getBody(User.class);
                return userService.updateUser(userId, updateUser);
            case "DELETE":
                userService.deleteUser(userId);
                return null;
            default:
                throw new IllegalArgumentException("Unsupported method: " + request.getMethod());
        }
    }

    @Override
```

```java
    protected boolean authorize(HttpRequest request) {
        // Users can only access their own data
        String requestedUserId = request.getPathParameter("id");
        String currentUserId = getCurrentUserId(request);
        return requestedUserId.equals(currentUserId) || isAdmin(request);
    }

    private String getCurrentUserId(HttpRequest request) {
        // Extract user ID from token
        return "currentUser";
    }

    private boolean isAdmin(HttpRequest request) {
        // Check if user has admin role
        return false;
    }
}
```

2. Game Development Framework

java

```
abstract class GameTemplate {
    private boolean gameRunning = true;
    private int score = 0;
    private int level = 1;

    public final void startGame() {
        initialize();

        while (gameRunning) {
            update();
            render();
            handleInput();

            if (shouldLevelUp()) {
                levelUp();
            }

            if (isGameOver()) {
                gameRunning = false;
            }

            sleep(getFrameDelay());
        }

        cleanup();
        showFinalScore();
    }

    // Abstract methods - must be implemented by specific games
    protected abstract void initialize();
    protected abstract void update();
    protected abstract void render();
    protected abstract void handleInput();
    protected abstract boolean isGameOver();

    // Hook methods - can be overridden
    protected boolean shouldLevelUp() {
        return score > level * 1000;
    }

    protected void levelUp() {
        level++;
        System.out.println("Level up! Now on level " + level);
    }

    protected int getFrameDelay() {
```

```
        return 16; // ~60 FPS
    }

    protected void cleanup() {
        System.out.println("Cleaning up game resources...");
    }

    protected void showFinalScore() {
        System.out.println("Game Over! Final Score: " + score + ", Level: " + level);
    }

    protected void sleep(int milliseconds) {
        try {
            Thread.sleep(milliseconds);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    // Utility methods for subclasses
    protected void addScore(int points) {
        score += points;
    }

    protected int getScore() {
        return score;
    }

    protected int getLevel() {
        return level;
    }
}

class SnakeGame extends GameTemplate {
    private Snake snake;
    private Food food;
    private Direction currentDirection = Direction.RIGHT;
    private Scanner scanner = new Scanner(System.in);

    @Override
    protected void initialize() {
        System.out.println("Initializing Snake Game...");
        snake = new Snake(10, 10);
        food = new Food();
        food.generateNew();
    }
```

```
@Override
protected void update() {
    snake.move(currentDirection);

    if (snake.hasEaten(food)) {
        addScore(100);
        snake.grow();
        food.generateNew();
    }
}

@Override
protected void render() {
    System.out.clear(); // Assume this clears the console
    System.out.println("Score: " + getScore() + " | Level: " + getLevel());

    // Render game board
    for (int y = 0; y < 20; y++) {
        for (int x = 0; x < 40; x++) {
            if (snake.occupies(x, y)) {
                System.out.print("O");
            } else if (food.isAt(x, y)) {
                System.out.print("*");
            } else {
                System.out.print(" ");
            }
        }
        System.out.println("|");
    }
}

@Override
protected void handleInput() {
    // Non-blocking input handling would be used in real implementation
    if (System.in.available() > 0) {
        String input = scanner.nextLine().toUpperCase();
        switch (input) {
            case "W": currentDirection = Direction.UP; break;
            case "S": currentDirection = Direction.DOWN; break;
            case "A": currentDirection = Direction.LEFT; break;
            case "D": currentDirection = Direction.RIGHT; break;
        }
    }
}

@Override
protected boolean isGameOver() {
```

```
        return snake.hasCollidedWithWalls() || snake.hasCollidedWithSelf();
    }

    @Override
    protected int getFrameDelay() {
        // Increase speed as level increases
        return Math.max(50, 200 - (getLevel() * 10));
    }
}
```

3. Testing Framework

java

```
abstract class TestTemplate {
    private List<String> failures = new ArrayList<>();
    private int testCount = 0;
    private int passCount = 0;

    public final TestResult runAllTests() {
        System.out.println("Starting test suite: " + getTestSuiteName());

        setUp();

        try {
            List<Method> testMethods = findTestMethods();

            for (Method testMethod : testMethods) {
                runSingleTest(testMethod);
            }

            return createTestResult();

        } finally {
            tearDown();
        }
    }

    private void runSingleTest(Method testMethod) {
        testCount++;
        String testName = testMethod.getName();

        System.out.print("Running " + testName + "... ");

        try {
            beforeEach();
            testMethod.invoke(this);
            afterEach();

            passCount++;
            System.out.println("PASS");

        } catch (Exception e) {
            failures.add(testName + ": " + e.getCause().getMessage());
            System.out.println("FAIL - " + e.getCause().getMessage());
        }
    }

    // Abstract methods
    protected abstract String getTestSuiteName();
```

```
    // Hook methods
    protected void setUp() {
        System.out.println("Setting up test suite...");
    }

    protected void tearDown() {
        System.out.println("Tearing down test suite...");
    }

    protected void beforeEach() {
        // Override if needed
    }

    protected void afterEach() {
        // Override if needed
    }

    // Helper methods
    protected void assertEquals(Object expected, Object actual) {
        if (!Objects.equals(expected, actual)) {
            throw new AssertionError(String.format("Expected <%s> but was <%s>", expected, actual));
        }
    }

    protected void assertTrue(boolean condition) {
        if (!condition) {
            throw new AssertionError("Expected true but was false");
        }
    }

    protected void assertFalse(boolean condition) {
        if (condition) {
            throw new AssertionError("Expected false but was true");
        }
    }

    private List<Method> findTestMethods() {
        return Arrays.stream(this.getClass().getDeclaredMethods())
                .filter(method -> method.getName().startsWith("test"))
                .collect(Collectors.toList());
    }

    private TestResult createTestResult() {
        return new TestResult(testCount, passCount, failures);
    }
}
```

```
class CalculatorTest extends TestTemplate {
    private Calculator calculator;

    @Override
    protected String getTestSuiteName() {
        return "Calculator Tests";
    }

    @Override
    protected void beforeEach() {
        calculator = new Calculator();
    }

    public void testAddition() {
        int result = calculator.add(2, 3);
        assertEquals(5, result);
    }

    public void testSubtraction() {
        int result = calculator.subtract(5, 3);
        assertEquals(2, result);
    }

    public void testDivisionByZero() {
        try {
            calculator.divide(10, 0);
            throw new AssertionError("Expected ArithmeticException");
        } catch (ArithmeticException e) {
            // Expected
        }
    }

    public void testMultiplication() {
        int result = calculator.multiply(4, 3);
        assertEquals(12, result);
    }
}
```

📊 Performance Considerations

1. Caching Template Results

java

```java
abstract class CachedTemplate<T> {
    private Map<String, CacheEntry<T>> cache = new ConcurrentHashMap<>();
    private final long cacheTimeout;

    public CachedTemplate(long cacheTimeoutMs) {
        this.cacheTimeout = cacheTimeoutMs;
    }

    public final T executeWithCaching(String cacheKey) {
        CacheEntry<T> cached = cache.get(cacheKey);

        if (cached != null && !cached.isExpired()) {
            return cached.getValue();
        }

        T result = executeTemplate();
        cache.put(cacheKey, new CacheEntry<>(result, System.currentTimeMillis() + cacheTimeout));

        return result;
    }

    protected abstract T executeTemplate();

    private static class CacheEntry<T> {
        private final T value;
        private final long expiration;

        public CacheEntry(T value, long expiration) {
            this.value = value;
            this.expiration = expiration;
        }

        public T getValue() { return value; }
        public boolean isExpired() { return System.currentTimeMillis() > expiration; }
    }
}
```

2. Asynchronous Template Execution

java

```
abstract class AsyncTemplate<T> {
    private final ExecutorService executor;

    public AsyncTemplate(ExecutorService executor) {
        this.executor = executor;
    }

    public final CompletableFuture<T> executeAsync() {
        return CompletableFuture
            .supplyAsync(() -> {
                preExecute();
                return execute();
            }, executor)
            .thenApply(result -> {
                postExecute(result);
                return result;
            })
            .exceptionally(this::handleException);
    }

    protected abstract T execute();
    protected abstract T handleException(Throwable throwable);

    protected void preExecute() {
        // Default implementation
    }

    protected void postExecute(T result) {
        // Default implementation
    }
}
```

🎓 Best Practices

1. Keep Template Methods Simple

java

```java
// Good: Clear, focused template method
public final void processOrder() {
    validateOrder();
    calculateTotal();
    applyDiscounts();
    processPayment();
    updateInventory();
    sendConfirmation();
}

// Avoid: Complex template method with too many responsibilities
public final void processEverything() {
    // 50+ lines of complex logic
}
```

2. Use Hook Methods Effectively

java

```java
abstract class ReportGenerator {

    public final String generateReport() {
        StringBuilder report = new StringBuilder();

        // Always executed
        report.append(generateHeader());
        report.append(generateBody());

        // Optional steps via hooks
        if (shouldIncludeFooter()) {
            report.append(generateFooter());
        }

        if (shouldIncludeSummary()) {
            report.append(generateSummary());
        }

        // Always executed
        formatReport(report);

        return report.toString();
    }

    // Abstract methods - must be implemented
    protected abstract String generateHeader();
    protected abstract String generateBody();

    // Hook methods - optional customization
    protected boolean shouldIncludeFooter() { return true; }
    protected boolean shouldIncludeSummary() { return false; }

    protected String generateFooter() {
        return "\n--- End of Report ---\n";
    }

    protected String generateSummary() {
        return "\n--- Summary section ---\n";
    }

    protected void formatReport(StringBuilder report) {
        // Default formatting - can be overridden
    }
}
```

3. Provide Clear Documentation

java

```java
/**
 * Template for processing financial transactions.
 *
 * Subclasses must implement:
 * - validateTransaction(): Validate transaction data
 * - calculateFees(): Calculate applicable fees
 * - executeTransaction(): Perform the actual transaction
 *
 * Optional hooks:
 * - shouldSendNotification(): Return true to send notifications (default: true)
 * - getNotificationTemplate(): Customize notification content
 * - shouldLogTransaction(): Return true to log transaction (default: true)
 */
abstract class TransactionProcessor {

    /**
     * Main template method for processing transactions.
     * This method cannot be overridden.
     *
     * @param transaction The transaction to process
     * @return TransactionResult indicating success/failure
     */
    public final TransactionResult processTransaction(Transaction transaction) {
        // Implementation...
    }
}
```

4. Handle Exceptions Properly

java

```
abstract class RobustTemplate {

    public final Result executeRobustly() {
        try {
            preExecute();
            Object result = execute();
            postExecute();
            return Result.success(result);

        } catch (ValidationException e) {
            handleValidationError(e);
            return Result.validationFailure(e.getMessage());

        } catch (BusinessException e) {
            handleBusinessError(e);
            return Result.businessFailure(e.getMessage());

        } catch (Exception e) {
            handleUnexpectedError(e);
            return Result.systemFailure("System error occurred");

        } finally {
            cleanup();
        }
    }

    // Abstract methods
    protected abstract Object execute() throws Exception;

    // Hook methods for error handling
    protected void handleValidationError(ValidationException e) {
        log.warn("Validation error: {}", e.getMessage());
    }

    protected void handleBusinessError(BusinessException e) {
        log.error("Business error: {}", e.getMessage());
    }

    protected void handleUnexpectedError(Exception e) {
        log.error("Unexpected error", e);
        // Could send alerts, create tickets, etc.
    }

    protected void cleanup() {
        // Always executed
```

```
    }
}
```

🔗 Related Patterns

1. Template Method + Strategy Pattern

java

```
abstract class DataProcessorTemplate {
    protected ProcessingStrategy strategy;

    public DataProcessorTemplate(ProcessingStrategy strategy) {
        this.strategy = strategy;
    }

    public final ProcessResult processData(String data) {
        validateInput(data);
        String processed = strategy.process(data); // Strategy
        return saveResult(processed);
    }

    protected abstract void validateInput(String data);
    protected abstract ProcessResult saveResult(String processed);
}

interface ProcessingStrategy {
    String process(String data);
}

class EncryptionProcessor extends DataProcessorTemplate {
    public EncryptionProcessor() {
        super(new AESEncryptionStrategy());
    }

    @Override
    protected void validateInput(String data) {
        if (data == null || data.length() < 8) {
            throw new ValidationException("Data too short for encryption");
        }
    }

    @Override
    protected ProcessResult saveResult(String processed) {
        // Save encrypted data
        return new ProcessResult(true, "Data encrypted and saved");
    }
}
```

2. Template Method + Factory Method

java

```
abstract class DocumentProcessorTemplate {

    public final void processDocument(String filePath) {
        Document document = createDocumentParser().parse(filePath); // Factory Method
        validateDocument(document);
        processContent(document);
        saveDocument(document);
    }

    // Factory Method
    protected abstract DocumentParser createDocumentParser();

    // Template steps
    protected abstract void validateDocument(Document document);
    protected abstract void processContent(Document document);
    protected abstract void saveDocument(Document document);
}

class PDFProcessor extends DocumentProcessorTemplate {
    @Override
    protected DocumentParser createDocumentParser() {
        return new PDFParser();
    }

    @Override
    protected void validateDocument(Document document) {
        // PDF-specific validation
    }

    @Override
    protected void processContent(Document document) {
        // PDF-specific processing
    }

    @Override
    protected void saveDocument(Document document) {
        // PDF-specific saving
    }
}
```

3. Template Method + Observer Pattern

java

```
abstract class ObservableTemplate {
    private List<TemplateObserver> observers = new ArrayList<>();

    public final void executeWithObservation() {
        notifyStart();

        try {
            execute();
            notifySuccess();
        } catch (Exception e) {
            notifyFailure(e);
            throw e;
        }
    }

    protected abstract void execute();

    public void addObserver(TemplateObserver observer) {
        observers.add(observer);
    }

    private void notifyStart() {
        observers.forEach(observer -> observer.onExecutionStart(this));
    }

    private void notifySuccess() {
        observers.forEach(observer -> observer.onExecutionSuccess(this));
    }

    private void notifyFailure(Exception e) {
        observers.forEach(observer -> observer.onExecutionFailure(this, e));
    }
}

interface TemplateObserver {
    void onExecutionStart(ObservableTemplate template);
    void onExecutionSuccess(ObservableTemplate template);
    void onExecutionFailure(ObservableTemplate template, Exception error);
}
```

🌍 Modern Applications

1. Microservices Request Processing

java

```
@RestController
abstract class BaseController {

    @RequestMapping(method = RequestMethod.POST)
    public final ResponseEntity<?> handleRequest(@RequestBody String requestBody) {

        try {
            // Template method structure
            RequestContext context = createContext();
            validateRequest(requestBody, context);

            Object result = processRequest(requestBody, context);

            if (shouldCache()) {
                cacheResult(result, context);
            }

            return ResponseEntity.ok(formatResponse(result, context));

        } catch (ValidationException e) {
            return ResponseEntity.badRequest().body(createErrorResponse(e));
        } catch (Exception e) {
            logError(e);
            return ResponseEntity.status(500).body(createErrorResponse(e));
        }
    }

    // Abstract methods for specific implementations
    protected abstract Object processRequest(String requestBody, RequestContext context);
    protected abstract void validateRequest(String requestBody, RequestContext context);

    // Hook methods
    protected boolean shouldCache() { return false; }
    protected RequestContext createContext() { return new RequestContext(); }
    protected Object formatResponse(Object result, RequestContext context) { return result; }
    protected void cacheResult(Object result, RequestContext context) { /* Default: no-op */ }
    protected void logError(Exception e) { /* Default logging */ }
    protected ErrorResponse createErrorResponse(Exception e) { /* Default error format */ }
}

@Component
class UserController extends BaseController {

    @Autowired
    private UserService userService;
```

```java
    @Override
    protected Object processRequest(String requestBody, RequestContext context) {
        CreateUserRequest request = parseJson(requestBody, CreateUserRequest.class);
        return userService.createUser(request);
    }

    @Override
    protected void validateRequest(String requestBody, RequestContext context) {
        // User-specific validation logic
        CreateUserRequest request = parseJson(requestBody, CreateUserRequest.class);

        if (request.getEmail() == null || !isValidEmail(request.getEmail())) {
            throw new ValidationException("Invalid email address");
        }

        if (request.getPassword() == null || request.getPassword().length() < 8) {
            throw new ValidationException("Password must be at least 8 characters");
        }
    }

    @Override
    protected boolean shouldCache() {
        return false; // Don't cache user creation results
    }
}
```

2. Machine Learning Pipeline

```java
```

```
abstract class MLPipelineTemplate<T, R> {

    public final MLResult<R> executePipeline(T inputData) {
        MLContext context = initializeContext();

        try {
            // Data preprocessing
            T preprocessedData = preprocessData(inputData, context);

            // Feature extraction
            FeatureVector features = extractFeatures(preprocessedData, context);

            // Model inference
            R prediction = runInference(features, context);

            // Post-processing
            R finalResult = postprocessResult(prediction, context);

            // Optional steps
            if (shouldValidateResult()) {
                validateResult(finalResult, context);
            }

            if (shouldLogPrediction()) {
                logPrediction(inputData, finalResult, context);
            }

            return MLResult.success(finalResult, context.getMetrics());

        } catch (Exception e) {
            handleError(e, context);
            return MLResult.failure(e.getMessage());
        }
    }

    // Abstract methods - must be implemented
    protected abstract T preprocessData(T inputData, MLContext context);
    protected abstract FeatureVector extractFeatures(T preprocessedData, MLContext context);
    protected abstract R runInference(FeatureVector features, MLContext context);

    // Hook methods - optional customization
    protected R postprocessResult(R prediction, MLContext context) { return prediction; }
    protected boolean shouldValidateResult() { return true; }
    protected boolean shouldLogPrediction() { return false; }
    protected void validateResult(R result, MLContext context) { /* Default: no validation */ }
    protected void logPrediction(T input, R result, MLContext context) { /* Default: no logging */ }
```

```
    protected void handleError(Exception e, MLContext context) { /* Default error handling */ }

    private MLContext initializeContext() {
        MLContext context = new MLContext();
        context.setStartTime(System.currentTimeMillis());
        return context;
    }
}

class ImageClassificationPipeline extends MLPipelineTemplate<BufferedImage, String> {

    private ImagePreprocessor preprocessor;
    private CNNModel model;

    @Override
    protected BufferedImage preprocessData(BufferedImage inputImage, MLContext context) {
        // Resize, normalize, etc.
        BufferedImage resized = preprocessor.resize(inputImage, 224, 224);
        BufferedImage normalized = preprocessor.normalize(resized);

        context.addMetric("preprocessing_time", preprocessor.getLastProcessingTime());
        return normalized;
    }

    @Override
    protected FeatureVector extractFeatures(BufferedImage preprocessedImage, MLContext context) {
        // Convert image to feature vector
        float[] pixels = preprocessor.imageToFloatArray(preprocessedImage);
        return new FeatureVector(pixels);
    }

    @Override
    protected String runInference(FeatureVector features, MLContext context) {
        long startTime = System.currentTimeMillis();
        String prediction = model.predict(features);
        long inferenceTime = System.currentTimeMillis() - startTime;

        context.addMetric("inference_time", inferenceTime);
        context.addMetric("confidence", model.getLastConfidence());

        return prediction;
    }

    @Override
    protected boolean shouldLogPrediction() {
        return true; // Log all image classifications
    }
```

```java
    @Override
    protected void validateResult(String result, MLContext context) {
        double confidence = (Double) context.getMetric("confidence");
        if (confidence < 0.7) {
            throw new ValidationException("Prediction confidence too low: " + confidence);
        }
    }

    @Override
    protected void logPrediction(BufferedImage input, String result, MLContext context) {
        System.out.printf("Image classified as: %s (confidence: %.2f)%n",
            result, context.getMetric("confidence"));
    }
}
```

3. DevOps Deployment Pipeline

java

```
abstract class DeploymentTemplate {

    public final DeploymentResult deploy(DeploymentConfig config) {
        DeploymentContext context = new DeploymentContext(config);

        try {
            // Pre-deployment phase
            validateEnvironment(context);
            backupCurrentVersion(context);

            // Deployment phase
            if (shouldRunTests()) {
                runPreDeploymentTests(context);
            }

            deployApplication(context);

            // Post-deployment phase
            runHealthChecks(context);

            if (shouldRunSmokeTests()) {
                runSmokeTests(context);
            }

            if (shouldNotifyStakeholders()) {
                notifyDeploymentSuccess(context);
            }

            return DeploymentResult.success(context);

        } catch (Exception e) {
            handleDeploymentFailure(e, context);
            return DeploymentResult.failure(e.getMessage(), context);
        }
    }

    // Abstract methods
    protected abstract void validateEnvironment(DeploymentContext context);
    protected abstract void deployApplication(DeploymentContext context);
    protected abstract void runHealthChecks(DeploymentContext context);

    // Hook methods
    protected boolean shouldRunTests() { return true; }
    protected boolean shouldRunSmokeTests() { return true; }
    protected boolean shouldNotifyStakeholders() { return false; }
```

```
    protected void backupCurrentVersion(DeploymentContext context) {
        // Default backup implementation
        System.out.println("Creating backup of current version...");
    }

    protected void runPreDeploymentTests(DeploymentContext context) {
        // Default test execution
        System.out.println("Running pre-deployment tests...");
    }

    protected void runSmokeTests(DeploymentContext context) {
        // Default smoke tests
        System.out.println("Running smoke tests...");
    }

    protected void notifyDeploymentSuccess(DeploymentContext context) {
        // Default notification
        System.out.println("Deployment completed successfully!");
    }

    protected void handleDeploymentFailure(Exception e, DeploymentContext context) {
        System.err.println("Deployment failed: " + e.getMessage());
        // Could trigger rollback, alerts, etc.
    }
}

class KubernetesDeployment extends DeploymentTemplate {

    private KubernetesClient k8sClient;
    private DockerRegistry registry;

    @Override
    protected void validateEnvironment(DeploymentContext context) {
        // Validate Kubernetes cluster
        if (!k8sClient.isClusterHealthy()) {
            throw new DeploymentException("Kubernetes cluster is not healthy");
        }

        // Check resource availability
        if (!k8sClient.hasEnoughResources(context.getConfig().getResourceRequirements())) {
            throw new DeploymentException("Insufficient cluster resources");
        }

        // Validate image exists
        String imageName = context.getConfig().getImageName();
        if (!registry.imageExists(imageName)) {
            throw new DeploymentException("Docker image not found: " + imageName);
```

```
    }
}

@Override
protected void deployApplication(DeploymentContext context) {
    DeploymentConfig config = context.getConfig();

    // Update deployment manifest
    Deployment deployment = k8sClient.getDeployment(config.getApplicationName());
    deployment.getSpec().getTemplate().getSpec().getContainers().get(0)
        .setImage(config.getImageName());

    // Apply rolling update
    k8sClient.updateDeployment(deployment);

    // Wait for rollout to complete
    k8sClient.waitForRollout(config.getApplicationName(), Duration.ofMinutes(10));
}

@Override
protected void runHealthChecks(DeploymentContext context) {
    String appName = context.getConfig().getApplicationName();

    // Check pod status
    List<Pod> pods = k8sClient.getPodsForApp(appName);
    boolean allPodsReady = pods.stream().allMatch(pod -> pod.getStatus().equals("Running"));

    if (!allPodsReady) {
        throw new DeploymentException("Not all pods are ready");
    }

    // Check application health endpoint
    String healthUrl = context.getConfig().getHealthCheckUrl();
    if (healthUrl != null && !isHealthEndpointOk(healthUrl)) {
        throw new DeploymentException("Health check endpoint failed");
    }
}

@Override
protected boolean shouldNotifyStakeholders() {
    return true; // Always notify for K8s deployments
}

@Override
protected void notifyDeploymentSuccess(DeploymentContext context) {
    // Send Slack notification, update deployment dashboard, etc.
    String message = String.format("✅ Deployment successful: %s -> %s",
```

```
            context.getConfig().getApplicationName(),
            context.getConfig().getImageName());

        notificationService.sendSlackMessage("#deployments", message);
        deploymentDashboard.updateStatus(context.getConfig().getApplicationName(), "DEPLOYED");
    }

    private boolean isHealthEndpointOk(String healthUrl) {
        // HTTP health check implementation
        return httpClient.get(healthUrl).getStatusCode() == 200;
    }
}
```

📈 Comparison with Other Patterns

Aspect	Template Method	Strategy	Command	State
Purpose	Define algorithm skeleton	Encapsulate algorithms	Encapsulate requests	Encapsulate state-dependent behavior
Structure	Inheritance-based	Composition-based	Composition-based	State machine
Flexibility	Fixed structure, variable steps	Runtime algorithm selection	Queue/undo operations	State transitions
Use Case	Similar algorithms with variations	Interchangeable algorithms	Decoupling invoker/receiver	State-dependent objects
Coupling	High (inheritance)	Low (composition)	Low (command interface)	Medium (state interface)

🎯 Summary

The Template Method Pattern is excellent for:

- **Algorithm frameworks** where the structure is stable but steps vary
- **Reducing code duplication** across similar processes
- **Enforcing a specific workflow** while allowing customization
- **Framework development** where you control the flow but allow extensions
- **Hook-based architectures** for optional behavior injection

Key benefits include code reuse, controlled extension points, and consistent algorithm structure. However, be mindful of the inheritance dependency and potential complexity as the number of steps and variations grows.

The pattern shines in scenarios like web frameworks, testing frameworks, data processing pipelines, and any domain where you have a stable process with variable implementation details.