Decorator Design Pattern

- Attaches additional responsibilities to an object dynamically.
- Provides a flexible alternative to subclassing for extending functionality.

Problem It Solves

1. Adding behavior at runtime
2. Avoiding class explosion from many feature combinations
3. Single Responsibility Principle maintenance
4. Need for flexible feature stacking

Bad Implementation (Without Decorator)

```
// Bad: Class explosion with inheritance
class NotificationService {
    void send(String message) {
        // Basic notification
    }
}

class SlackNotificationService extends NotificationService {
    @Override
    void send(String message) {
        // Slack specific logic
    }
}

class SlackNotificationWithLoggingService extends SlackNotificationService {
    @Override
    void send(String message) {
        log();
        super.send(message);
    }
}

class SlackNotificationWithLoggingAndRetryService extends SlackNotificationWithLoggingS
    @Override
    void send(String message) {
        retry(() -> super.send(message));
    }
}
```

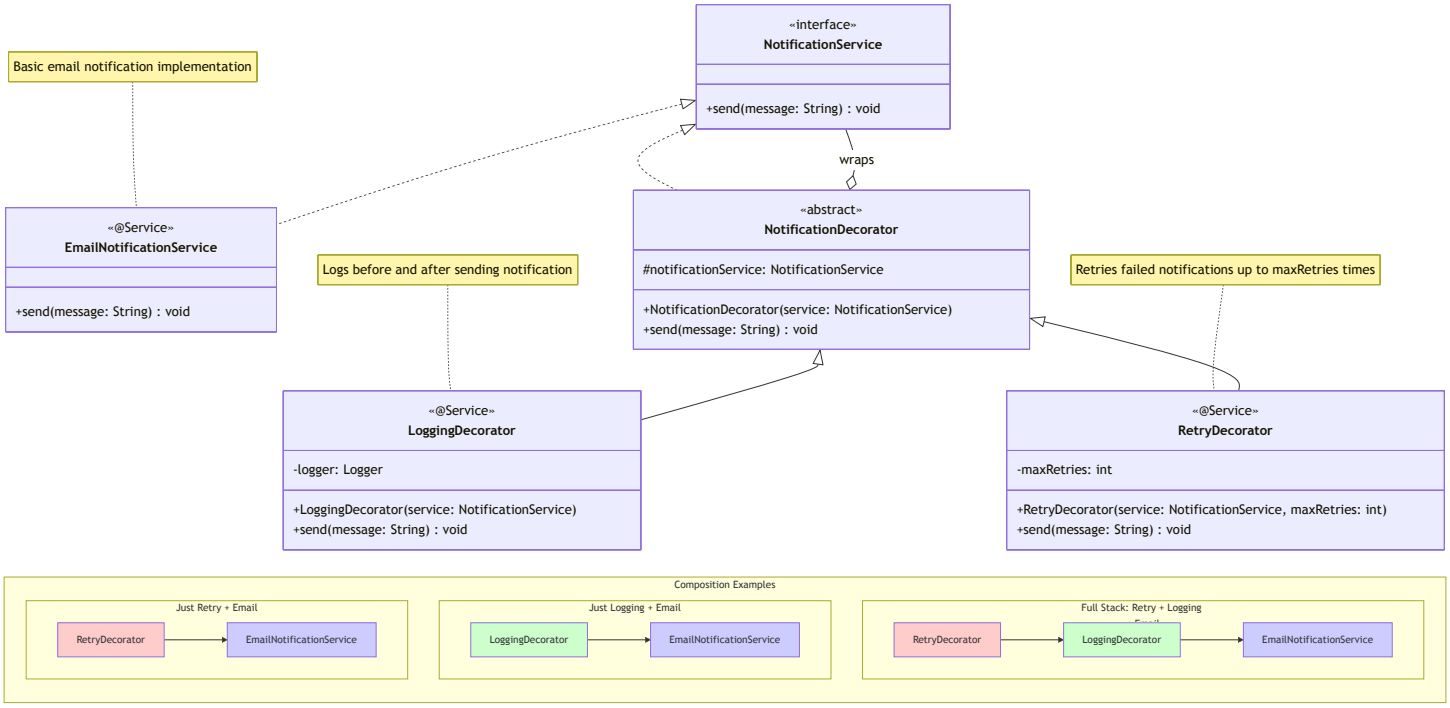
- leading to class explosion

Good Implementation

1. Basic Structure

```
// Component interface
public interface NotificationService {
    void send(String message);
}

// Concrete component
@Service
public class EmailNotificationService implements NotificationService {
    @Override
    public void send(String message) {
        // Basic email notification logic
    }
}

// Base decorator
public abstract class NotificationDecorator implements NotificationService {
    protected final NotificationService notificationService;

    public NotificationDecorator(NotificationService notificationService) {
        this.notificationService = notificationService;
    }

    @Override
    public void send(String message) {
        notificationService.send(message);
    }
}

// Concrete decorators
@Service
public class LoggingDecorator extends NotificationDecorator {
    private final Logger logger = LoggerFactory.getLogger(this.getClass());

    public LoggingDecorator(NotificationService notificationService) {
        super(notificationService);
    }

    @Override
```

```
    public void send(String message) {
        logger.info("Sending notification: {}", message);
        super.send(message);
        logger.info("Notification sent successfully");
    }
}

@Service
public class RetryDecorator extends NotificationDecorator {
    private final int maxRetries;

    public RetryDecorator(NotificationService service, int maxRetries) {
        super(service);
        this.maxRetries = maxRetries;
    }

    @Override
    public void send(String message) {
        for (int i = 0; i < maxRetries; i++) {
            try {
                super.send(message);
                return;
            } catch (Exception e) {
                if (i == maxRetries - 1) throw e;
            }
        }
    }
}

NotificationService service = new RetryDecorator(new LoggingDecorator(new EmailNotifica
service.send("Hello"); // Has both retry and logging

// Just logging
service = new LoggingDecorator(new EmailNotificationService());

// Just retry
service = new RetryDecorator(new EmailNotificationService(), 3);
```


«interface»
NotificationService
+send(message: String) : void
wraps
Basic email notification implementation
«@Service»
EmailNotificationService
+send(message: String) : void
«abstract»
NotificationDecorator
#notificationService: NotificationService
+NotificationDecorator(service: NotificationService)
+send(message: String) : void
Logs before and after sending notification
Retries failed notifications up to maxRetries times
«@Service»
LoggingDecorator
-logger: Logger
+LoggingDecorator(service: NotificationService)
+send(message: String) : void
«@Service»
RetryDecorator
-maxRetries: int
+RetryDecorator(service: NotificationService, maxRetries: int)
+send(message: String) : void
Composition Examples
Just Retry + Email
RetryDecorator
EmailNotificationService
Just Logging + Email
LoggingDecorator
EmailNotificationService
Full Stack: Retry + Logging
RetryDecorator
LoggingDecorator
EmailNotificationService

2. Real-World Example: API Response Handling

```
// Component interface
public interface ApiResponse {
    String getResponse();
}

// Concrete component
@Service
public class BasicApiResponse implements ApiResponse {
    private final String response;

    public BasicApiResponse(String response) {
        this.response = response;
    }

    @Override
    public String getResponse() {
        return response;
    }
}

// Decorators
@Service
public class CompressionDecorator extends ApiResponseDecorator {
    public CompressionDecorator(ApiResponse apiResponse) {
        super(apiResponse);
    }

    @Override
    public String getResponse() {
        return compress(super.getResponse());
    }

    private String compress(String data) {
        // Compression logic
    }
}

@Service
public class EncryptionDecorator extends ApiResponseDecorator {
    private final EncryptionService encryptionService;
```

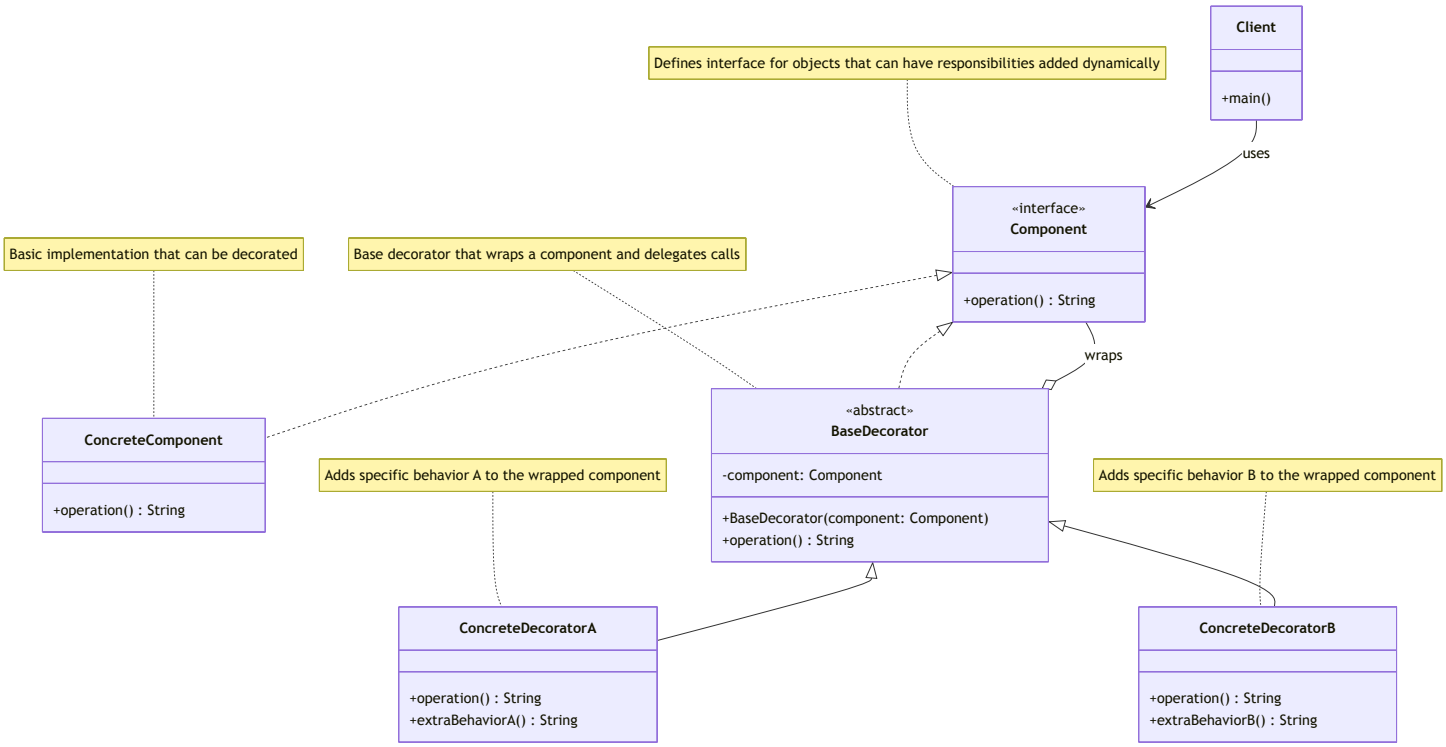
```
    public EncryptionDecorator(ApiResponse apiResponse,
                               EncryptionService encryptionService) {
        super(apiResponse);
        this.encryptionService = encryptionService;
    }

    @Override
    public String getResponse() {
        return encryptionService.encrypt(super.getResponse());
    }
}

// Usage in Spring Boot
@RestController
@RequestMapping("/api")
public class ApiController {
    private final ApiResponse apiResponse;

    public ApiController(ApiResponse basicResponse,
                         EncryptionService encryptionService) {
        this.apiResponse = new CompressionDecorator(
            new EncryptionDecorator(
                basicResponse,
                encryptionService
            )
        );
    }

    @GetMapping("/data")
    public ResponseEntity<String> getData() {
        return ResponseEntity.ok(apiResponse.getResponse());
    }
}
```

Feature Comparison Table

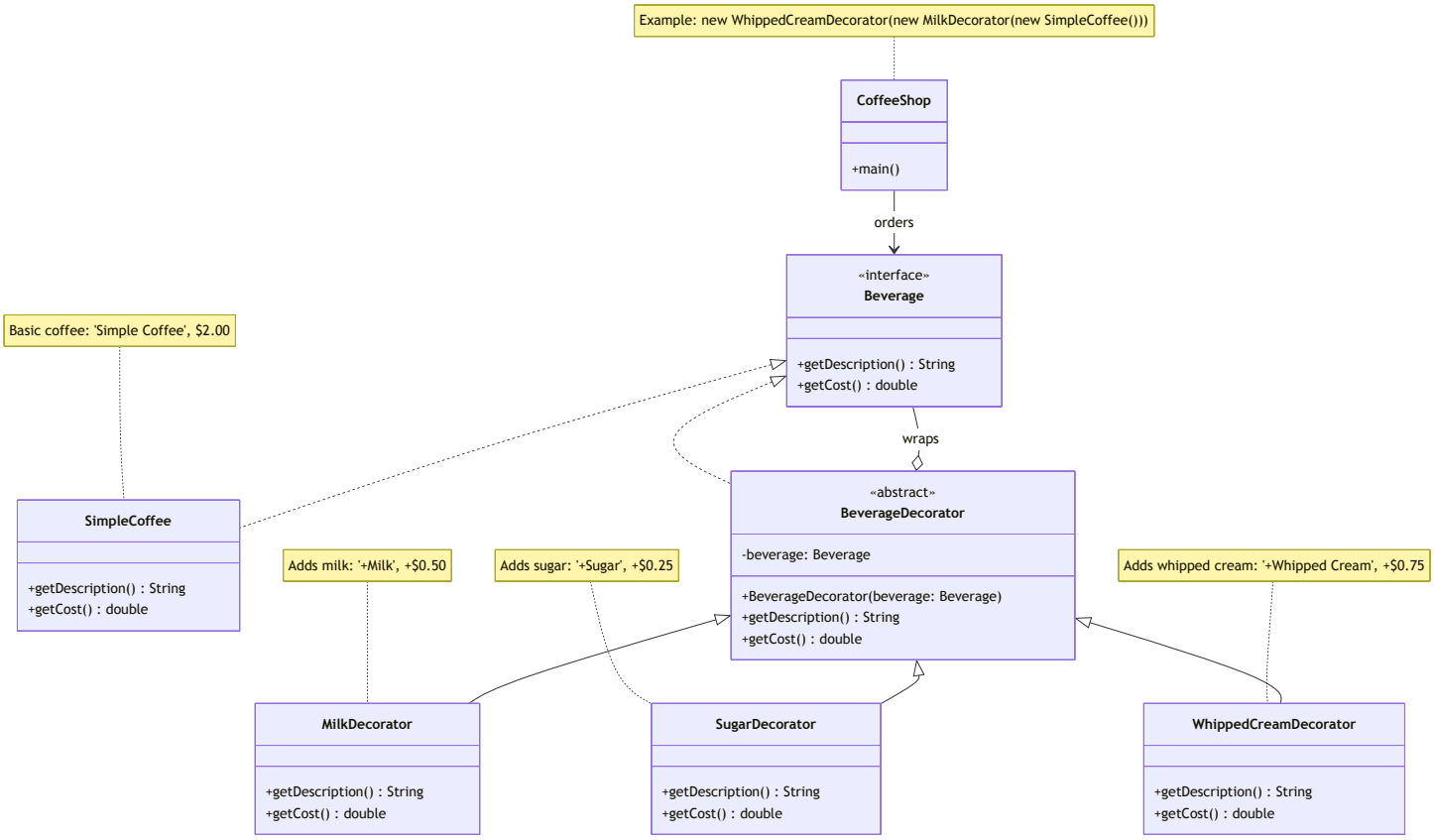
Feature	Inheritance	Decorator
Runtime Modification	❌	✅
Multiple Behaviors	Complex	Simple

Feature	Inheritance	Decorator
Code Flexibility	Low	High
Implementation Complexity	Low	Medium
Class Count	High	Medium

Architecture



Example



Example usage

```
// Basic coffee
Beverage coffee = new SimpleCoffee(); // "Simple Coffee", $2.00

// Add milk
coffee = new MilkDecorator(coffee); // "Simple Coffee + Milk", $2.50

// Add whipped cream
coffee = new WhippedCreamDecorator(coffee); // "Simple Coffee + Milk + Whipped Cream",

// Add sugar
coffee = new SugarDecorator(coffee); // "Simple Coffee + Milk + Whipped Cream + Sugar",
```