# Software Design Patterns

## Your Toolkit for Better Code

Reusable solutions to common software design problems

Instructor: Mehdi Lotfinejad

# 🎯 Learning Objectives

By the end of this lesson, you will be able to:

1. **Explain** what design patterns are and why they matter
2. **Identify** three main categories: Creational, Structural, Behavioral
3. **Recognize** common patterns and when to apply them
4. **Implement** basic patterns in your code
5. **Evaluate** when patterns are appropriate vs. over-engineering

# The Building Blocks Analogy

**Imagine building a house:**

You wouldn't invent a new way to construct a door frame or design stairs from scratch—you'd use proven architectural solutions that have worked for centuries.

**Software design patterns work the same way.**

They're tested, reusable solutions to problems developers encounter repeatedly.

# What Are Design Patterns?

## NOT Code Templates

- ❌ Copy-paste solutions
- ❌ Finished code libraries
- ❌ Language-specific syntax

## ARE Design Templates

- ✅ Conceptual blueprints
- ✅ Problem-solving approaches
- ✅ Reusable design ideas
- ✅ Shared vocabulary

**Example:** When you say "let's use a Factory pattern," you're communicating a complete solution concept in just a few words.

# Why Design Patterns Matter

**Speed up development:**

- Proven solutions vs. reinventing the wheel
- Faster problem recognition

**Improve code quality:**

- Prevent subtle bugs
- Incorporate lessons from thousands of developers

**Enhance communication:**

- Shared vocabulary among team members
- Clear design intent

**Increase maintainability:**

- Familiar structures are easier to understand
- Standardized solutions

# 📚 Pattern Categories: The Taxonomy

**Four main categories based on purpose:**

1. **Creational** - Object creation mechanisms
2. **Structural** - Object composition
3. **Behavioral** - Object communication
4. **Concurrency** - Multi-threading (advanced)

**Each category solves different types of problems**

# Pattern Categories Overview

```
Design Patterns Taxonomy
|
+-- Creational (Object Creation)
|   +-- Singleton
|   +-- Factory Method
|   +-- Abstract Factory
|   +-- Builder
|
+-- Structural (Object Composition)
|   +-- Adapter
|   +-- Decorator
|   +-- Facade
|   +-- Composite
|
+-- Behavioral (Object Interaction)
|   +-- Observer
|   +-- Strategy
|   +-- Command
|   +-- Iterator
|
+-- Concurrency (Multi-threading)
    +-- Thread Pool
    +-- Producer-Consumer
```

# 🏗️ Creational Patterns

**Purpose:** Control object creation mechanisms

**Goal:** Make systems independent of how objects are created, composed, and represented

### Key Questions Answered:

- What gets created?
- Who creates it?
- How is it created?
- When is it created?

# Pattern 1: Singleton

**Ensures a class has only ONE instance with global access**

## Use Cases:

- Database connections
- Configuration managers
- Logging systems
- Thread pools
- Cache managers

## Benefits:

✅ Single instance guaranteed
✅ Global access point
✅ Lazy initialization
✅ Resource control

## Drawbacks:

❌ Testing difficulty
❌ Hidden dependencies
❌ Threading issues

# Singleton: Implementation

```python
import threading

class DatabaseConnection:
    """
    Singleton pattern ensures only one database connection exists.
    Thread-safe implementation using class-level locking.
    """
    _instance = None
    _lock = threading.Lock()

    def __new__(cls):
        # Double-checked locking for thread safety
        if cls._instance is None:
            with cls._lock:
                if cls._instance is None:
                    cls._instance = super().__new__(cls)
                    cls._instance._initialize_connection()
        return cls._instance

    def _initialize_connection(self):
        """Initialize database connection once"""
        self.connection = self._create_db_connection()
        print("Database connection established")

    def query(self, sql):
        """Execute query using the single connection"""
        return self.connection.execute(sql)
```

# Singleton: Usage Example

```python
# Usage: Both variables reference the same instance
db1 = DatabaseConnection()
db2 = DatabaseConnection()

assert db1 is db2  # True - same object

# Only one connection created
db1.query("SELECT * FROM users")
db2.query("SELECT * FROM products")
# Both use the same underlying connection
```

**Caution:** Use Singletons sparingly!

- Makes testing difficult
- Creates hidden dependencies
- Can cause issues in multi-threaded environments

# Pattern 2: Factory Method

**Defines interface for creating objects, lets subclasses decide which class to instantiate**

## Use Cases:

- Multiple implementations
- Runtime object creation
- Plugin systems
- Framework extensions

## Benefits:

✅ Loose coupling
✅ Easy to extend
✅ Single responsibility
✅ Centralized creation

## Drawbacks:

❌ More classes
❌ Complexity increases

# Factory Method: Abstract Interface

```python
from abc import ABC, abstractmethod

class NotificationSender(ABC):
    """Abstract base class for notification senders"""

    @abstractmethod
    def send(self, message: str, recipient: str):
        """Send notification - implemented by concrete classes"""
        pass

class EmailSender(NotificationSender):
    """Concrete implementation for email notifications"""

    def send(self, message: str, recipient: str):
        print(f"Sending email to {recipient}: {message}")
        # Actual email sending logic here

class SMSSender(NotificationSender):
    """Concrete implementation for SMS notifications"""

    def send(self, message: str, recipient: str):
        print(f"Sending SMS to {recipient}: {message}")
        # Actual SMS sending logic here
```

# Factory Method: Factory Implementation

```python
class NotificationFactory:
    """
    Factory that creates appropriate notification sender.
    Centralizes object creation logic.
    """

    @staticmethod
    def create_sender(notification_type: str) -> NotificationSender:
        """
        Factory method that returns appropriate sender.
        Easy to extend with new notification types.
        """
        if notification_type == "email":
            return EmailSender()
        elif notification_type == "sms":
            return SMSSender()
        else:
            raise ValueError(f"Unknown notification type: {notification_type}")

# Usage: Client code doesn't need to know about concrete classes
def notify_user(user_preference: str, message: str, recipient: str):
    sender = NotificationFactory.create_sender(user_preference)
    sender.send(message, recipient)

notify_user("email", "Welcome!", "user@example.com")
notify_user("sms", "Your code is 1234", "+1234567890")
```

# Factory Method: Benefits

**Decoupling:**

- Client code doesn't depend on concrete classes
- Adding new types doesn't break existing code

**Centralized Logic:**

- One place to manage object creation
- Easy to modify creation rules

**Extensibility:**

- New notification types = new class + factory update
- Existing code unchanged

**Example:** Adding push notifications:

1. Create `PushSender` class
2. Update factory with new case

# 🔗 Structural Patterns

**Purpose:** Compose objects into larger structures

**Goal:** Build flexible, efficient structures while maintaining simplicity

**Key Focus:**

- How classes and objects are composed
- Relationships between entities
- Building complex structures from simple parts

# Pattern 3: Adapter

**Allows incompatible interfaces to work together**

## Use Cases:

- Third-party libraries
- Legacy code integration
- Interface standardization
- System migration

**Like a power adapter:**

US plug → European outlet

## Benefits:

✅ Reuse existing code
✅ Maintain compatibility
✅ Decouple systems
✅ Single interface

## Drawbacks:

❌ Extra abstraction layer
❌ Slight complexity

# Adapter: The Problem

```python
class LegacyPaymentProcessor:
    """
    Old payment system with different interface.
    We can't modify this code (third-party or legacy).
    """

    def process_legacy_payment(self, account_number: str, amount: float):
        print(f"Processing ${amount} from account {account_number}")
        return {"status": "success", "transaction_id": "LEG123"}

class ModernPaymentInterface(ABC):
    """
    New interface that our application expects.
    All payment processors should implement this.
    """

    @abstractmethod
    def pay(self, payment_details: dict) -> dict:
        pass

# Problem: LegacyPaymentProcessor doesn't implement ModernPaymentInterface
# How do we use it without modifying existing code?
```

# Adapter: The Solution

```python
class PaymentAdapter(ModernPaymentInterface):
    """
    Adapter makes legacy system compatible with modern interface.
    Translates between old and new method signatures.
    """

    def __init__(self, legacy_processor: LegacyPaymentProcessor):
        self.legacy_processor = legacy_processor

    def pay(self, payment_details: dict) -> dict:
        """
        Adapts modern interface to legacy system.
        Extracts data and calls old method.
        """
        account = payment_details.get("account_number")
        amount = payment_details.get("amount")

        # Call legacy method with adapted parameters
        result = self.legacy_processor.process_legacy_payment(account, amount)
        return result

# Usage: Client code uses modern interface consistently
legacy_system = LegacyPaymentProcessor()
adapted_processor = PaymentAdapter(legacy_system)

payment_info = {"account_number": "ACC123", "amount": 99.99}
result = adapted_processor.pay(payment_info)  # Works!
```

# Pattern 4: Decorator

**Adds new functionality to objects dynamically without modifying structure**

## Use Cases:

- Adding features incrementally
- I/O streams
- Middleware pipelines
- UI components

**Like pizza toppings:**

Each topping adds functionality without changing the base

## Benefits:

✅ Add features dynamically
✅ No inheritance explosion
✅ Single Responsibility
✅ Flexible combinations

## Drawbacks:

❌ Many small objects
❌ Order matters
❌ Complexity

# Decorator: Base Component

```python
class Coffee(ABC):
    """Base component interface"""

    @abstractmethod
    def cost(self) -> float:
        pass

    @abstractmethod
    def description(self) -> str:
        pass

class SimpleCoffee(Coffee):
    """Concrete component - basic coffee"""

    def cost(self) -> float:
        return 2.0

    def description(self) -> str:
        return "Simple coffee"

class CoffeeDecorator(Coffee):
    """
    Base decorator class.
    Maintains reference to wrapped component.
    """

    def __init__(self, coffee: Coffee):
        self._coffee = coffee
```

# Decorator: Concrete Decorators

```python
class MilkDecorator(CoffeeDecorator):
    """Concrete decorator that adds milk"""

    def cost(self) -> float:
        return self._coffee.cost() + 0.5

    def description(self) -> str:
        return self._coffee.description() + ", milk"

class SugarDecorator(CoffeeDecorator):
    """Concrete decorator that adds sugar"""

    def cost(self) -> float:
        return self._coffee.cost() + 0.2

    def description(self) -> str:
        return self._coffee.description() + ", sugar"

class WhipDecorator(CoffeeDecorator):
    """Concrete decorator that adds whipped cream"""

    def cost(self) -> float:
        return self._coffee.cost() + 0.7

    def description(self) -> str:
        return self._coffee.description() + ", whipped cream"
```

# Decorator: Usage Example

```
# Start with simple coffee
coffee = SimpleCoffee()
print(f"{coffee.description()}: ${coffee.cost()}")
# Output: Simple coffee: $2.0

# Add milk
coffee_with_milk = MilkDecorator(coffee)
print(f"{coffee_with_milk.description()}: ${coffee_with_milk.cost()}")
# Output: Simple coffee, milk: $2.5

# Stack multiple decorators
fancy_coffee = WhipDecorator(SugarDecorator(MilkDecorator(SimpleCoffee())))
print(f"{fancy_coffee.description()}: ${fancy_coffee.cost()}")
# Output: Simple coffee, milk, sugar, whipped cream: $3.4

# Different combinations possible
simple_sweet = SugarDecorator(SimpleCoffee())
print(f"{simple_sweet.description()}: ${simple_sweet.cost()}")
# Output: Simple coffee, sugar: $2.2
```

# Decorator: Power of Flexibility

**Without Decorator Pattern:**

- SimpleCoffee
- CoffeeWithMilk
- CoffeeWithSugar
- CoffeeWithMilkAndSugar
- CoffeeWithMilkAndWhip
- CoffeeWithSugarAndWhip
- CoffeeWithMilkSugarAndWhip
- ... 7 classes for 3 toppings!

**With Decorator Pattern:**

- SimpleCoffee (1 base)
- MilkDecorator (1 decorator)

# 🎭 Behavioral Patterns

**Purpose:** Define how objects communicate and distribute responsibilities

**Goal:** Make interactions flexible and easy to maintain

**Key Focus:**

- Communication between objects
- Assignment of responsibilities
- Algorithms and workflows
- Object collaboration

# Pattern 5: Observer

**Defines one-to-many dependency where one object's state changes notify all dependents**

## Use Cases:

- Event handling
- GUI frameworks
- Pub/Sub systems
- Reactive programming
- Real-time updates

## Benefits:

✅ Loose coupling
✅ Dynamic subscriptions
✅ Broadcast communication
✅ Event-driven

## Drawbacks:

❌ Update order uncertain
❌ Memory leaks risk
❌ Performance overhead

# Observer: Subject (Observable)

```python
from typing import List

class Subject:
    """
    Subject (Observable) maintains list of observers.
    Notifies all observers when state changes.
    """

    def __init__(self):
        self._observers: List[Observer] = []
        self._state = None

    def attach(self, observer):
        """Register an observer"""
        if observer not in self._observers:
            self._observers.append(observer)

    def detach(self, observer):
        """Unregister an observer"""
        self._observers.remove(observer)

    def notify(self):
        """Notify all observers about state change"""
        for observer in self._observers:
            observer.update(self)

    def set_state(self, state):
        """Change state and notify observers"""
        print(f"Subject: State changed to {state}")
        self._state = state
        self.notify()
```

# Observer: Observers

```
class Observer(ABC):
    """Observer interface"""

    @abstractmethod
    def update(self, subject: Subject):
        pass

class EmailNotifier(Observer):
    """Concrete observer that sends email notifications"""

    def update(self, subject: Subject):
        state = subject.get_state()
        print(f"EmailNotifier: Sending email about state: {state}")

class LoggingObserver(Observer):
    """Concrete observer that logs state changes"""

    def update(self, subject: Subject):
        state = subject.get_state()
        print(f"LoggingObserver: Logging state change: {state}")

class AnalyticsObserver(Observer):
    """Concrete observer that tracks analytics"""

    def update(self, subject: Subject):
        state = subject.get_state()
        print(f"AnalyticsObserver: Recording metric: {state}")
```

# Observer: Usage Example

```
# Create subject
order_system = Subject()

# Create and attach observers
email_notifier = EmailNotifier()
logger = LoggingObserver()
analytics = AnalyticsObserver()

order_system.attach(email_notifier)
order_system.attach(logger)
order_system.attach(analytics)

# Single state change triggers all observers
order_system.set_state("Order Placed")
# Output:
# Subject: State changed to Order Placed
# EmailNotifier: Sending email about state: Order Placed
# LoggingObserver: Logging state change: Order Placed
# AnalyticsObserver: Recording metric: Order Placed

# Can detach observers dynamically
order_system.detach(email_notifier)
order_system.set_state("Order Shipped")
# Now only logger and analytics are notified
```

# Observer: Real-World Applications

**GUI Frameworks:**

- Button click → Multiple UI components update
- Model changes → View updates automatically

**Reactive Systems:**

- Data stream changes → Subscribers react
- Price updates → Multiple displays refresh

**Event Systems:**

- User registration → Email, log, analytics, billing
- Order placement → Inventory, payment, notification

**Key Benefit:** Loose coupling between components

- Add/remove observers without modifying subject
- Subject doesn't know observer details

# Pattern 6: Strategy

**Defines family of algorithms, encapsulates each, makes them interchangeable**

## Use Cases:

- Multiple algorithms
- Payment methods
- Sorting strategies
- Compression methods
- Validation rules

## Benefits:

✅ Runtime flexibility
✅ Eliminate conditionals
✅ Easy to add strategies
✅ Isolated algorithms

## Drawbacks:

❌ More classes
❌ Client awareness
❌ Communication overhead

# Strategy: Strategy Interface

```python
class PaymentStrategy(ABC):
    """Strategy interface for payment methods"""

    @abstractmethod
    def pay(self, amount: float) -> bool:
        pass

class CreditCardPayment(PaymentStrategy):
    """Concrete strategy for credit card"""

    def __init__(self, card_number: str, cvv: str):
        self.card_number = card_number
        self.cvv = cvv

    def pay(self, amount: float) -> bool:
        print(f"Processing ${amount} via credit card ending in {self.card_number[-4:]}")
        return True

class PayPalPayment(PaymentStrategy):
    """Concrete strategy for PayPal"""

    def __init__(self, email: str):
        self.email = email

    def pay(self, amount: float) -> bool:
        print(f"Processing ${amount} via PayPal account {self.email}")
        return True
```

# Strategy: Context Class

```python
class ShoppingCart:
    """
    Context class that uses a payment strategy.
    Strategy can be changed at runtime.
    """

    def __init__(self):
        self.items = []
        self.payment_strategy = None

    def add_item(self, item: str, price: float):
        self.items.append({"item": item, "price": price})

    def set_payment_strategy(self, strategy: PaymentStrategy):
        """Change payment method dynamically"""
        self.payment_strategy = strategy

    def checkout(self):
        """Process payment using selected strategy"""
        if not self.payment_strategy:
            raise ValueError("No payment method selected")

        total = sum(item["price"] for item in self.items)
        print(f"Total amount: ${total}")

        success = self.payment_strategy.pay(total)
        if success:
            print("Payment successful!")
            self.items.clear()
        return success
```

# Strategy: Usage Example

```
# Create shopping cart
cart = ShoppingCart()
cart.add_item("Laptop", 999.99)
cart.add_item("Mouse", 29.99)

# Customer chooses credit card
cart.set_payment_strategy(CreditCardPayment("1234567890123456", "123"))
cart.checkout()
# Output:
# Total amount: $1029.98
# Processing $1029.98 via credit card ending in 3456
# Payment successful!

# Later, customer uses PayPal for another purchase
cart.add_item("Keyboard", 79.99)
cart.set_payment_strategy(PayPalPayment("user@example.com"))
cart.checkout()
# Output:
# Total amount: $79.99
# Processing $79.99 via PayPal account user@example.com
# Payment successful!
```

# Strategy: Eliminating Conditionals

## ❌ Without Strategy

```
def process_payment(method, amount):
    if method == "credit_card":
        # Credit card logic
        print(f"CC: ${amount}")
    elif method == "paypal":
        # PayPal logic
        print(f"PP: ${amount}")
    elif method == "crypto":
        # Crypto logic
        print(f"Crypto: ${amount}")
    # Add new method = modify function
```

## ✅ With Strategy

```
# Each strategy is separate class
cart.set_payment_strategy(
    CreditCardPayment("...")
)
cart.checkout()

# Add new method = new class
# No modification to existing code
# Open/Closed Principle!
```

# 🎯 Practical Example: Notification System

**Scenario:** E-commerce notification system

**Requirements:**

- Multiple channels (email, SMS, push)
- Different notification types (orders, promotions, security)
- Easy to extend
- Logging and retry
- Analytics and billing

**Solution:** Combine multiple patterns!

# Notification System: Architecture

```
PATTERNS USED:

Strategy Pattern
+-- NotificationChannel (Email, SMS, Push)
+-- Runtime channel selection

Factory Pattern
+-- ChannelFactory
+-- Creates appropriate channel

Decorator Pattern
+-- LoggingDecorator
+-- RetryDecorator
+-- Add features to channels

Observer Pattern
+-- AnalyticsObserver
+-- BillingObserver
+-- Monitor notification events

Singleton Pattern
+-- NotificationService
+-- Central coordination
```

# Notification System: Strategy + Factory

```python
# STRATEGY: Notification channels
class NotificationChannel(ABC):
    @abstractmethod
    def send(self, recipient: str, message: str, metadata: dict) -> bool:
        pass

class EmailChannel(NotificationChannel):
    def send(self, recipient: str, message: str, metadata: dict) -> bool:
        print(f"📧 Email to {recipient}: {message[:50]}")
        return True

class SMSChannel(NotificationChannel):
    def send(self, recipient: str, message: str, metadata: dict) -> bool:
        print(f"📱 SMS to {recipient}: {message[:160]}")
        return True

# FACTORY: Create channels
class ChannelFactory:
    @staticmethod
    def create_channel(channel_type: str) -> NotificationChannel:
        channels = {"email": EmailChannel, "sms": SMSChannel}
        return channels[channel_type.lower()]()
```

# Notification System: Decorator

```python
# DECORATOR: Add features to channels
class LoggingDecorator(NotificationChannel):
    """Decorator that logs all attempts"""

    def __init__(self, channel: NotificationChannel):
        self._channel = channel

    def send(self, recipient: str, message: str, metadata: dict) -> bool:
        print(f"[LOG] Attempting send to {recipient}")
        result = self._channel.send(recipient, message, metadata)
        print(f"[LOG] Result: {'SUCCESS' if result else 'FAILED'}")
        return result

class RetryDecorator(NotificationChannel):
    """Decorator that retries failures"""

    def __init__(self, channel: NotificationChannel, max_retries: int = 3):
        self._channel = channel
        self.max_retries = max_retries

    def send(self, recipient: str, message: str, metadata: dict) -> bool:
        for attempt in range(self.max_retries):
            try:
                if self._channel.send(recipient, message, metadata):
                    return True
                print(f"   Retry {attempt + 1}/{self.max_retries}")
            except Exception as e:
                print(f"   Attempt {attempt + 1} failed: {e}")
        return False
```

# Notification System: Observer

```
# OBSERVER: Monitor events
class NotificationObserver(ABC):
    @abstractmethod
    def on_notification_sent(self, event_data: dict):
        pass

class AnalyticsObserver(NotificationObserver):
    """Track metrics"""

    def on_notification_sent(self, event_data: dict):
        print(f"📊 Analytics: Recorded notification")
        print(f"   Channel: {event_data['channel']}")
        print(f"   Type: {event_data['type']}")

class BillingObserver(NotificationObserver):
    """Track costs"""

    def on_notification_sent(self, event_data: dict):
        costs = {"email": 0.001, "sms": 0.05, "push": 0.0}
        cost = costs.get(event_data['channel'], 0)
        print(f"💰 Billing: Charged ${cost}")
```

# Notification System: Singleton Service

```python
# SINGLETON: Central service
class NotificationService:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance._observers = []
        return cls._instance

    def attach_observer(self, observer: NotificationObserver):
        self._observers.append(observer)

    def send_notification(self, recipient: str, message: str,
                          channel_type: str, notification_type: str = "general"):
        # Factory: Create channel
        base_channel = ChannelFactory.create_channel(channel_type)

        # Decorator: Add features
        enhanced = RetryDecorator(LoggingDecorator(base_channel))

        # Strategy: Execute
        success = enhanced.send(recipient, message, {})

        # Observer: Notify
        if success:
            for obs in self._observers:
                obs.on_notification_sent({
                    "channel": channel_type, "type": notification_type
                })
```

# Notification System: Usage

```
# Get singleton service
service = NotificationService()

# Attach observers
service.attach_observer(AnalyticsObserver())
service.attach_observer(BillingObserver())

# Send notifications
service.send_notification(
    recipient="customer@example.com",
    message="Your order has been confirmed",
    channel_type="email",
    notification_type="order_confirmation"
)

service.send_notification(
    recipient="+1234567890",
    message="Security alert: New login detected",
    channel_type="sms",
    notification_type="security_alert"
)
```

# Notification System: Benefits

**Pattern Synergy:**

**Factory** creates channels → Easy to add new channels

**Strategy** selects channel → Runtime flexibility

**Decorator** adds features → Logging, retry without modifying channels

**Observer** monitors events → Analytics, billing decoupled

**Singleton** coordinates → Central service management

**Result:**

- Easy to extend (new channels, decorators, observers)
- Each pattern solves specific problem
- Loose coupling throughout
- Testable components
- Production-ready architecture

# 🚫 Common Pitfalls

## Pitfall #1: Over-Engineering with Patterns

**Problem:** Using patterns when not needed

**Example:**

- Factory for single concrete class
- Singleton for non-shared resources
- Strategy for 2 simple conditionals

**Impact:**

- Unnecessary complexity
- Harder to understand
- More code to maintain

**Solution:** "Rule of Three"



- Wait for 3+ instances before introducing pattern

# Pitfall #2: Singleton Abuse

**Problem:** Overusing Singletons

**Why it's tempting:**

- Global access to anything
- Seems convenient
- Easy to implement

**Why it's bad:**

- Hidden dependencies
- Testing nightmare (can't mock)
- Threading issues
- Global state problems

**Solution:**

- Use only for truly shared resources (config, logs, pools)

# Pitfall #3: Ignoring Pattern Context

**Problem:** Using patterns in wrong context

**Examples:**

- Observer for synchronous operations
- Strategy for simple if/else
- Factory for objects created once
- Decorator when inheritance works fine

**Solution:**

- Study "When to Use" sections
- Understand pattern applicability
- Consider simpler alternatives first
- Pattern should simplify, not complicate

# Pitfall #4: Mixing Pattern Responsibilities

**Problem:** Hybrid patterns doing too much

**Examples:**

- Singleton + Factory in one class
- Decorator + Adapter combined
- Observer + Strategy mixed

**Why it's bad:**

- Violates Single Responsibility
- Confusing to understand
- Hard to maintain

**Solution:**

- Keep patterns focused
- Compose cleanly

## ✅ Best Practices

### Best Practice #1: Document Pattern Usage

```python
class DatabaseConnection:
    """
    Singleton pattern ensures only one database connection exists.

    This prevents resource exhaustion and ensures consistent state
    across the application. Thread-safe implementation using locks.

    Pattern: Singleton
    Reason: Single shared connection pool
    Alternatives considered: Connection per request (too expensive)
    """
    pass
```

**Benefits:**

- Future developers understand design
- Documents "why" not just "what"
- Prevents well-intentioned breaking changes

# Best Practice #2: Use Pattern When It Simplifies

## ❌ Pattern Overkill

```
# Using Strategy for simple choice
class TaxCalculator:
    def __init__(self, strategy):
        self.strategy = strategy

# Could be:
def calculate_tax(amount, rate):
    return amount * rate
```

## ✅ Pattern Appropriate

```
# Multiple complex algorithms
class SortStrategy(ABC):
    def sort(self, data): pass

class QuickSort(SortStrategy): ...
class MergeSort(SortStrategy): ...
class HeapSort(SortStrategy): ...

# Runtime selection needed
# Algorithms are complex
```

# Best Practice #3: Prefer Composition

## Many design patterns are about composition:

- Decorator: Wrap objects to add behavior
- Strategy: Compose with different algorithms
- Observer: Compose subject with observers

```python
# Instead of deep inheritance:
class CoffeeWithMilkAndSugar(Coffee): pass

# Use composition:
coffee = SugarDecorator(MilkDecorator(SimpleCoffee()))
```

## Benefits:

- More flexible
- Easier to test
- Runtime changes possible
- "Favor composition over inheritance"

# Best Practice #4: Start Simple, Refactor to Patterns

**Development Flow:**

1. **Write simple code** that works
2. **Notice code smells** (duplication, rigidity)
3. **Identify pattern** that solves the problem
4. **Refactor** to pattern
5. **Test** that it still works

**Don't start with patterns!**

- Start with simplest solution
- Let patterns emerge from real needs
- Refactor when complexity justifies it

**"Make it work, make it right, make it fast"**

# 📊 Patterns vs. Alternatives

## When NOT to Use Patterns

### Use Simple Function When:

- Problem is straightforward
- No runtime flexibility needed
- Single responsibility
- Utility operations

**Example:** Date formatting

### Use Pattern When:

- Anticipate change
- Need runtime flexibility
- Multiple implementations
- Complex behavior

**Example:** Multiple payment methods

**Key:** Choose simplest solution that solves your problem

# Patterns vs. Language Features

## Many patterns built into modern languages:

| Pattern | Language Feature |
|---|---|
| Iterator | `for item in collection` |
| Decorator | `@decorator` syntax |
| Observer | Event listeners |
| Strategy | First-class functions |
| Singleton | Module imports |

**Guideline:** Use language features when available, explicit patterns when you need full structure

# Patterns vs. Frameworks

**Frameworks implement patterns internally:**

- **Django signals** → Observer pattern
- **React components** → Composite pattern
- **Express middleware** → Chain of Responsibility
- **Spring @Bean** → Factory pattern

**Guideline:**

- Use framework mechanisms when possible
- Implement patterns for framework-independent code
- Explicit patterns for library development

# 🎯 Key Takeaways

**Essential Lessons:**

1. **Patterns are proven solutions**, not code templates
2. **Three main categories**: Creational, Structural, Behavioral
3. **Patterns work together** in real applications
4. **Avoid over-engineering** - Rule of Three
5. **Context matters** - when to use, not just how
6. **Modern languages/frameworks** often have built-in support
7. **Document your usage** - explain the "why"

# Pattern Selection Guide

```
CHOOSING THE RIGHT PATTERN:

Need to control object creation?
    +-- Single instance? → Singleton
    +-- Multiple implementations? → Factory
    +-- Complex construction? → Builder

Need to compose objects?
    +-- Incompatible interfaces? → Adapter
    +-- Add features dynamically? → Decorator
    +-- Simplify complex system? → Facade

Need object communication?
    +-- One-to-many notification? → Observer
    +-- Multiple algorithms? → Strategy
    +-- Request as object? → Command

Not sure? Start simple!
+-- Write straightforward code first
+-- Refactor to pattern when needed
+-- Don't force patterns
```

# When to Introduce Patterns

**Green Light (Use Pattern):**

✅ Three or more similar implementations
✅ Anticipate significant future changes
✅ Need runtime flexibility
✅ Pattern simplifies the design
✅ Team familiar with pattern

**Red Light (Keep Simple):**

❌ Only one or two implementations
❌ Requirements are stable
❌ No need for runtime changes
❌ Pattern adds complexity
❌ Team unfamiliar with pattern

# 📝 Practice Quiz #1

## Pattern Category Question:

You need a logging system that writes to multiple destinations (file, database, cloud) simultaneously. When a log entry is created, all destinations should update automatically.

**Which pattern category and specific pattern?**

A) Creational - Factory Method
B) Structural - Adapter
C) Behavioral - Observer
D) Structural - Decorator

**Think before next slide...**

# Quiz #1: Answer

**Answer: C) Behavioral - Observer**

## Explanation:

**Why Observer:**

- One-to-many notification relationship
- Log entry creation (Subject) notifies all destinations (Observers)
- Automatic updates when event occurs
- Destinations are independent

**Why not others:**

- **Factory:** Creates destinations, doesn't handle notification
- **Adapter:** Makes incompatible interfaces work together
- **Decorator:** Adds features to single object, not multiple destinations

**Key indicator:** "all destinations should be updated automatically" = Observer

# 📝 Practice Quiz #2

## Singleton Application Question:

**Which is the BEST use case for Singleton?**

A) User class representing individual users

B) Database connection pool shared across application

C) Product class in e-commerce system

D) Utility class with static helper methods

**Think before next slide...**

# Quiz #2: Answer

**Answer: B) Database connection pool shared across application**

## Explanation:

**Why connection pool:**

- Shared resource
- Single point of coordination
- Global access needed
- Resource management

**Why not others:**

- **User class:** Need many instances (one per user)
- **Product class:** Need many instances (one per product)
- **Utility class:** Static methods don't need Singleton (no state)

**Singleton indicators:**

# 📝 Practice Quiz #3

## Pattern Combination Question:

In the notification system, we used Factory and Decorator together.

**Why use BOTH patterns?**

A) Factory creates channels, Decorator adds features like logging/retry
B) Factory and Decorator do same thing, so using both is redundant
C) Factory handles runtime, Decorator handles compile-time
D) Decorator creates channels, Factory adds features

**Think before next slide...**

# Quiz #3: Answer

**Answer: A) Factory creates channels, Decorator adds features**

## Explanation:

**Separation of concerns:**

- **Factory:** Handles object creation (Email/SMS/Push)
- **Decorator:** Adds cross-cutting concerns (logging, retry)

**Why it works:**

- Factory decides WHAT to create
- Decorator decides WHAT FEATURES to add
- Can add new channels without changing decorators
- Can add new decorators without changing channels

**Pattern synergy:**

```
# Factory creates
```

# 📝 Practice Quiz #4

## Avoiding Pitfalls Question:

You need to format dates in two ways: "MM/DD/YYYY" or "DD-MM-YYYY". Colleague suggests Strategy pattern.

**What's the BEST response?**

A) Great! Strategy perfect for multiple options
B) No, too simple for Strategy. Use function with parameter
C) Yes, but only if we add Factory pattern too
D) Strategy never appropriate for formatting

**Think before next slide...**

# Quiz #4: Answer

### Answer: B) No, too simple for Strategy. Use function with parameter

## Explanation:

### Why not Strategy:

- Only 2 simple format options
- Strategy adds unnecessary complexity
- Multiple classes, interfaces, indirection

### Better solution:

```
def format_date(date, format_type):
    if format_type == "US":
        return date.strftime("%m/%d/%Y")
    else:
        return date.strftime("%d-%m-%Y")
```

### When to use Strategy:



- Multiple COMPLEX algorithms

# 📝 Practice Quiz #5

## Pattern Selection Question:

Integrating third-party payment library: `processPayment(amount, card)`, but your app expects: `pay(payment_details)`.

### Which pattern solves this?

A) Factory Method - create compatible processors
B) Adapter - translate between incompatible interfaces
C) Decorator - add compatibility features
D) Observer - notify when payments occur

### Think before next slide...

# Quiz #5: Answer

### Answer: B) Adapter - translate between incompatible interfaces

## Explanation:

### Why Adapter:

- Two incompatible interfaces
- Third-party code (can't modify)
- Need translation layer
- Make them work together

### How it works:

```
class PaymentAdapter(YourInterface):
    def __init__(self, third_party_lib):
        self.lib = third_party_lib

    def pay(self, payment_details):
        # Translate your interface to their interface
        amount = payment_details['amount']
        card = payment_details['card']
```

# Summary: Pattern Quick Reference

| Pattern | Purpose | When to Use |
|---|---|---|
| **Singleton** | One instance | Shared resources |
| **Factory** | Object creation | Multiple implementations |
| **Adapter** | Interface translation | Incompatible interfaces |
| **Decorator** | Add features | Dynamic behavior |
| **Observer** | Event notification | One-to-many updates |
| **Strategy** | Algorithm selection | Runtime flexibility |

**Remember:** Patterns are tools, not goals. Choose simplest solution that solves your problem.

# Resources for Continued Learning

**Essential Books:**

- **Design Patterns** by Gang of Four (Gamma et al.)
- **Clean Code** by Robert Martin
- **Refactoring** by Martin Fowler
- **Head First Design Patterns** by Freeman & Freeman

**Online Resources:**

- Refactoring.Guru (refactoring.guru/design-patterns)
- SourceMaking (sourcemaking.com/design_patterns)
- Microsoft Patterns (docs.microsoft.com/patterns)

**Practice:**

- Identify patterns in frameworks you use
- Refactor existing code to patterns

# Action Items

## This Week:

1. Identify ONE pattern in code you're currently working on
2. Read about one pattern in depth
3. Implement one simple pattern in a side project
4. Discuss patterns with your team

## This Month:

- Read "Head First Design Patterns"
- Refactor existing code using appropriate patterns
- Document pattern usage in your codebase
- Share pattern knowledge with team

## Practice:

- Look for patterns in libraries you use

# Final Thoughts

**Design Patterns Are:**

- Templates, not code
- Solutions to common problems
- Shared vocabulary
- Tools in your toolkit

**Success Comes From:**

- Understanding when to use patterns
- Knowing when NOT to use patterns
- Starting simple, adding complexity when needed
- Documenting your design decisions
- Learning from real-world examples
- Practicing pattern recognition

# Thank You! 🚀

## Questions?

**Key Message:**

Design patterns are powerful tools for solving common problems, but use them judiciously. Start simple, let patterns emerge from real needs, and always choose readability over cleverness.

**Pattern wisely, code simply, document clearly.**

# Lesson Complete

## You Are Now Equipped To:

✓ Explain what design patterns are and why they matter
✓ Identify three main categories (Creational, Structural, Behavioral)
✓ Recognize common patterns in codebases
✓ Implement basic patterns in your code
✓ Evaluate when patterns help vs. when they over-engineer
✓ Combine patterns for real-world solutions
✓ Avoid common pattern pitfalls
✓ Document pattern usage effectively

**Now go build better code!**

**Keep learning, keep coding, keep simplifying!**