

# All Design Patterns

Generated from repository files.

## NullObjectPattern

Overview (README.md)

## Null Object Pattern

### What is it?

The Null Object pattern provides an object as a surrogate for the lack of an object. Instead of returning null or checking for null, use a special “null” object that does nothing.

### When to use it?

- Avoid null pointer exceptions (NullPointerException)
- Eliminate null checks throughout code
- Provide default behavior for missing objects
- Simplify client code logic

### Real-world Example

**Vehicle Factory:** Return a NullVehicle instead of null when a vehicle type is not found, avoiding null checks in client code.

### Key Benefits

✓ Eliminates null checks ✓ Prevents NullPointerException ✓ Simplifies client code ✓ Provides default behavior transparently ✓ Follows Open/Closed Principle

### Key Drawbacks

✗ Can hide bugs (silent failures) ✗ May mask incorrect behavior ✗ Adds extra classes (null objects) ✗ Requires discipline to implement correctly

### Easy Analogy

**Think of it like a backup player on a sports team:** When the main player gets injured, instead of having “no player”, you put in a backup player who can do basic things but doesn’t contribute much. Your team keeps running without checking “do we have a player?”

### Implementation Notes

- Create a NullObject class that implements the same interface
- Override methods to do nothing or return default values
- Return NullObject instead of null from factory methods
- Use composition where null object provides safe defaults

### UML / Class Diagram

## Null Object - Class Diagram



### Class Relationships

	Class	Responsibility	Depends On
Vehicle		Interface defining vehicle behavior (contract)	None
Car		Real implementation of vehicle	Implements Vehicle
NullVehicle		Null object - implements interface but does nothing	Implements Vehicle
VehicleFactory		Creates vehicles or NullVehicle instead of returning null	Returns Vehicle

### How to Code This Pattern

1. **Define Interface:** Create Vehicle interface
2. **Create Real Class:** Car implements Vehicle with actual behavior
3. **Create Null Class:** NullVehicle implements Vehicle with empty methods
4. **Update Factory:** Return NullVehicle instead of null
5. **No Null Checks:** Client code doesn’t need to check for null

## ObserverPattern

# Observer Pattern

## What is it?

The Observer pattern defines a one-to-many dependency where when one object changes state, all dependent objects are notified automatically and updated.

## When to use it?

- A change to one object requires changing unknown number of other objects
- An object should notify others without assuming who they are
- Model real-world event systems (event listeners, MVC architecture)
- Implement pub-sub systems

## Real-world Example

**E-commerce Stock System:** When product stock changes, all observers (email notifier, SMS notifier, mobile app) are notified automatically.

## Key Benefits

✓ Loose coupling between observers and subject ✓ Dynamic subscription/unsubscription ✓ One-to-many communication ✓ Supports event-driven architecture ✓ Easy to extend with new observers

## Key Drawbacks

✗ Observer notification order is not guaranteed ✗ All observers are notified even if they don't need the update ✗ Memory leaks if observers not unsubscribed properly ✗ Debugging can be difficult (implicit dependencies)

## Easy Analogy

**Think of it like YouTube subscriptions:** You (Observer) subscribe to a channel (Subject) → When the creator uploads a video (state change), YouTube notifies all subscribers automatically. If you unsubscribe, no more notifications.

## Implementation Notes

- **Subject:** Maintains list of observers, notifies on state change
- **Observer:** Interface with update() method
- **ConcreteObserver:** Implements update() to react to changes
- Use weak references to avoid memory leaks
- Consider thread-safety for concurrent observers

## UML / Class Diagram

## Observer - Class Diagram



## Class Relationships

Class	Responsibility	Depends On
StockObserver	Interface for observers receiving notifications	None
EmailNotifier	Concrete observer - sends email when notified	Implements StockObserver
SMSNotifier	Concrete observer - sends SMS when notified	Implements StockObserver
StockSubject	Subject/Observable - notifies all observers on state change	Maintains list of StockObserver

## How to Code This Pattern

1. **Create Observer Interface:** Define update() method signature
2. **Create Concrete Observers:** EmailNotifier, SMSNotifier implement update()
3. **Create Subject:** Maintain list of observers
4. **Implement attach():** Add observer to list
5. **Implement detach():** Remove observer from list
6. **Implement notifyObservers():** Call update() on all observers
7. **On State Change:** Call notifyObservers() when data changes

# Chain of Responsibility Pattern

## What is it?

The Chain of Responsibility pattern allows you to pass requests along a chain of handlers. Each handler decides whether to process the request or pass it to the next handler in the chain.

## When to use it?

- Multiple objects may handle a request, and the handler isn't known in advance
- You want to issue requests without specifying the receiver (e.g., logging at different levels)
- You need to dynamically configure the chain of handlers

## Real-world Example

**Logging System:** Different log levels (Debug → Info → Error) where each level handles its corresponding messages and passes others down the chain.

## Key Benefits

✓ Decouples sender from receiver ✓ Flexible chain configuration at runtime ✓ Single Responsibility: Each handler handles one responsibility ✓ Open/Closed Principle: Add new handlers without modifying existing ones

## Key Drawbacks

✗ Request may not be handled if chain is not properly configured ✗ Difficult to debug (request path not always clear) ✗ Performance overhead from passing through chain

## Easy Analogy

**Think of it like a complaint handling system in a company:** Your complaint goes to the frontend → If they can't solve it, they pass to manager → If manager can't solve, they pass to director. Each person (handler) decides if they can handle it or pass it on.

## Implementation Notes

- Each handler should have a reference to the next handler
- Handler should process request and decide to pass or not
- Create chain before using it

### UML / Class Diagram

## Chain of Responsibility - Class Diagram



## Class Relationships

Class	Responsibility	Depends On
LogProcessor	Abstract base - defines chain structure and template for logging	nextProcessor (self-reference)
DebugLogProcessor	Handles debug-level log messages	LogProcessor (extends)
InfoLogProcessor	Handles info-level log messages	LogProcessor (extends)
ErrorLogProcessor	Handles error-level log messages	LogProcessor (extends)

## How to Code This Pattern

1. **Create Abstract Processor:** Define LogProcessor with next processor reference
2. **Implement Processors:** Each concrete class handles specific log level
3. **Build Chain:** Connect processors in order (Debug → Info → Error)
4. **Send Request:** Processor either handles it or passes to next

## mementoPattern

Overview (README.md)

## Memento Pattern

## What is it?

The Memento pattern captures and stores an object's internal state without exposing it, allowing the object to be restored to that state later.

## When to use it?

- Implement undo/redo functionality

- Save checkpoints of an object's state
- Restore previous state without breaking encapsulation
- Maintain history of object states

## Real-world Example

**Text Editor:** Save snapshots of document at different points so users can undo/redo edits without directly accessing internal editor state.

## Key Benefits

✓ Preserves encapsulation (internal state stays hidden) ✓ Provides undo/redo functionality ✓ Doesn't violate Single Responsibility Principle ✓ Allows state restoration without side effects

## Key Drawbacks

✗ Memory overhead if too many states are stored ✗ Time overhead to create/restore snapshots ✗ More complex implementation than simple state management

## Easy Analogy

**Think of it like Google Docs version history:** Your document (Originator) saves snapshots at certain points → Each snapshot is a Memento → Google (CareTaker) keeps all versions → You can click on any version to restore it.

## Implementation Notes

- **Originator:** Object whose state we want to save
- **Memento:** Stores snapshot of originator's state
- **CareTaker:** Manages memento objects and restoration logic
- Store mementos in a list for undo/redo operations

## UML / Class Diagram

## Memento - Class Diagram



## Class Relationships

Class	Responsibility	Depends On
Originator	Creates memento snapshots of its state, can restore from memento	Creates Memento
Memento	Immutable snapshot of originator's state at a point in time	None (value object)
CareTaker	Manages collection of mementos, provides history management	Stores/retrieves Memento

## How to Code This Pattern

1. **Create Originator:** Add createMemento() and restoreFromMemento() methods
2. **Create Memento:** Immutable class storing state snapshot
3. **Create CareTaker:** Maintain list of mementos (undo/redo stack)
4. **Save State:** Call createMemento() before changes
5. **Restore State:** Call restoreFromMemento() to go back

## objectPoolPattern

Overview (README.md)

## Object Pool Pattern

### What is it?

The Object Pool pattern reuses objects that are expensive to create by maintaining a pool of initialized objects. When needed, objects are borrowed from the pool and returned after use.

### When to use it?

- Creating objects is expensive (database connections, threads)
- Frequent object creation/destruction causes performance issues
- Need controlled resource allocation
- Multiple threads need access to limited resources

## Real-world Example

**Database Connection Pool:** Maintain a pool of reusable database connections instead of creating new ones for each request. Connections are borrowed and returned to pool.

## Key Benefits

✓ Improved performance (avoid repeated object creation) ✓ Better resource management ✓ Thread-safe connection/resource management ✓ Reduces garbage collection overhead ✓ Predictable resource allocation

## Key Drawbacks

✗ Increased memory usage (maintaining pool) ✗ More complex implementation ✗ Synchronization overhead for thread-safety ✗ Risk of resource leaks if not managed properly

## Easy Analogy

**Think of it like a library lending system:** Instead of making a new book for each person, the library has a pool of books. You borrow a book (getConnection) → Use it → Return it (releaseConnection). The next person can reuse the same book.

## Implementation Notes

- Maintain available and in-use lists/queues
- Implement acquire() to borrow from pool
- Implement release() to return to pool
- Handle pool exhaustion gracefully
- Ensure proper initialization of pooled objects

## UML / Class Diagram

## Object Pool - Class Diagram



## Class Relationships

Class	Responsibility	Depends On
DBConnection	Represents a database connection that can be reused	None
DBConnectionPoolManager	Manages pool of connections, hands out and accepts returns	Creates/manages DBConnection

## How to Code This Pattern

1. **Create Pooled Class:** DBConnection with proper initialization/cleanup
2. **Create Pool Manager:** DBConnectionPoolManager with queues for available/in-use
3. **Implement getConnection():** Get from available pool or create new
4. **Implement releaseConnection():** Return connection to available pool
5. **Handle Pool Exhaustion:** Either wait for available or create new (with limits)
6. **Thread Safety:** Use synchronized collections or locks

## statePattern

Overview (README.md)

## State Pattern

### What is it?

The State pattern allows an object to alter its behavior when its internal state changes. The object appears to change its class when the state changes.

### When to use it?

- Object behavior depends on state and must change at runtime
- Different behaviors for different states
- State transitions are complex
- Avoid large if-else chains checking object state

## Real-world Example

**ATM Machine:** Behavior changes based on states (IdleState → CardInsertedState → PinVerifiedState). Operations allowed depend on current state.

## Key Benefits

✓ Eliminates large conditional statements ✓ Single Responsibility: Each state class handles one state ✓ Open/Closed Principle: Add new states without modifying existing ones ✓ Makes state transitions explicit ✓ Improves code readability and maintainability

## Key Drawbacks

✗ Creates many state classes (increases complexity) ✗ Overkill for simple state machines ✗ States need access to context (tight coupling possible)

# Easy Analogy

**Think of it like an ATM machine:** IdleState → You insert card (CardInsertedState) → You enter PIN (PinVerifiedState) → You withdraw money. At each state, different operations are allowed. You can't withdraw without entering PIN.

## Implementation Notes

- **Context:** Maintains instance of concrete state
- **State:** Interface defining state-specific behavior
- **ConcreteState:** Implements behavior for specific state
- Context delegates calls to current state
- States can change context state via context reference
- Use state pattern for 3+ states or complex transitions

### UML / Class Diagram

## State - Class Diagram



## Class Relationships

Class	Responsibility	Depends On
ATMState	Interface defining operations for a state	None
IdleStateATM	Initial state - only allows card insertion	Implements ATMState
CardInsertedState	Card inserted - only allows PIN entry	Implements ATMState
PinVerifiedState	PIN verified - only allows withdrawal	Implements ATMState
ATM	Context - delegates to current state	Holds reference to ATMState

## How to Code This Pattern

1. **Create State Interface:** Define operations like `insertCard()`, `insertPin()`, `withdraw()`
2. **Create Concrete States:** Each state implements the interface
3. **In State Implementation:** Only allow valid operations, change ATM state accordingly
4. **Create Context (ATM):** Maintain current state, delegate method calls to it
5. **State Transitions:** From within state methods, call `atm.setState(newState)`
6. **Invalid Operations:** Throw exception or do nothing for invalid state transitions

## strategyPattern

Overview (README.md)

## Strategy Pattern

### What is it?

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.

### When to use it?

- Many related classes differ only in behavior
- You need different variants of an algorithm
- Avoid conditional statements for algorithm selection
- Need to switch algorithms at runtime

## Real-world Example

**Vehicle Driving Modes:** Normal vehicle drives normally, sports vehicle can switch to sports drive mode. Different driving strategies without changing vehicle class.

### Key Benefits

✓ Eliminates conditional statements ✓ Easy to switch algorithms at runtime ✓ Single Responsibility: Each strategy handles one algorithm ✓ Open/Closed Principle: Add new strategies easily ✓ Improved testability (test each strategy independently)

### Key Drawbacks

✗ Creates many strategy classes (increases classes count) ✗ Overkill for simple algorithms ✗ Client must know about strategies ✗ Runtime overhead from dynamic dispatch

## Easy Analogy

**Think of it like choosing different routes to reach your destination:** You have a car (Vehicle) → You can use GoogleMaps route (Strategy1) → Or Waze route (Strategy2) → Or local knowledge route (Strategy3). Same destination, different strategies. You can switch anytime.

# Implementation Notes

- **Context:** Uses strategy interface, doesn't know concrete strategy
- **Strategy:** Interface defining algorithm contract
- **ConcreteStrategy:** Implements specific algorithm
- Strategy can be set during initialization or at runtime
- Use dependency injection for strategy assignment
- Strategies should be stateless when possible

## UML / Class Diagram

# Strategy - Class Diagram



## Class Relationships

Class	Responsibility	Depends On
DriveStrategy	Interface defining driving algorithms	None
NormalDrive	Concrete strategy - normal driving behavior	Implements DriveStrategy
SportsDrive	Concrete strategy - aggressive driving behavior	Implements DriveStrategy
Vehicle	Context - uses strategy to drive	Holds reference to DriveStrategy

## How to Code This Pattern

1. **Create Strategy Interface:** Define `drive()` method
2. **Create Concrete Strategies:** `NormalDrive` and `SportsDrive`
3. **Create Context (Vehicle):** Maintain reference to strategy
4. **Implement `setDriveStrategy()`:** Allow changing strategy at runtime
5. **Delegate to Strategy:** `vehicle.drive()` calls `strategy.drive()`
6. **Switch Algorithms:** Change strategy based on user input or conditions

## templatePattern

### Overview (README.md)

# Template Pattern

## What is it?

The Template Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. It lets subclasses redefine certain steps without changing the algorithm's structure.

## When to use it?

- Multiple classes have similar algorithm structure
- Want to avoid code duplication
- Define invariant parts in base class, variant parts in subclasses
- Invert control (Hollywood principle: "Don't call us, we'll call you")

## Real-world Example

**Payment Processing:** `PaymentFlow` template defines steps (`validate` → `charge` → `confirm`). Different payment types (`ToFriend`, `ToMerchant`) implement specific steps differently.

## Key Benefits

✓ Eliminates code duplication ✓ Single Responsibility: Separate algorithm structure from implementation ✓ Open/Closed Principle: Extend without modifying template ✓ Consistent algorithm execution ✓ Easy to maintain and extend

## Key Drawbacks

✗ Class hierarchy may become complex ✗ Violation of Liskov Substitution if subclasses don't follow contract ✗ Tight coupling between base and derived classes ✗ Limited flexibility in algorithm structure

## Easy Analogy

**Think of it like a cooking recipe:** All recipes have basic steps: `Prepare ingredients` → `Cook` → `Plate up`. But within each step, the details differ. `Pizza` preparation is different from `Biryani` preparation. The template (steps) stays the same, details vary.

## Implementation Notes

- **AbstractClass:** Defines template method with algorithm skeleton

- Template method calls abstract methods (hooks)
- Subclasses override specific abstract methods
- Use final keyword on template method to prevent override
- Consider using hooks (extension points) for optional customization
- Keep base class focused on algorithm structure

## UML / Class Diagram

# Template Method - Class Diagram



## Class Relationships

Class	Responsibility	Depends On
PaymentFlow	Abstract class - defines payment algorithm skeleton (template)	None
PaymentToFriend	Concrete implementation - specific steps for friend payment	Extends PaymentFlow
PaymentToMerchant	Concrete implementation - specific steps for merchant payment	Extends PaymentFlow

## How to Code This Pattern

1. **Create Abstract Class:** Define `pay()` as final (can't override)
2. **Define Template Method:** `pay()` calls abstract methods in order
3. **Define Abstract Methods:** `validate()`, `charge()`, `confirm()` (protected/abstract)
4. **Create Concrete Classes:** Implement abstract methods with specific logic
5. **Skeleton in Base:** Algorithm structure stays in base, details in subclasses
6. **Call Order:** Template method controls execution order, subclasses fill in steps

## abstractFacotoryPattern

Overview (README.md)

# Abstract Factory Pattern

## What is it?

The Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

## When to use it?

- System needs to work with multiple families of related objects
- You want to provide a library showing only interfaces, not implementation
- Need to enforce creating products from same family
- Consistency required among products

## Real-world Example

**Car Manufacturing:** Different car types (Economy, Luxury) need matching components (Engine, Tyre). Abstract factory ensures economy cars get economy components.

## Key Benefits

✓ Isolates concrete classes from client code ✓ Easy to swap product families ✓ Enforces consistency among related products ✓ Simplifies extending to support new families ✓ Single Responsibility: Separate object creation from usage

## Key Drawbacks

✗ More complex than simple factory (multiple factory classes) ✗ Adding new product type requires modifying all factories ✗ Can be overkill for simple scenarios ✗ Indirect object creation increases complexity

## Easy Analogy

**Think of it like different restaurant chains:** McDonalds Factory makes burgers, fries, coke → KFC Factory makes chicken, fries, coke. Both have same product categories but different implementations. Customers don't care, they just ask the factory.

## Implementation Notes

- **AbstractFactory:** Declares factory methods for creating products
- **ConcreteFactory:** Implements creation of specific product family
- **AbstractProduct:** Interface for related products
- **ConcreteProduct:** Concrete implementation of products
- Client works with abstract interfaces only
- Use when you have 2+ product families with multiple products each

# Abstract Factory - Class Diagram



Syntax error in text  
mermaid version 11.12.2

## Class Relationships

Class	Responsibility	Depends On
CarFactory	Abstract factory interface - declares creation methods	None
EconomyCarFactory	Creates economy family products	Creates Economy*, implements CarFactory
LuxuryCarFactory	Creates luxury family products	Creates Luxury*, implements CarFactory
Car/Engine/Tyre	Product interfaces	None
Economy/Luxury variants	Concrete products for each family	Implement product interfaces

## How to Code This Pattern

1. **Create Abstract Factory:** Interface with methods for each product
2. **Create Product Interfaces:** Car, Engine, Tyre
3. **Create Concrete Products:** Economy and Luxury variants
4. **Create Concrete Factories:** Each factory creates its family
5. **Ensure Consistency:** Factory ensures related products are from same family
6. **Client Code:** Use factory interface only, don't know concrete classes

## abstractFactoryDesignPattern

## builderPattern

Overview (README.md)

# Builder Pattern

## What is it?

The Builder pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

## When to use it?

- Object has many optional parameters
- Creating object with many constructor parameters (telescoping constructors problem)
- Want step-by-step construction of complex objects
- Object construction is expensive or multi-step

## Real-world Example

**Student Registration:** Build student objects with many optional fields (name, age, branch, courses). Avoid complex constructor signatures.

## Key Benefits

✓ Clear, fluent API (readable code) ✓ Handles many optional parameters elegantly ✓ Immutable objects (once built, cannot change) ✓ Single Responsibility: Separate construction from representation ✓ Flexible construction (different representations possible)

## Key Drawbacks

✗ More classes (builder class needed) ✗ More code required compared to simple constructors ✗ Slightly more memory overhead ✗ Overkill for simple objects

## Easy Analogy

**Think of it like ordering a custom pizza:** You don't say "give me all toppings" or "nothing". You build it step by step: "Base → Cheese → Tomato → Pepperoni → Done!". Finally you get your customized pizza.

## Implementation Notes

- **Builder:** Nested static class with fluent API
- Use method chaining for readability (return this)
- Implement build() to create final object
- Provide sensible defaults for optional parameters
- Constructor should be private to force builder use
- Consider immutability after building
- Can use lombok @Builder annotation for less boilerplate

# Builder - Class Diagram



## Class Relationships

	Class	Responsibility	Depends On
Student		Object with multiple optional fields (immutable after building)	Built by StudentBuilder
StudentBuilder		Builds Student step-by-step with fluent API	Creates Student

## How to Code This Pattern

1. **Create Product Class:** Student with private constructor accepting builder
2. **Create Builder Class:** Nested static class with same fields as product
3. **Implement Fluent Methods:** Each setter returns `this` for chaining
4. **Add Defaults:** Provide sensible default values for optional fields
5. **Implement build():** Create and return final product
6. **Usage:** `new StudentBuilder().setName("X").setAge(20).build()`
7. **Optional:** Use `@Builder` annotation from Lombok to auto-generate

## factoryPattern

Overview (README.md)

# Factory Pattern

## What is it?

The Factory pattern provides an interface for creating objects, but lets subclasses decide which class to instantiate. It creates objects without specifying the exact classes to create.

## When to use it?

- A class can't anticipate the type of objects it needs to create
- Want to delegate object creation to subclasses
- Object creation logic is complex and needs to be centralized
- Need to decouple object creation from usage

## Real-world Example

**Shape Creation:** Factory creates different shapes (Circle, Rectangle) based on input type. Client doesn't need to know concrete shape classes.

## Key Benefits

✓ Decouples client from concrete classes ✓ Centralizes object creation logic ✓ Easy to add new object types ✓ Follows Open/Closed Principle ✓ Simplifies client code

## Key Drawbacks

✗ More classes needed (factory + concrete classes) ✗ Can be overkill for simple object creation ✗ Indirection makes code harder to follow ✗ Over-abstraction in simple scenarios

## Easy Analogy

**Think of it like a car dealership:** You go to the dealership (Factory) and ask for a "Honda Civic" (type). The dealership gives you the right car without you worrying how it's built. You just know the interface: "it has 4 wheels, can start and stop".

## Implementation Notes

- **Creator/Factory:** Interface with factory method
- **ConcreteCreator:** Implements factory method for specific type
- **Product:** Interface for created objects
- **ConcreteProduct:** Concrete implementation
- Use enums or string identifiers for object type selection
- Can be static method factory instead of inheritance-based
- Ensure all created objects conform to product interface

# Factory Method - Class Diagram



## Class Relationships

	Class	Responsibility	Depends On
Shape		Product interface	None
Circle/Rectangle/Square		Concrete products	Implement Shape
ShapeFactory		Creator - factory method for creating shapes	Returns Shape interface

## How to Code This Pattern

1. **Create Product Interface:** Define draw() method
2. **Create Concrete Products:** Circle, Rectangle, Square
3. **Create Factory Class:** Static method getShape(String type)
4. **Implement Factory Logic:** Switch/if-else to create right shape
5. **Return Interface:** Always return Shape type, not concrete class
6. **Client Usage:** Shape shape = ShapeFactory.getShape("circle")
7. **Encapsulate Creation:** Client doesn't know how to create shapes

## prototypePattern

Overview (README.md)

# Prototype Pattern

## What is it?

The Prototype pattern creates new objects by copying an existing object (prototype) rather than creating from scratch. Useful when object creation is expensive.

## When to use it?

- Object creation is expensive or slow
- Need to avoid subclassing for object creation
- Want to create clones of objects with different states
- Need to decouple object creation from concrete classes

## Real-world Example

**Network Connection Cloning:** Clone existing network connections (shallow/deep cloning) with different configurations instead of creating from scratch.

## Key Benefits

✓ Faster object creation (copy existing object) ✓ Avoids expensive initialization ✓ Reduces memory usage in some cases ✓ Decouples client from concrete classes ✓ Flexible cloning strategies

## Key Drawbacks

✗ Shallow vs deep cloning complexity ✗ Clone method implementation required for all classes ✗ Circular references complicate deep cloning ✗ Performance overhead from copying large objects

## Easy Analogy

**Think of it like photocopying documents:** You have an original document → You make a photocopy (shallow clone) → But if the original has attachments, the copy references the same attachments. With deep copy, you also copy the attachments.

## Implementation Notes

- Implement Cloneable interface
- Override clone() method to create copy
- **Shallow Clone:** Only copy reference fields (uses default clone)
- **Deep Clone:** Create new instances of referenced objects
- Use copy constructor alternative to clone()
- Be careful with mutable fields (may need deep clone)
- Handle CloneNotSupportedException

UML / Class Diagram

# Prototype - Class Diagram



## Class Relationships

	Class	Responsibility	Depends On
Cloneable		Interface for cloneable objects	None
NetworkConnection		Implements cloning - supports both shallow and deep	Implements Cloneable

## How to Code This Pattern

1. **Implement Cloneable:** Make class implement Cloneable interface
2. **Override clone():** Create and return copy of object
3. **Shallow Clone:** Copy primitive fields, share reference fields
4. **Deep Clone:** Create new instances of referenced objects
5. **Handle Arrays/Lists:** Create new list and add cloned elements
6. **Catch Exception:** Catch CloneNotSupportedException in implementation
7. **Usage:** NetworkConnection copy = original.clone()
8. **Alternative:** Use copy constructor instead of clone()

## singletonDesignPattern

Overview (README.md)

## Singleton Pattern

### What is it?

The Singleton pattern restricts the instantiation of a class to a single object and provides a global point of access to that instance.

### When to use it?

- Only one instance of a class should exist (e.g., database connection, logger)
- Need global access to the instance
- Ensure instance is created only once
- Control access to shared resource

### Real-world Example

**Database Connection:** Only one DB connection instance should exist for the entire application. Singleton ensures single instance and global access.

### Key Benefits

✓ Ensures single instance ✓ Global point of access ✓ Lazy initialization possible ✓ Thread-safe implementation available ✓ Prevents multiple instantiation

### Key Drawbacks

✗ Difficult to test (global state) ✗ Hidden dependencies (singleton dependency not clear) ✗ Violates Single Responsibility Principle ✗ Not thread-safe in all implementations ✗ Makes code less flexible

### Easy Analogy

**Think of it like a country’s president:** There’s only ONE president at a time. When you need to talk to the president, you ask for “the president” not “create a new president”. Everyone gets the same president instance.

### Implementation Notes

- Private constructor to prevent instantiation
- Static instance variable
- Public static getInstance() method
- **Eager Initialization:** Create instance at class loading
- **Lazy Initialization:** Create instance on first access
- **Thread-safe:** Use synchronized block or double-checked locking
- Consider using Enums for thread-safe singleton
- Implement Serializable/Cloneable carefully to maintain singleton

UML / Class Diagram

## Singleton - Class Diagram



## Class Relationships

Class	Responsibility	Depends On
DBConnection	Singleton - only one instance, global access	Self-manages instance

## How to Code This Pattern

### Eager Initialization (Thread-Safe)

```
class DBConnection {
    private static final DBConnection instance = new DBConnection();

    private DBConnection() { }

    public static DBConnection getInstance() {
        return instance;
    }
}
```

### Lazy Initialization (Thread-Safe)

```
class DBConnection {
    private static DBConnection instance;

    private DBConnection() { }

    public static synchronized DBConnection getInstance() {
        if (instance == null) {
            instance = new DBConnection();
        }
        return instance;
    }
}
```

### Double-Checked Locking

```
class DBConnection {
    private static volatile DBConnection instance;

    private DBConnection() { }

    public static DBConnection getInstance() {
        if (instance == null) {
            synchronized(DBConnection.class) {
                if (instance == null) {
                    instance = new DBConnection();
                }
            }
        }
        return instance;
    }
}
```

## Key Points

- **Private Constructor:** Prevent instantiation from outside
- **Static Instance:** Hold single instance
- **getInstance():** Return the singleton instance
- **Thread-Safe:** Use synchronized or volatile for concurrency
- **Enum Alternative:** Use enum for automatic singleton

## AdapterDesignPattern

### Overview (README.md)

## Adapter Pattern

### What is it?

The Adapter pattern converts the interface of a class into another interface clients expect. It allows incompatible objects to collaborate.

### When to use it?

- Have incompatible interfaces that need to work together
- Integrate third-party libraries with different interfaces
- Want to reuse existing class with incompatible interface
- Create bridge between old and new code

## Real-world Example

**Weight Machine Adapter:** Weight machine returns weight in different formats. Adapter converts to pounds/kg format that clients expect.

## Key Benefits

✓ Allows collaboration between incompatible objects ✓ Improves code reusability ✓ Follows Single Responsibility Principle ✓ Follows Open/Closed Principle ✓ Helps integrate legacy code

## Key Drawbacks

✗ Added complexity with extra adapter class ✗ Can hide design problems in original interfaces ✗ May reduce performance due to indirection ✗ Overkill for simple interface changes

## Easy Analogy

**Think of it like an electrical adapter:** You have a device (WeightMachine) that outputs in kg, but your client needs it in pounds. The adapter converts kg to pounds without changing the original device.

## Implementation Notes

- **Target:** Interface clients expect
- **Adaptee:** Existing interface that needs adaptation
- **Adapter:** Converts adaptee interface to target interface
- **Two types:** Class adapter (inheritance) and object adapter (composition)
- Object adapter is preferred (more flexible)
- Keep adapter simple (avoid complex logic)
- Consider whether modification of adaptee is possible

## UML / Class Diagram

## Adapter - Class Diagram



## Class Relationships

Class	Responsibility	Depends On
WeightMachine	Adaptee - existing class with incompatible interface	None
WeightMachineAdapter	Target interface - interface client expects	None
WeightMachineAdapterImpl	Adapter - converts adaptee to target interface	Has WeightMachine, implements Adapter

## How to Code This Pattern

### Object Adapter (Composition - Recommended)

```
class WeightMachineAdapterImpl implements WeightMachineAdapter {
    private WeightMachine weightMachine;

    public WeightMachineAdapterImpl(WeightMachine wm) {
        # Adapter - Class Diagram

        ```mermaid
        classDiagram
            class WeightMachine {
                +getWeight(): double
            }

            class WeightReader {
                <<interface>>
                +getWeightInPounds(): double
            }

            class WeightMachineAdapterImpl {
                -weightMachine: WeightMachine
                +WeightMachineAdapterImpl(wm: WeightMachine)
                +getWeightInPounds(): double
            }

            %% Relationships
            WeightMachineAdapterImpl --> WeightMachine : uses
            WeightReader <|.. WeightMachineAdapterImpl

            ...

            ## Class Relationships

            | Class | Responsibility | Depends On |
            |-----|-----|-----|
            | **WeightMachine** | Adaptee - existing class providing weight in its own units | None |
            | **WeightReader** | Target interface - what clients expect (weight in pounds) | None |
            | **WeightMachineAdapterImpl** | Adapter (object adapter) - converts WeightMachine to WeightReader | Uses `WeightMachine`, implements `WeightReader`

            ## Plain Explanation

            The client expects `WeightReader` (method `getWeightInPounds()`), but `WeightMachine` only provides `getWeight()`. The adapter (`WeightMachineAdapterImpl`) bridges this gap.

            ## How to Code This Pattern (Minimal)

            ### Object Adapter (Composition - recommended)
            ```java
```

```
// Target
public interface WeightReader {
    double getWeightInPounds();
}

// Adaptee
public class WeightMachine {
    public double getWeight() { return 50.0; /* kg */ }
}

// Adapter
public class WeightMachineAdapterImpl implements WeightReader {
    private final WeightMachine weightMachine;

    public WeightMachineAdapterImpl(WeightMachine wm) { this.weightMachine = wm; }

    @Override
    public double getWeightInPounds() {
        return weightMachine.getWeight() * 2.20462; // kg -> lb
    }
}

### Class Adapter (Inheritance – less flexible)
```java
public class WeightMachineAdapter extends WeightMachine implements WeightReader {
    @Override
    public double getWeightInPounds() {
        return getWeight() * 2.20462;
    }
}
```

## Key Points

- Use object adapter (composition) when you can: it works with final/adapted classes and is more flexible.
- Use class adapter (inheritance) only when you must inherit and the adaptee's interface is compatible with inheritance.
- Adapter focuses on converting interfaces, not changing behaviour.
```

---

## ## DecoratorDesign

### ### Overview (README.md)

#### # Decorator Pattern

#### ## What is it?

The Decorator pattern attaches additional responsibilities to an object dynamically. It provides a flexible alternative to subclassing for extending functionality.

#### ## When to use it?

- Add responsibilities to individual objects without affecting others
- Subclassing would create too many classes
- Need to combine multiple behaviors dynamically
- Avoid "explosion" of subclasses

#### ## Real-world Example

**\*\*Ice Cream Shop\*\***: Base ice cream (Vanilla, Chocolate) can be decorated with toppings (ChocolateSyrup, ChocoChips). Each decorator adds features.

#### ## Key Benefits

- ✓ More flexible than subclassing
- ✓ Add/remove responsibilities at runtime
- ✓ Combine behaviors dynamically
- ✓ Single Responsibility: Each decorator has one responsibility
- ✓ Avoids subclass explosion

#### ## Key Drawbacks

- x Many small classes (decorators)
- x Complex object composition
- x Decoration order matters (may affect functionality)
- x Can be overkill for simple additions

#### ## Easy Analogy

**\*\*Think of it like customizing a shirt:\*\***

Basic shirt → Add sleeves (decorator) → Add buttons (another decorator) → Add collar (another decorator). Each decorator adds functionality without changing the basic shirt.

#### ## Implementation Notes

- **\*\*Component\*\***: Interface for decorators and concrete objects
- **\*\*ConcreteComponent\*\***: Original object being decorated
- **\*\*Decorator\*\***: Abstract class maintaining reference to component
- **\*\*ConcreteDecorator\*\***: Adds specific responsibility
- Decorator wraps component and delegates calls
- Can wrap decorator with another decorator
- Maintain same interface as component
- Watch order of decoration (may affect results)

### ### UML / Class Diagram

#### # Decorator – Class Diagram

```
```mermaid
classDiagram
    class IceCream {
        <<interface>>
        +getCost(): double*
        +getDescription(): String*
    }

    class VanillaIceCream {
        +getCost(): double
        +getDescription(): String
    }

    class IceCreamDecorator {
        <<abstract>>
        -iceCream: IceCream
        +getCost(): double
        +getDescription(): String
    }
```
```

```
class ChocolateSyrupDecorator {
    +getCost(): double
    +getDescription(): String
}

class ChocoChipsDecorator {
    +getCost(): double
    +getDescription(): String
}

IceCream <|.. VanillaIceCream
IceCream <|.. IceCreamDecorator
IceCreamDecorator <|-- ChocolateSyrupDecorator
IceCreamDecorator <|-- ChocoChipsDecorator
IceCreamDecorator --> IceCream: wraps
```

## Class Relationships

| Class                   | Responsibility   | Depends On                        |
|-------------------------|--|-----------------------------------|
| IceCream                | Component interface - defines ice cream operations     | None                              |
| VanillaIceCream         | Concrete component - base ice cream                    | Implements IceCream               |
| IceCreamDecorator       | Abstract decorator - wraps ice cream and adds features | Has IceCream, implements IceCream |
| ChocolateSyrupDecorator | Concrete decorator - adds chocolate syrup              | Extends IceCreamDecorator         |
| ChocoChipsDecorator     | Concrete decorator - adds choco chips                  | Extends IceCreamDecorator         |

## How to Code This Pattern

1. **Create Component Interface:** IceCream with getCost() and getDescription()
2. **Create Concrete Component:** VanillaIceCream implements interface
3. **Create Abstract Decorator:** Implements interface, wraps component
4. **Decorator holds Component:** IceCreamDecorator has IceCream reference
5. **Decorator delegates:** Call wrapped component's methods first, then add own behavior
6. **Create Concrete Decorators:** Each adds specific feature
7. **Stack Decorators:** Can wrap decorator with another decorator

## Example Usage

```
IceCream iceCream = new VanillaIceCream(); // cost: 30
iceCream = new ChocolateSyrupDecorator(iceCream); // cost: 30 + 15 = 45
iceCream = new ChocoChipsDecorator(iceCream); // cost: 45 + 10 = 55
```

## Key Points

- **Wrapper Pattern:** Decorators wrap components
- **Same Interface:** Decorator implements same interface as component
- **Recursive Composition:** Can nest decorators
- **Runtime Addition:** Add features at runtime

## ProxyPattern

Overview (README.md)

# Proxy Pattern

## What is it?

The Proxy pattern provides a surrogate or placeholder for another object to control access to it. Proxy acts on behalf of the real object.

## When to use it?

- Defer expensive object creation (lazy initialization)
- Control access to sensitive objects
- Log access to objects
- Cache results of expensive operations
- Restrict operations on objects

## Real-world Example

**Employee Data Cache Proxy:** Real expensive employee data fetches from DB. Proxy caches data in Redis, returning cached results when available.

## Key Benefits

✓ Control access to objects ✓ Lazy initialization (create object only when needed) ✓ Add logging/caching without modifying original ✓ Follows Single Responsibility Principle ✓ Transparent to client code

## Key Drawbacks

✗ Added complexity with extra proxy class ✗ Performance overhead from proxy indirection ✗ Response time slightly increased ✗ Overkill for simple objects

## Easy Analogy

Think of it like a personal assistant: You (Real Subject) are busy. Your assistant (Proxy) handles your calls → If someone asks your salary, assistant checks notes (cache) → If not there, asks you. No one directly bothers you.

## Implementation Notes

- **Subject:** Interface for proxy and real object
- **RealSubject:** Actual object being proxied
- **Proxy:** Manages access to real subject
- Proxy implements same interface as subject
- Can control access, cache, or defer creation
- **Types:** Protection proxy, virtual proxy, cache proxy, remote proxy
- Maintain reference to real subject
- Delegate calls to real subject after proxy logic

## UML / Class Diagram

## Proxy - Class Diagram



Syntax error in text  
mermaid version 11.12.2

## Class Relationships

| Class              | Responsibility                         | Depends On               |
|--------------------|--|--------------------------|
| Employee           | Subject interface - defines operations | None                     |
| EmployeeImpl       | Real subject - actual expensive object | Implements Employee      |
| EmployeeCacheProxy | Proxy - controls access, adds caching  | Has Employee, uses cache |
| RedisCacheClient   | External cache - stores cached data    | None                     |

## How to Code This Pattern

1. **Create Subject Interface:** Employee with `getSalary()`, `getName()`
2. **Create Real Subject:** EmployeeImpl does expensive DB operations
3. **Create Proxy:** Implements same interface

```
class EmployeeCacheProxy implements Employee {
    private Employee employee;
    private Map<String, Object> cache;

    public EmployeeCacheProxy(Employee emp) {
        this.employee = emp;
        this.cache = new HashMap<>();
    }

    @Override
    public int getSalary() {
        if (cache.containsKey("salary")) {
            return (int) cache.get("salary");
        }
        int salary = employee.getSalary();
        cache.put("salary", salary);
        return salary;
    }
}
```

4. **Proxy Controls Access:**
  - Cache before delegating
  - Lazy load real object
  - Log/track access
  - Restrict access
5. **Client uses Proxy:** Employee emp = new EmployeeCacheProxy(...)

## Proxy Types

| Type             | Purpose                                  |
|------------------|--|
| Protection Proxy | Control access (security)                |
| Virtual Proxy    | Lazy initialization of expensive objects |
| Cache Proxy      | Cache expensive operations               |
| Remote Proxy     | Represent remote object                  |

## Key Points

- **Same Interface:** Proxy implements same interface as real object
- **Transparent:** Client doesn't know it's using proxy
- **Add Behavior:** Before/after delegating to real object
- **Control Access:** Decide when and how to delegate

## bridgePattern

### Overview (README.md)

# Bridge Pattern

## What is it?

The Bridge pattern decouples an abstraction from its implementation so that the two can vary independently. It bridges the gap between abstraction and implementation.

## When to use it?

- Abstraction and implementation should be independent
- Changes in implementation shouldn't affect client code
- Want to avoid permanent binding between abstraction and implementation
- Multiple implementations for same abstraction

## Real-world Example

**Living Things Breathing:** Different living things (Dog, Fish, Tree) have different breathing mechanisms (Lungs, Gills, Photosynthesis). Bridge separates living things from breathing processes.

## Key Benefits

✓ Decouples abstraction from implementation ✓ Implementation can change without affecting clients ✓ Avoid permanent binding between abstraction and implementation ✓ Improves code maintainability ✓ Follows Open/Closed Principle

## Key Drawbacks

✗ More complex design (more classes needed) ✗ Overkill for simple hierarchies ✗ Increased indirection affects performance ✗ Can overcomplicate simple designs

## Easy Analogy

**Think of it like a car and its engine types:** A car is abstraction (LivingThing) → It can be Sedan, SUV (Dog, Fish) → Engine type is implementation (Breathing) → Can be Petrol, Diesel (Lungs, Gills). You can mix any car with any engine.

## Implementation Notes

- **Abstraction:** Defines high-level interface
- **RefinedAbstraction:** Extends abstraction
- **Implementor:** Defines lower-level interface
- **ConcreteImplementor:** Implements specific behavior
- Abstraction should contain reference to Implementor
- Use composition instead of inheritance
- Separate abstraction hierarchy from implementation hierarchy

## UML / Class Diagram

## Bridge - Class Diagram



Syntax error in text  
mermaid version 11.12.2

## Class Relationships

|                           | Class | Responsibility                              | Depends On                 |
|---------------------------|-------|---|----------------------------|
| LivingThing               |       | Abstraction - high-level interface          | Uses BreathingProcess      |
| Dog/Fish/Tree             |       | Refined abstractions                        | Extend LivingThing         |
| BreathingProcess          |       | Implementor interface - low-level interface | None                       |
| Lungs/Gill/Photosynthesis |       | Concrete implementors                       | Implement BreathingProcess |

## How to Code This Pattern

1. **Separate Hierarchies:** Keep abstraction and implementation in separate hierarchies
2. **Abstraction contains Implementor:** LivingThing has reference to BreathingProcess
3. **Concrete Abstractions:** Dog, Fish, Tree extend abstraction
4. **Concrete Implementors:** Implement the implementor interface
5. **Combine dynamically:** Dog can use any BreathingProcess
6. **Avoid Explosion:** Without bridge, would have Dog-Lungs, Dog-Gills, etc. classes

## Key Points

- **Bridge:** Separate abstraction from implementation
- **Two Hierarchies:** One for abstraction, one for implementation
- **Composition:** Abstraction uses composition with implementor
- **Flexibility:** Change implementation independently from abstraction

# Composite Pattern

## What is it?

The Composite pattern composes objects into tree structures to represent part-whole hierarchies. It allows clients to treat individual objects and compositions uniformly.

## When to use it?

- Represent hierarchical structures (trees)
- Clients should treat individual and composite objects the same way
- Part-whole relationships need to be represented
- File system or menu structures

## Real-world Example

**File System:** Files and directories form a tree. Both can have operations like get size, copy. Composite pattern treats them uniformly.

## Key Benefits

✓ Simplifies client code (same treatment for leaf and composite) ✓ Flexible tree structures (easy to add/remove nodes) ✓ Follows Single Responsibility Principle ✓ Follows Open/Closed Principle ✓ Recursive composition is natural

## Key Drawbacks

✗ May force inappropriate operations on leaves ✗ Less type safety (both leaf and composite look same) ✗ Complexity increases with tree depth ✗ Performance issues with deep trees

## Easy Analogy

**Think of it like a folder structure on your computer:** Folders contain files and other folders → You can ask any folder/file “what’s your size?” → A folder calculates size by adding all children’s sizes. Same operation, different implementations.

## Implementation Notes

- **Component:** Declares common operations
- **Leaf:** Represents leaf objects (no children)
- **Composite:** Represents composite objects (has children)
- Composite implements add/remove/get children methods
- Both leaf and composite implement component operations
- Watch out for inappropriate operations on leaves
- Consider empty checks for operations on leaves

## UML / Class Diagram

## Composite - Class Diagram



Syntax error in text  
mermaid version 11.12.2

## Class Relationships

| Class      | Responsibility  | Depends On                                 |
|------------|---|--|
| FileSystem | Component interface - defines common operations         | None                                       |
| File       | Leaf - represents file with no children                 | Implements FileSystem                      |
| Directory  | Composite - contains FileSystem objects (files/folders) | Implements FileSystem, contains FileSystem |

## How to Code This Pattern

1. **Create Component Interface:** Define getSize(), add(), remove()
2. **Create Leaf Class:** File implements interface, getSize() returns file size
3. **Create Composite Class:** Directory implements interface
4. **Maintain Children:** Directory has list of FileSystem objects
5. **Add/Remove Methods:** Composite implements add() and remove()
6. **Recursive Operations:** Directory.getSize() sums all children’s sizes
7. **Uniform Treatment:** Both File and Directory are treated as FileSystem

## Key Points

- **Leaf:** File - no children, performs actual work
- **Composite:** Directory - can have children
- **Recursive:** Operations work recursively through tree
- **Uniform:** Client treats both File and Directory same way

# Flyweight Pattern

## What is it?

The Flyweight pattern uses sharing to support large numbers of fine-grained objects efficiently by sharing common state between multiple objects.

## When to use it?

- Application uses large number of similar objects
- Memory usage is a concern
- Intrinsic state can be shared between objects
- Many objects with same properties

## Real-world Example

**Robot Game:** Game has thousands of robots. Instead of creating separate objects, share intrinsic state (sprite type) and keep extrinsic state (position) separate.

## Key Benefits

✓ Significantly reduces memory usage ✓ Improves application performance ✓ Centralizes management of shared state ✓ Good for large-scale applications ✓ Transparent to client code

## Key Drawbacks

✗ Increased complexity in code ✗ Thread-safety concerns (shared objects) ✗ Performance overhead from object lookup ✗ Intrinsic/extrinsic state separation not always clear

## Easy Analogy

**Think of it like a multiplayer game with 10,000 soldiers:** All soldiers have the same armor type (shared/intrinsic) but different positions (unique/extrinsic). Instead of creating 10,000 objects, create 1 shared soldier object, reuse it with different positions.

## Implementation Notes

- **Flyweight:** Interface for shared and unique states
- **ConcreteFlyweight:** Stores intrinsic state (shared)
- **FlyweightFactory:** Creates and manages flyweight objects
- **Client:** Maintains extrinsic state
- Separate intrinsic (shared) from extrinsic (unique) state
- Use factory pattern to create/retrieve flyweights
- Ensure flyweight objects are immutable
- Consider thread-safety for shared objects
- Useful when you have thousands of similar objects

## UML / Class Diagram

## Flyweight - Class Diagram



## Class Relationships

| Class                        | Responsibility  | Depends On                           |
|------------------------------|---|--------------------------------------|
| IRobot                       | Flyweight interface - defines robot display method      | None                                 |
| HumanoidRobot/RoboticDogImpl | Concrete flyweights - store intrinsic state (sprite)    | Implements IRobot, has shared Sprite |
| RoboticFactory               | Flyweight factory - creates and caches flyweights       | Creates and manages IRobot           |
| Sprite                       | Intrinsic state - shared across many robots (immutable) | None                                 |

## How to Code This Pattern

1. **Separate Intrinsic/Extrinsic State:**
  - **Intrinsic:** Sprite (shared) - stored in flyweight
  - **Extrinsic:** Position x, y (unique) - passed as parameter
2. **Create Flyweight Interface:** IRobot with display(x, y)
3. **Create Concrete Flyweights:** Store only intrinsic state

#### 4. Create Factory: Cache and reuse flyweights

```
class RoboticFactory {  
    private Map<String, IRobot> robots = new HashMap<>();  
  
    public IRobot createRobot(String type) {  
        if (!robots.containsKey(type)) {  
            robots.put(type, new HumanoidRobot(new Sprite(...)));  
        }  
        return robots.get(type);  
    }  
}
```

5. Client provides extrinsic state: robot.display(100, 200)

## Key Points

- **Shared State:** Intrinsic state shared across many objects
- **Unique State:** Extrinsic state kept in client
- **Memory Optimization:** 10,000 robots, 1 sprite = huge savings
- **Immutable:** Flyweight objects must be immutable
- **Factory:** Factory manages caching and reuse