Prototype Design Pattern

Concept

The Prototype pattern creates new objects by cloning existing instances (prototypes) rather than creating them from scratch. It's particularly useful when object creation is expensive or complex.

UML Class Diagram

```
┌─────────────────┐
│   Prototype     │ (Interface)
├─────────────────┤
│ + clone()       │
└─────────────────┘
        △
        │
┌─────────┴─────────┐
│ ConcretePrototype │
├───────────────────┤
│ - state           │
├───────────────────┤
│ + clone()         │
│ + operation()     │
└───────────────────┘


┌─────────────────┐    ┌─────────────────┐
│    Client       │◇───────│  Prototype      │
├─────────────────┤    │                 │
│ + operation()   │    │                 │
└─────────────────┘    └─────────────────┘

Optional:
┌─────────────────┐    ┌─────────────────┐
│ PrototypeRegistry│◇──────│  Prototype      │
├─────────────────┤    │                 │
│ + getPrototype()│    │                 │
│ + addPrototype()│    │                 │
└─────────────────┘    └─────────────────┘
```

Key Components

1. Prototype Interface

- Declares `clone()` method for copying objects
- Usually extends `Cloneable` interface in Java

2. Concrete Prototype

- Implements the cloning method
- Contains the actual object state
- Provides copy constructor or clone logic

3. Client

- Creates new objects by calling `clone()` on prototypes
- Customizes cloned objects as needed

4. Prototype Registry (Optional)

- Manages and provides access to prototypes
- Centralizes prototype creation and retrieval

Flow Diagram

```
Client Request → Registry → Get Prototype → Clone → Customize → Return
     │             │           │          │        │
     │             │           │          │        ▼
     │             │           │          │    ┌─────────┐
     │             │           │          │    │New Object│
     │             │           │          │    └─────────┘
     │             │           │          │
     ▼             ▼           ▼          ▼
┌─────────┐  ┌─────────────┐  ┌─────────┐  ┌─────────┐
│ Request │  │Find/Return  │  │Original │  │ Clone   │
│ Object  │  │ Prototype   │  │Prototype│  │ Object  │
└─────────┘  └─────────────┘  └─────────┘  └─────────┘
```

Implementation Types

1. Shallow Cloning

- Copies primitive fields
- Copies references (not the objects they point to)
- Fast but shared mutable objects can cause issues

java

```
@Override
public Object clone() throws CloneNotSupportedException {
    return super.clone(); // Shallow copy
}
```

2. Deep Cloning

- Copies all fields including referenced objects
- Creates completely independent objects
- Slower but safer for complex objects

```java
@Override
public MyClass clone() {
    MyClass cloned = new MyClass();
    cloned.list = new ArrayList<>(this.list);
    cloned.map = new HashMap<>(this.map);
    return cloned;
}
```

Advantages

- **Performance**: Faster than creating complex objects from scratch
- **Simplicity**: Avoids complex initialization logic
- **Dynamic Configuration**: Can create objects at runtime
- **Reduced Subclassing**: Alternative to Factory Method pattern
- **State Preservation**: Maintains object state during cloning

Disadvantages

- **Complex Cloning**: Deep cloning can be tricky with circular references
- **Memory Usage**: Keeping prototypes in memory
- **Clone Method Complexity**: Implementing proper cloning logic
- **Cloneable Interface Issues**: Java's Cloneable has design problems
- **Maintenance**: Changes to object structure require clone method updates

When to Use Prototype Pattern

Use When:

- Object creation is expensive (database queries, file I/O, network calls)

- Need many similar objects with slight variations
- Want to avoid complex initialization
- System should be independent of how objects are created
- Need to create objects dynamically at runtime

Don't Use When:

- Object creation is simple and fast
- Objects have few variations
- Deep cloning becomes too complex
- Memory usage is a primary concern
- Objects have complex circular references

Real-World Examples

1. Database Connections

```java
// Expensive: SSL setup, connection pooling
DatabaseConnection prototype = new DatabaseConnection("host", 5432, "db");

// Fast cloning for different databases
DatabaseConnection testConn = prototype.clone();
testConn.setDatabase("test_db");
```

2. Document Templates

```java
// Expensive: Loading from disk/database
DocumentTemplate contractTemplate = new DocumentTemplate("contract.docx");

// Fast customization for clients
DocumentTemplate clientContract = contractTemplate.clone();
clientContract.customizeForClient("Acme Corp");
```

3. Game Objects

```java
```

```java
// Expensive: 3D model loading, texture loading
Enemy skeleton = new Enemy("skeleton", "models/skeleton.obj");

// Fast cloning for multiple enemies
Enemy enemy1 = skeleton.clone();
enemy1.setPosition(100, 200);
```

4. UI Components

```java
// Expensive: Style loading, event binding
Button prototype = new Button("default-style");

// Fast cloning with variations
Button saveButton = prototype.clone();
saveButton.setText("Save");
```

Best Practices

1. Implement Proper Deep Cloning

```java
public class ComplexObject implements Cloneable {
    private List<String> items;
    private Map<String, Object> properties;

    @Override
    public ComplexObject clone() {
        ComplexObject cloned = new ComplexObject();
        cloned.items = new ArrayList<>(this.items);
        cloned.properties = new HashMap<>(this.properties);
        return cloned;
    }
}
```

2. Use Copy Constructors Instead of Clone

```java
```

```java
public class MyClass {
    public MyClass(MyClass other) {
        this.field1 = other.field1;
        this.field2 = new ArrayList<>(other.field2);
    }
}
```

3. Registry Pattern for Management

```java
public class PrototypeRegistry {
    private Map<String, Prototype> prototypes = new HashMap<>();

    public void register(String key, Prototype prototype) {
        prototypes.put(key, prototype);
    }

    public Prototype create(String key) {
        return prototypes.get(key).clone();
    }
}
```

Performance Comparison

Creating from Scratch:

- Database connection: 200ms (SSL setup, authentication)
- Document template: 150ms (disk I/O, parsing)
- Game enemy: 100ms (3D model loading)

Using Prototype:

- Clone database connection: 2ms
- Clone document template: 1ms
- Clone game enemy: 3ms

Improvement: 50-100x faster!

Common Pitfalls

1. Shallow vs Deep Copy Issues

```java
```

```java
// WRONG: Shallow copy shares mutable objects
List<String> original = Arrays.asList("A", "B");
List<String> copy = original; // Shares same list!

// RIGHT: Deep copy creates new objects
List<String> copy = new ArrayList<>(original);
```

2. Circular Reference Problems

- Use visited object tracking
- Consider using serialization for deep copying
- Design objects to avoid circular references

3. Thread Safety

- Ensure prototypes are thread-safe
- Use synchronization if prototypes are shared
- Consider using ThreadLocal for per-thread prototypes

Related Patterns

- **Factory Method**: Alternative creation pattern
- **Builder**: For complex object construction
- **Singleton**: Registry often implements Singleton
- **Memento**: Similar state preservation concept

Sequence Diagram

```
Client → Registry: getPrototype("key")
Registry → Registry: find prototype
Registry → Prototype: clone()
Prototype → Prototype: create copy
Prototype → Registry: return cloned object
Registry → Client: return new instance
Client → ClonedObject: customize()
Client → ClonedObject: use()
```

Summary

The Prototype pattern is invaluable when object creation is expensive. By maintaining pre-configured prototypes and cloning them, you can achieve significant performance improvements while maintaining flexibility. It's particularly effective in scenarios involving database connections, document templates, game objects, and complex UI components where initialization overhead is substantial.