All Design Patterns

Generated design-pattern handbook — interview-ready.

What is a Design Pattern?

A design pattern is a reusable solution to a commonly occurring problem within a given context in software design. Patterns provide templates for how to solve problems that arise frequently, improving communication, maintainability, and design quality.

Why Use Design Patterns?

- Capture proven best-practices and trade-offs.
- Improve readability by using standard vocabulary.
- Promote decoupling, extensibility and testability.

Categories

- Creational Patterns — deal with object creation mechanisms.
- Structural Patterns — compose classes and objects to form larger structures.
- Behavioral Patterns — manage communication between objects.

Behavioral Patterns

Problem domain: how objects interact and distribute responsibility.

NullObjectPattern

NullObjectPattern — Interview Reference

Intent

Provide a concise intent for the NullObjectPattern pattern.

Problem Statement

Null Object Pattern

Why Simple Code Fails

Often ad-hoc solutions (if/else, scattered constructors, tight coupling) make code hard to extend and test.

Solution Overview

Describe how the NullObjectPattern pattern solves the problem by providing structure and separation of concerns.

Minimal Java Example

Without Pattern

```
// Example not provided.
```

With Pattern

```
// Example not provided.

// Example not provided in original README.
```

Advantages

- Describe key advantages (decoupling, extensibility, reuse).

Disadvantages

- Describe trade-offs (complexity, indirection, overuse).

When NOT to Use

- Situations where the pattern is unnecessary.

Common Mistakes

- Frequent anti-patterns or pitfalls.

Framework / Library Usage

- Notes on common frameworks or language features.

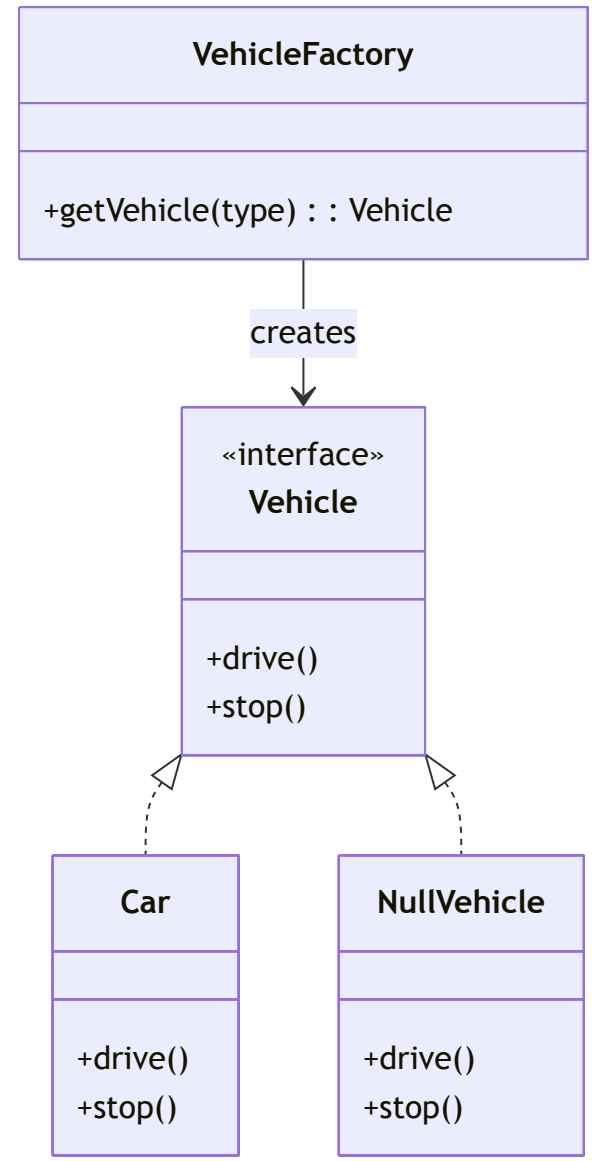
System Design Use Cases

- Real-world systems where this pattern helps.

Interview One-Liner

One-line summary of the pattern.

UML / Class Diagram



NullObjectPattern — UML Class Diagram

ObserverPattern

Observer Pattern — Interview Reference

Intent

Define a one-to-many dependency so when one object changes state, its dependents are notified and updated automatically.

Problem Statement

Multiple components need to react to state changes; tight coupling makes it hard to add/remove listeners.

Why Simple Code Fails

Hard-coded callbacks scatter notification logic and duplicate update code across observers.

Solution Overview

Introduce Subject (observable) that maintains a list of Observers; notify them on state changes.

UML Diagram

See `observable` folder UML and generated diagram at `build/diagrams/behavioralDesign_ObserverPattern_UML_ClassDiagram.md.png` (if present).

Participants / Roles

- Subject/Observable: holds state and observers
- Observer: interface for update callback
- ConcreteObserver: implements reaction to changes

Runtime Execution Flow

1. Observers register with Subject
2. Subject changes state and calls notifyObservers()
3. Each Observer.update() executes handling logic

Minimal Java Example

Without Pattern

```
// Without Observer: manual callback invocations across components
serviceChanged(); loggerUpdate(); cacheUpdate();
```

With Pattern

```
// With Observer: register observers to subject
subject.register(new LoggerObserver());
subject.register(new CacheObserver());
subject.notifyAll("UPDATE");

public interface Observer { void update(String evt); }
public class EventSource {
  private List<Observer> observers = new ArrayList<>();
  public void register(Observer o){ observers.add(o); }
  public void notifyAll(String evt){ observers.forEach(o->o.update(evt)); }
}
```

Advantages

- Loose coupling between subject and observers
- Dynamic subscription/unsubscription

Disadvantages

- Can introduce unexpected update order dependencies
- Possible memory leaks if observers not unsubscribed

When NOT to Use

- High-frequency updates where push costs are too high; prefer polling or debounced events

Common Mistakes

- Forgetting to unregister observers
- Heavy work inside update() blocking subject

Framework / Library Usage

- Use event buses (Guava EventBus), Reactive frameworks (RxJava), or Spring ApplicationEvents for production systems

System Design Use Cases

- UI event handling, cache invalidation, event-driven microservices

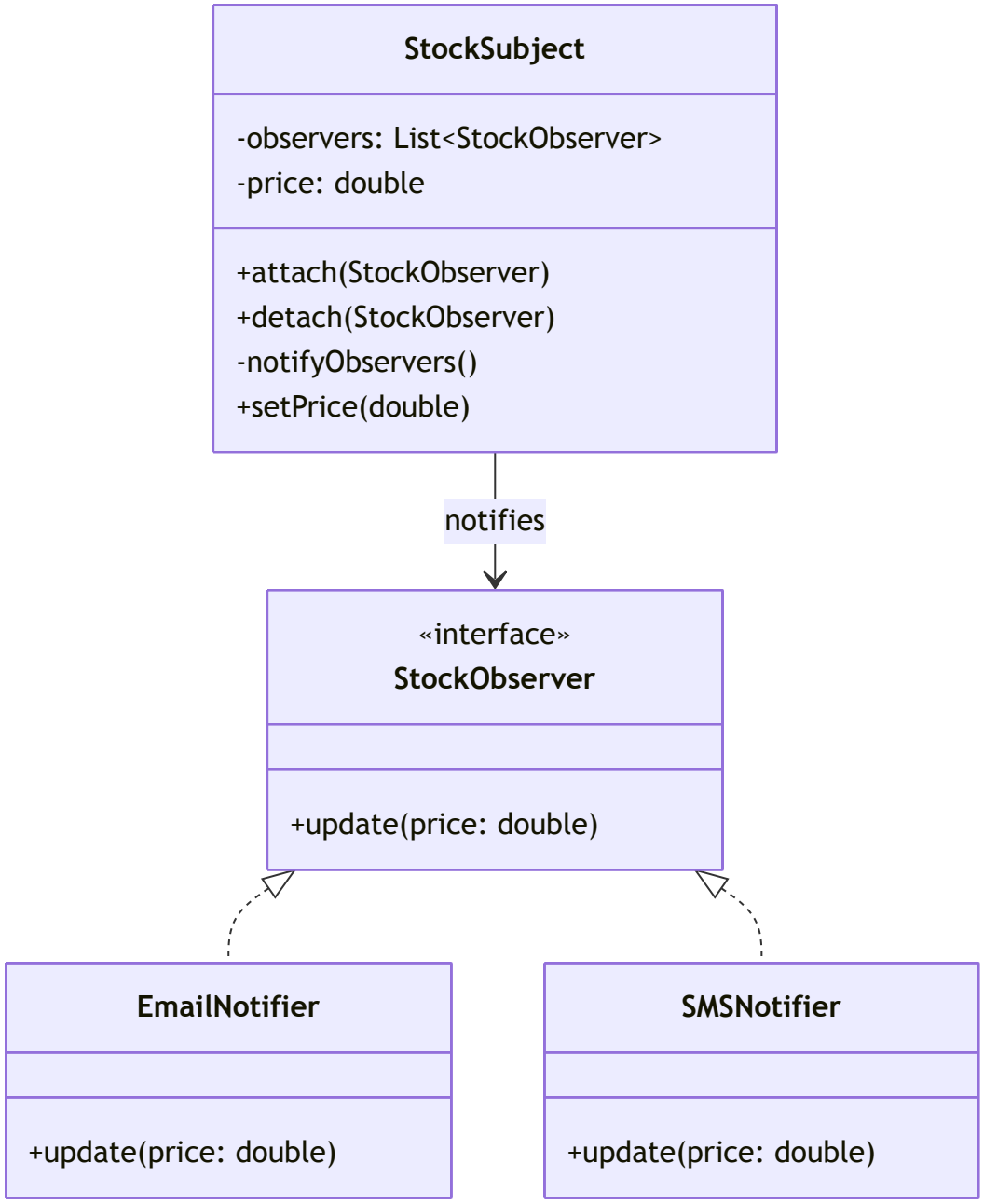
Interview One-Liner

Observer decouples state changes from reaction logic by subscribing observers to subjects.

Common Interview Questions

- How to handle slow/unreliable observers?
- How to order notifications or handle failures in observers?

UML / Class Diagram



ObserverPattern — UML Class Diagram

---

chainOfResponsibilty

chainOfResponsibilty — Interview Reference

Intent

Provide a concise intent for the chainOfResponsibilty pattern.

Problem Statement

Chain of Responsibility Pattern

Why Simple Code Fails

Often ad-hoc solutions (if/else, scattered constructors, tight coupling) make code hard to extend and test.

Solution Overview

Describe how the chainOfResponsibilty pattern solves the problem by providing structure and separation of concerns.

Minimal Java Example

Without Pattern

```
// Example not provided.
```

With Pattern

```
// Example not provided.

// Example not provided in original README.
```

Advantages

- Describe key advantages (decoupling, extensibility, reuse).

Disadvantages

- Describe trade-offs (complexity, indirection, overuse).

When NOT to Use

- Situations where the pattern is unnecessary.

Common Mistakes

- Frequent anti-patterns or pitfalls.

Framework / Library Usage

- Notes on common frameworks or language features.

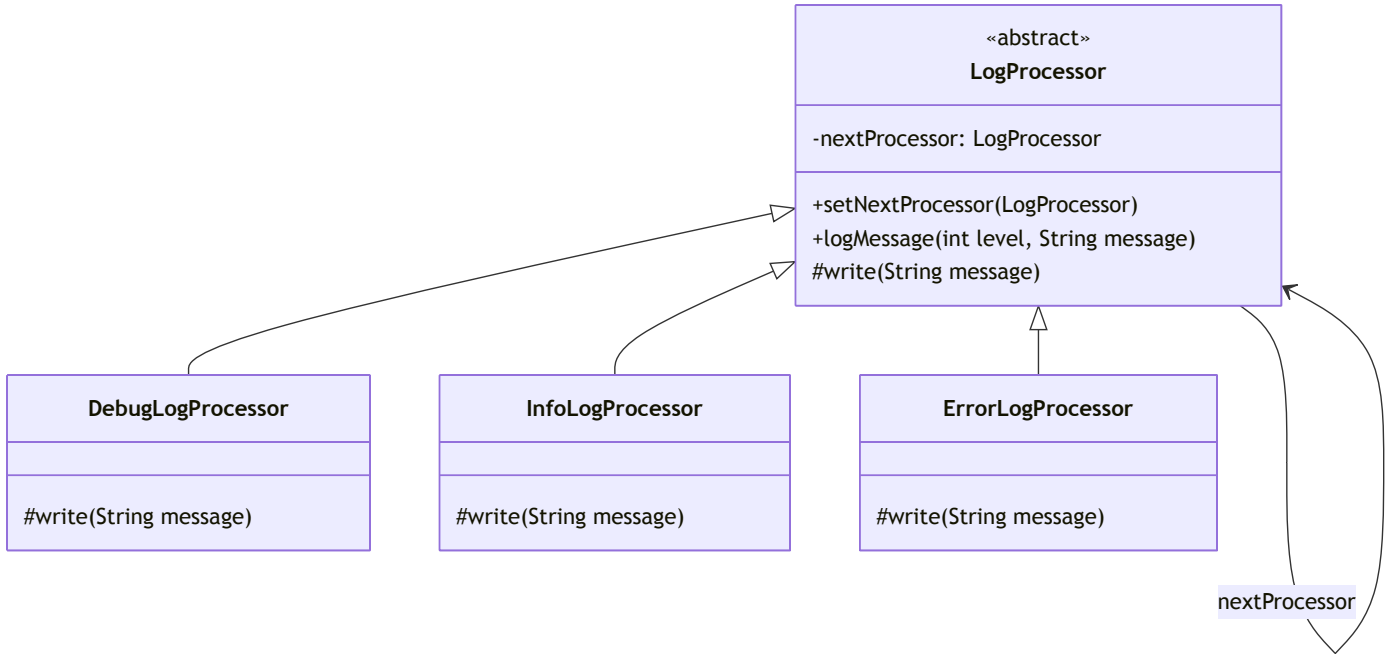
System Design Use Cases

- Real-world systems where this pattern helps.

Interview One-Liner

One-line summary of the pattern.

UML / Class Diagram



chainOfResponsibilty — UML Class Diagram

---

mementoPattern

mementoPattern — Interview Reference

Intent

Provide a concise intent for the mementoPattern pattern.

Problem Statement

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.

Why Simple Code Fails

Often ad-hoc solutions (if/else, scattered constructors, tight coupling) make code hard to extend and test.

Solution Overview

Describe how the mementoPattern pattern solves the problem by providing structure and separation of concerns.

Minimal Java Example

Without Pattern

```
// Example not provided.
```

With Pattern

```
// Example not provided.
```

```
// Example not provided in original README.
```

Advantages

- Describe key advantages (decoupling, extensibility, reuse).

Disadvantages

- Describe trade-offs (complexity, indirection, overuse).

When NOT to Use

- Situations where the pattern is unnecessary.

Common Mistakes

- Frequent anti-patterns or pitfalls.

Framework / Library Usage

- Notes on common frameworks or language features.

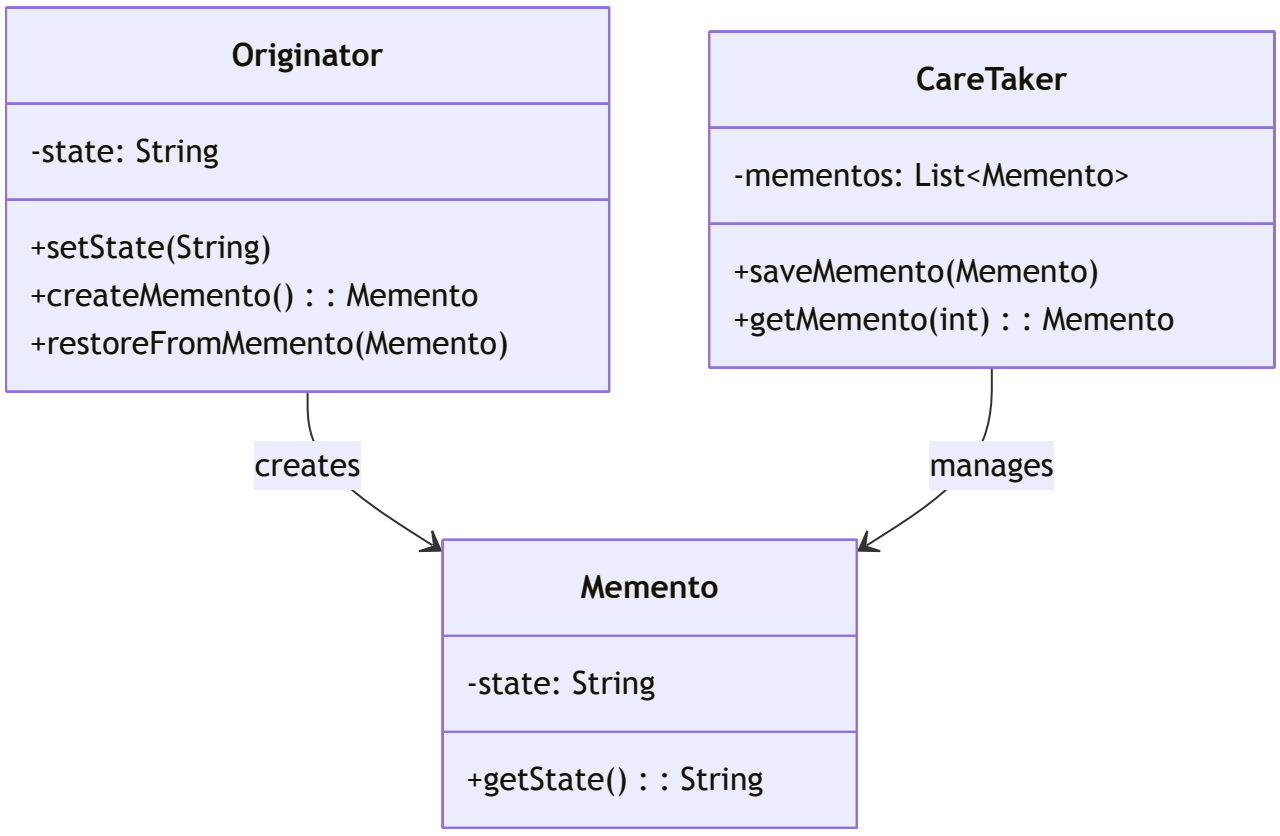
System Design Use Cases

- Real-world systems where this pattern helps.

Interview One-Liner

One-line summary of the pattern.

UML / Class Diagram



mementoPattern — UML Class Diagram

---

objectPoolPattern

objectPoolPattern — Interview Reference

Intent

Provide a concise intent for the objectPoolPattern pattern.

Problem Statement

Object Pool Pattern

Why Simple Code Fails

Often ad-hoc solutions (if/else, scattered constructors, tight coupling) make code hard to extend and test.

Solution Overview

Describe how the objectPoolPattern pattern solves the problem by providing structure and separation of concerns.

Minimal Java Example

Without Pattern

```
// Example not provided.
```

With Pattern

```
// Example not provided.

// Example not provided in original README.
```

Advantages

- Describe key advantages (decoupling, extensibility, reuse).

Disadvantages

- Describe trade-offs (complexity, indirection, overuse).

When NOT to Use

- Situations where the pattern is unnecessary.

Common Mistakes

- Frequent anti-patterns or pitfalls.

Framework / Library Usage

- Notes on common frameworks or language features.

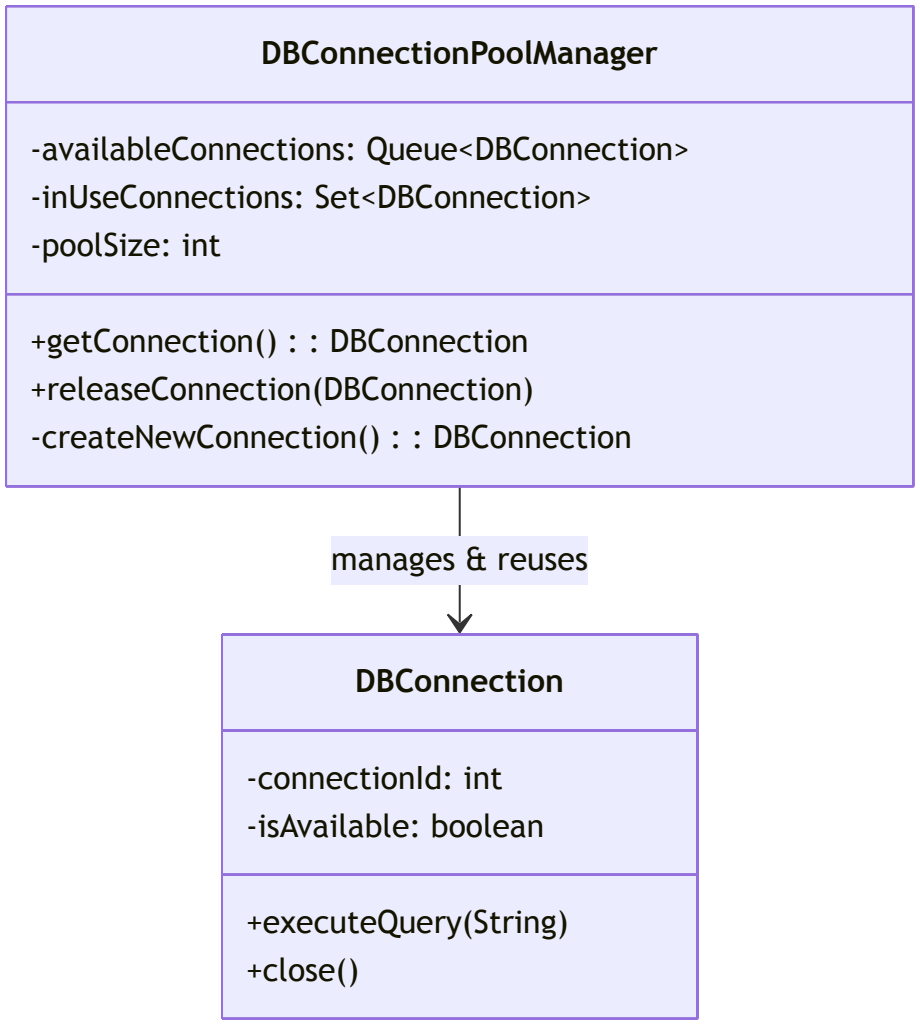
System Design Use Cases

- Real-world systems where this pattern helps.

Interview One-Liner

One-line summary of the pattern.

UML / Class Diagram



objectPoolPattern — UML Class Diagram

---

statePattern

statePattern — Interview Reference

Intent

Provide a concise intent for the statePattern pattern.

Problem Statement

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.

Why Simple Code Fails

Often ad-hoc solutions (if/else, scattered constructors, tight coupling) make code hard to extend and test.

Solution Overview

Describe how the statePattern pattern solves the problem by providing structure and separation of concerns.

Minimal Java Example

Without Pattern

```
// Example not provided.
```

With Pattern

```
// Example not provided.

// Example not provided in original README.
```

Advantages

- Describe key advantages (decoupling, extensibility, reuse).

Disadvantages

- Describe trade-offs (complexity, indirection, overuse).

When NOT to Use

- Situations where the pattern is unnecessary.

Common Mistakes

- Frequent anti-patterns or pitfalls.

Framework / Library Usage

- Notes on common frameworks or language features.

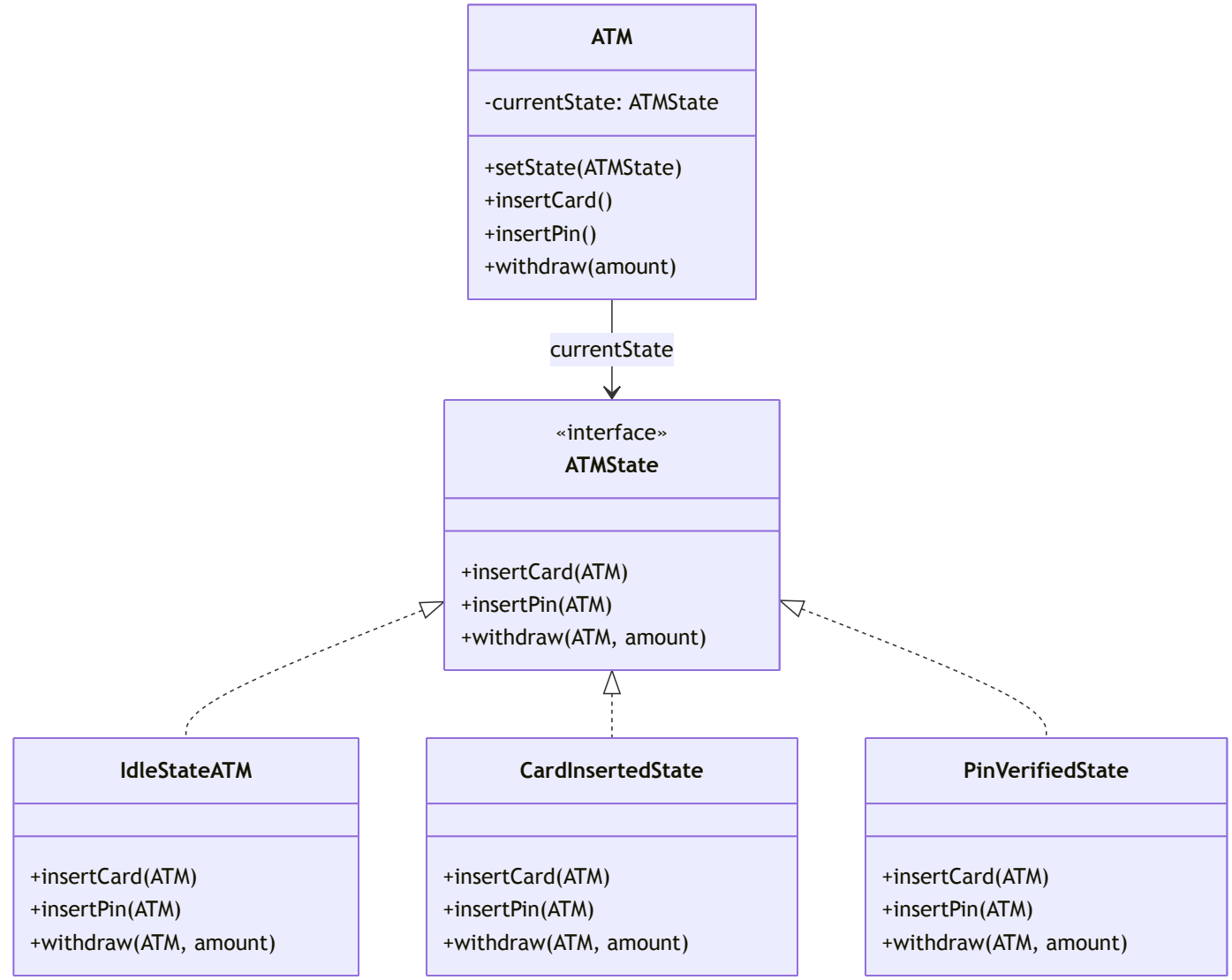
System Design Use Cases

- Real-world systems where this pattern helps.

Interview One-Liner

One-line summary of the pattern.

UML / Class Diagram



statePattern — UML Class Diagram

---

strategyPattern

Strategy Pattern — Interview Reference

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable.

Problem Statement

Multiple algorithms are implemented with conditionals in client code making it hard to extend or test.

Why Simple Code Fails

Conditionals couple clients to algorithm implementations; adding new strategies requires modifying client code.

Solution Overview

Extract algorithms into Strategy interfaces and inject appropriate Strategy into the Context.

UML Diagram

See `UML/ClassDiagram.md` and generated diagram at `build/diagrams/behavioralDesign_strategyPattern_UML_ClassDiagram.md.png` (if present).

Participants / Roles

- Strategy (interface)
- ConcreteStrategy (implementations)
- Context (uses Strategy)

Runtime Execution Flow

1. Client selects or injects a Strategy into Context
2. Context delegates algorithm calls to Strategy
3. Strategy executes algorithm and returns result

Minimal Java Example

Without Pattern

```
// Without Strategy: conditional selection of algorithm
if(mode=="zip") compressZip(data); else compressGzip(data);
```

With Pattern

```
// With Strategy: inject Compression strategy
Compressor c = new Compressor(new ZipCompression());
c.compress(data);

public interface Compression { byte[] compress(byte[] data); }
public class ZipCompression implements Compression { public byte[] compress(byte[] d){/*...*/} }
public class Compressor {
  private Compression compression;
  public Compressor(Compression c){ this.compression = c; }
  public void compress(byte[] data){ compression.compress(data); }
}
```

Advantages

- Algorithms are isolated and interchangeable
- Follows Open/Closed: add new strategies without changing clients

Disadvantages

- More classes/code
- Clients must be aware of Strategy selection

When NOT to Use

- If only one algorithm exists and unlikely to change

Common Mistakes

- Exposing too many strategy methods
- Putting selection logic back into Context

Framework / Library Usage

- Use DI frameworks (Spring) to wire strategies as beans and select by qualifier or profile

System Design Use Cases

- Sorting strategies, compression formats, payment routing strategies, retry/backoff policies

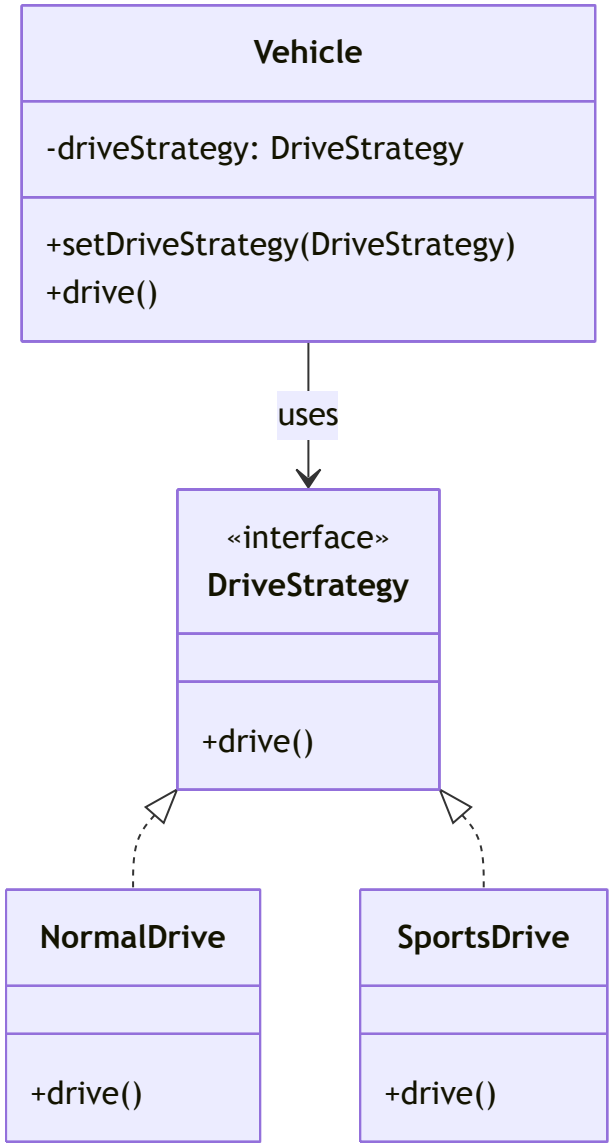
Interview One-Liner

Strategy encapsulates an interchangeable algorithm so clients can select behavior at runtime.

Common Interview Questions

- How to combine Strategy with Factory for selection?
- How to avoid strategy explosion?

UML / Class Diagram



strategyPattern — UML Class Diagram

---

templatePattern

templatePattern — Interview Reference

Intent

Provide a concise intent for the templatePattern pattern.

Problem Statement

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.

Why Simple Code Fails

Often ad-hoc solutions (if/else, scattered constructors, tight coupling) make code hard to extend and test.

Solution Overview

Describe how the templatePattern pattern solves the problem by providing structure and separation of concerns.

Minimal Java Example

Without Pattern

```
// Example not provided.
```

With Pattern

```
// Example not provided.

// Example not provided in original README.
```

Advantages

- Describe key advantages (decoupling, extensibility, reuse).

Disadvantages

- Describe trade-offs (complexity, indirection, overuse).

When NOT to Use

- Situations where the pattern is unnecessary.

Common Mistakes

- Frequent anti-patterns or pitfalls.

Framework / Library Usage

- Notes on common frameworks or language features.

System Design Use Cases

- Real-world systems where this pattern helps.

Interview One-Liner

One-line summary of the pattern.

UML / Class Diagram

 templatePattern UML

Creational Patterns

Problem domain: object creation mechanisms and lifecycle.

---

abstractFacotoryPattern

abstractFacotoryPattern — Interview Reference

Intent

Provide a concise intent for the abstractFacotoryPattern pattern.

Problem Statement

Abstract Factory Pattern

Why Simple Code Fails

Often ad-hoc solutions (if/else, scattered constructors, tight coupling) make code hard to extend and test.

Solution Overview

Describe how the abstractFacotoryPattern pattern solves the problem by providing structure and separation of concerns.

Minimal Java Example

Without Pattern

```
// Example not provided.
```

With Pattern

```
// Example not provided.

// Example not provided in original README.
```

Advantages

- Describe key advantages (decoupling, extensibility, reuse).

Disadvantages

- Describe trade-offs (complexity, indirection, overuse).

When NOT to Use

- Situations where the pattern is unnecessary.

Common Mistakes

- Frequent anti-patterns or pitfalls.

Framework / Library Usage

- Notes on common frameworks or language features.

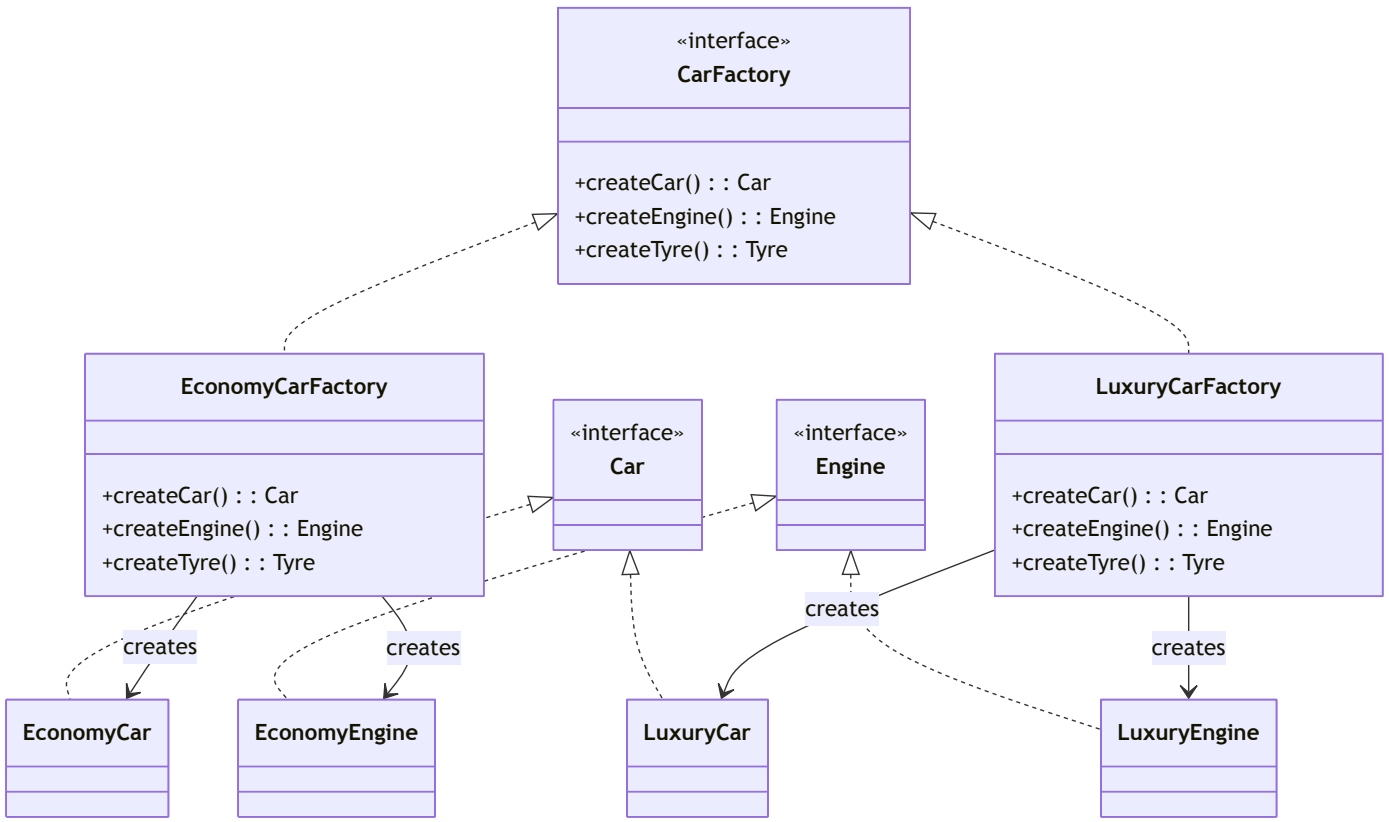
System Design Use Cases

- Real-world systems where this pattern helps.

Interview One-Liner

One-line summary of the pattern.

UML / Class Diagram



abstractFacotoryPattern — UML Class Diagram

---

abstractFactoryDesignPattern — Interview Reference

Intent

Provide a concise intent for the abstractFactoryDesignPattern pattern.

Problem Statement

No overview available.

Why Simple Code Fails

Often ad-hoc solutions (if/else, scattered constructors, tight coupling) make code hard to extend and test.

Solution Overview

Describe how the abstractFactoryDesignPattern pattern solves the problem by providing structure and separation of concerns.

Minimal Java Example

Without Pattern

```
// Example not provided.
```

With Pattern

```
// Example not provided.

// Example not available.
```

Advantages

- Describe key advantages (decoupling, extensibility, reuse).

Disadvantages

- Describe trade-offs (complexity, indirection, overuse).

When NOT to Use

- Situations where the pattern is unnecessary.

Common Mistakes

- Frequent anti-patterns or pitfalls.

Framework / Library Usage

- Notes on common frameworks or language features.

System Design Use Cases

- Real-world systems where this pattern helps.

Interview One-Liner

One-line summary of the pattern.

---

builderPattern

builderPattern — Interview Reference

Intent

Provide a concise intent for the builderPattern pattern.

Problem Statement

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.

Why Simple Code Fails

Often ad-hoc solutions (if/else, scattered constructors, tight coupling) make code hard to extend and test.

Solution Overview

Describe how the builderPattern pattern solves the problem by providing structure and separation of concerns.

Minimal Java Example

Without Pattern

```
// Example not provided.
```

With Pattern

```
// Example not provided.

// Example not provided in original README.
```

Advantages

- Describe key advantages (decoupling, extensibility, reuse).

Disadvantages

- Describe trade-offs (complexity, indirection, overuse).

When NOT to Use

- Situations where the pattern is unnecessary.

Common Mistakes

- Frequent anti-patterns or pitfalls.

Framework / Library Usage

- Notes on common frameworks or language features.

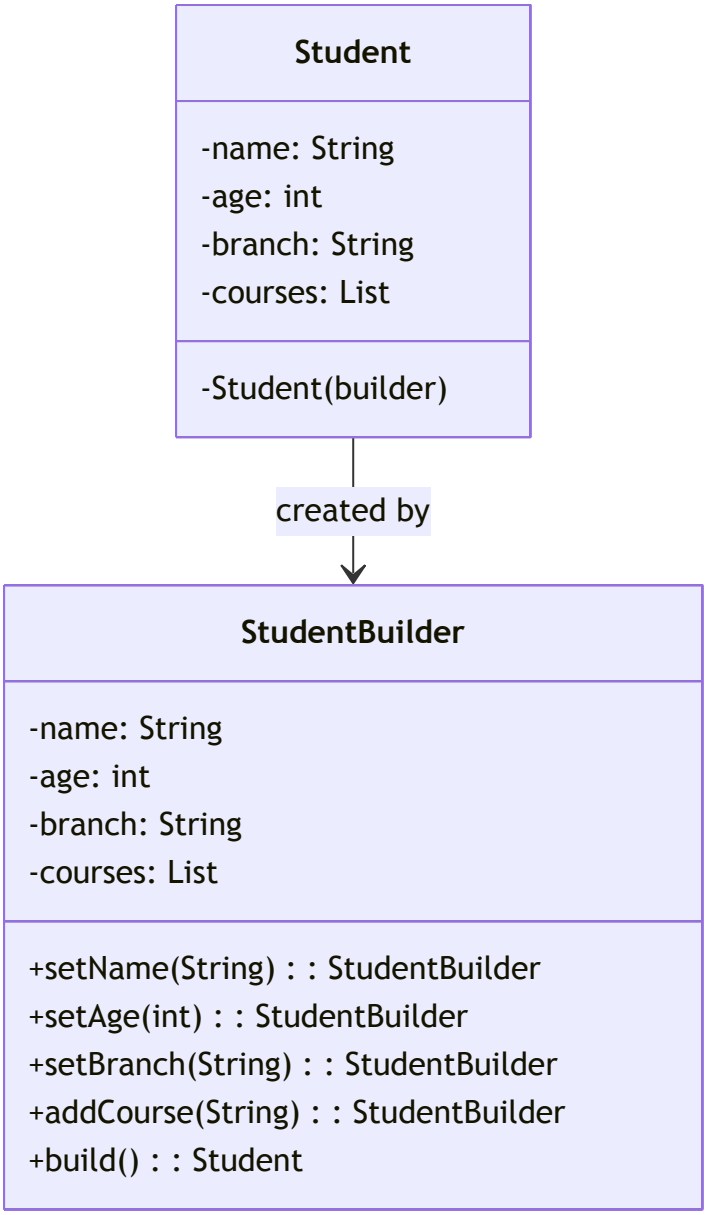
System Design Use Cases

- Real-world systems where this pattern helps.

Interview One-Liner

One-line summary of the pattern.

UML / Class Diagram



builderPattern — UML Class Diagram

---

factoryPattern

Factory Pattern — Interview Reference

Intent

Create objects without exposing instantiation logic; return interface types to callers.

Problem Statement

Client code depends on concrete classes; changes to construction scatter across codebase.

Why Simple Code Fails

Using `new` and `if-else` spreads knowledge of implementations, breaking Open/Closed and making tests harder.

Solution Overview

Introduce a Factory that encapsulates object creation behind a method returning a product interface.

UML Diagram

See UML/ClassDiagram.md and generated diagram at `build/diagrams/creationalDesign_factoryPattern_UML_ClassDiagram.md.png`.

Participants / Roles

- Product (interface)
- ConcreteProduct (implementations)
- Creator/Factory (declares factory method)
- ConcreteCreator (constructs ConcreteProduct)

Runtime Execution Flow

1. Client calls Factory.get(…)
2. Factory decides which ConcreteProduct to instantiate
3. Factory returns Product interface to client

Minimal Java Example

Without Pattern

```
// Without Factory: client instantiates concrete types directly
Shape s = new Circle();
s.draw();
```

With Pattern

```
// With Factory: use ShapeFactory to obtain interface
Shape s = ShapeFactory.getShape("circle");
s.draw();

public interface Shape { void draw(); }
public class Circle implements Shape { public void draw(){ /*...*/ } }
public class ShapeFactory {
  public static Shape getShape(String type){
    if("circle".equals(type)) return new Circle();
    return null;
  }
}
```

Advantages

- Decouples clients from concrete classes
- Centralizes creation logic
- Easier to add new product types

Disadvantages

- Extra indirection and classes
- Can be overused for trivial construction

When NOT to Use

When construction is simple and unlikely to change.

Common Mistakes

- Putting business logic in factories
- Returning concrete types instead of interfaces

Framework / Library Usage

- Spring: use `FactoryBean` or configuration classes
- Java: `Supplier<T>`, enums, or dependency injection containers

System Design Use Cases

- Plugin loaders, parser factories, message handler factories, UI component creation

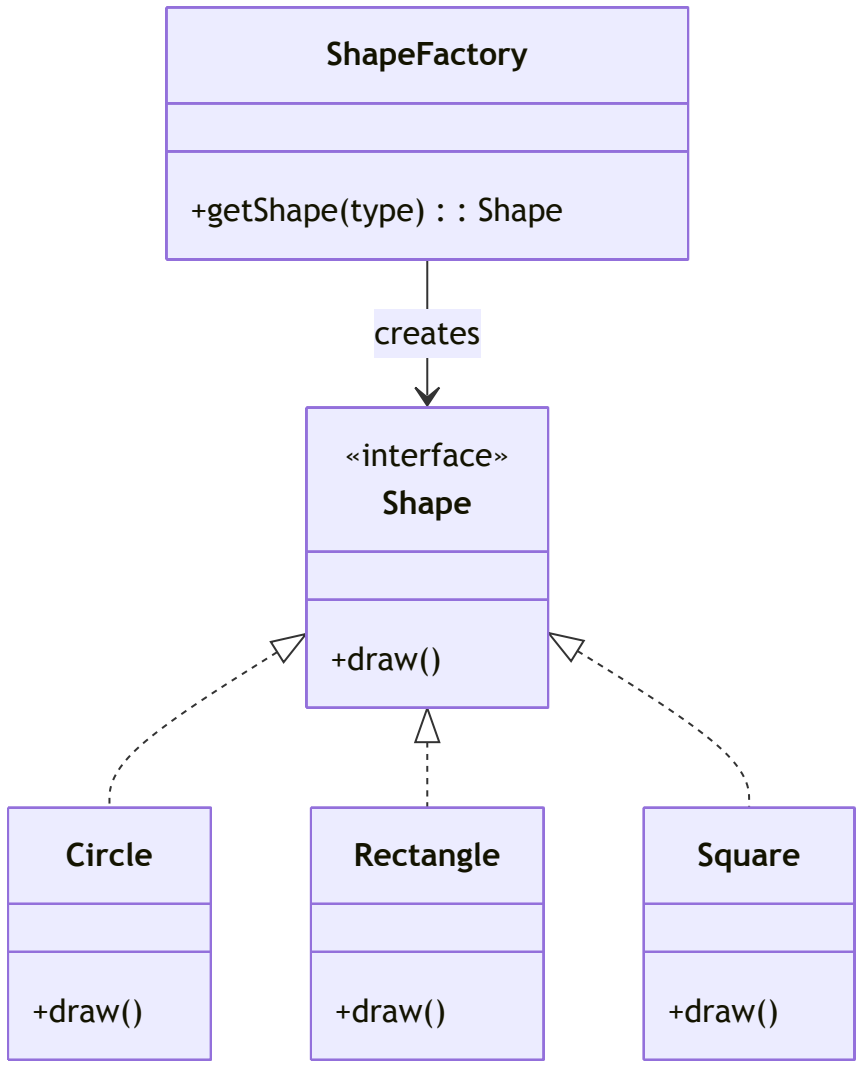
Interview One-Liner

Factory encapsulates object creation and returns abstractions to decouple clients.

Common Interview Questions

- How to avoid switch/if-else in factories?
- When choose Factory vs Builder?

UML / Class Diagram



factoryPattern — UML Class Diagram

---

prototypePattern

prototypePattern — Interview Reference

Intent

Provide a concise intent for the prototypePattern pattern.

Problem Statement

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.

Why Simple Code Fails

Often ad-hoc solutions (if/else, scattered constructors, tight coupling) make code hard to extend and test.

Solution Overview

Describe how the prototypePattern pattern solves the problem by providing structure and separation of concerns.

Minimal Java Example

Without Pattern

```
// Example not provided.
```

With Pattern

```
// Example not provided.

// Example not provided in original README.
```

Advantages

- Describe key advantages (decoupling, extensibility, reuse).

Disadvantages

- Describe trade-offs (complexity, indirection, overuse).

When NOT to Use

- Situations where the pattern is unnecessary.

Common Mistakes

- Frequent anti-patterns or pitfalls.

Framework / Library Usage

- Notes on common frameworks or language features.

System Design Use Cases

- Real-world systems where this pattern helps.

Interview One-Liner

One-line summary of the pattern.

UML / Class Diagram


prototypePattern UML

---

singletonDesignPattern

Singleton Pattern — Interview Reference

Intent

Ensure a class has only one instance and provide a global access point.

Problem Statement

Multiple parts of a system require a single shared resource/configuration; naive globals lead to uncontrolled instantiation and state issues.

Why Simple Code Fails

Using public static fields or unguarded lazy init leads to thread-safety bugs and testing difficulties.

Solution Overview

Expose a single instance through a controlled access method; handle lazy initialization safely and consider dependency injection alternatives.

UML Diagram

See `UML/ClassDiagram.md` and generated diagram at `build/diagrams/creationalDesign_singletonDesignPattern_UML_ClassDiagram.md.png` (if present).

Participants / Roles

- Singleton (the single instance holder)

Runtime Execution Flow

1. Client requests Singleton.getInstance()
2. getInstance() either returns existing instance or initializes it safely

3. Client uses shared instance

Minimal Java Example

Without Pattern

```java
// Without Singleton: using public static field or multiple instances
Config a = new Config();
Config b = new Config();
```

With Pattern

```java
// With Singleton: single shared instance
Config cfg = Config.getInstance();

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

Advantages

- Controlled access to single instance
- Useful for shared resources

Disadvantages

- Global state hampers testability
- Can hide dependencies and increase coupling

When NOT to Use

- When DI can provide better lifecycle and scoping
- For per-request or short-lived objects

Common Mistakes

- Not handling thread-safety
- Using Singleton as a catch-all global

Framework / Library Usage

- Use dependency injection (Spring beans with singleton scope) rather than hand-rolled singletons where possible.

System Design Use Cases

- Configuration managers, caches, connection pools (careful: pools are better as separate objects), logging facades

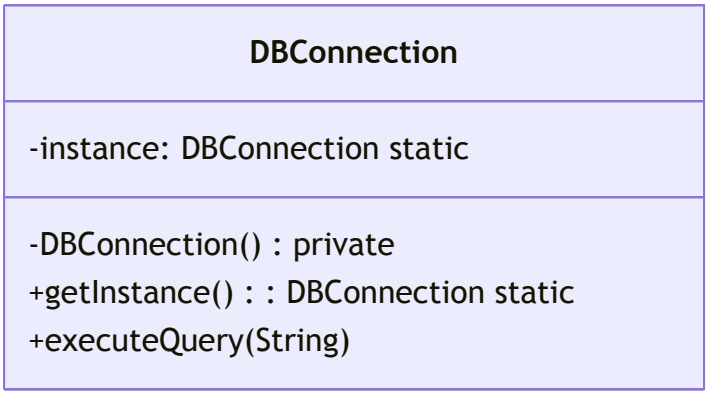
Interview One-Liner

Singleton ensures a single instance and global access, but can introduce global state and testability problems.

Common Interview Questions

- How to write thread-safe lazy singleton in Java?
- When are singletons harmful?

UML / Class Diagram



singletonDesignPattern — UML Class Diagram

Structural Patterns

Problem domain: composing objects and classes for larger structures.

AdapterDesignPattern

Adapter Pattern — Interview Reference

Intent

Convert the interface of a class into another interface clients expect, allowing incompatible interfaces to work together.

Problem Statement

Existing classes provide useful behavior but expose an incompatible interface required by clients.

Why Simple Code Fails

Copying or modifying existing classes to fit new interfaces introduces duplication and brittle changes.

Solution Overview

Provide an Adapter that wraps the adaptee and implements the target interface, translating calls.

UML Diagram

See `UML/ClassDiagram.md` and generated diagram at `build/diagrams/StructuralDesign_AdapterDesignPattern_UML_ClassDiagram.md.png` (if present).

Participants / Roles

- Target: desired interface
- Adaptee: existing class with incompatible interface
- Adapter: implements Target and delegates to Adaptee

Runtime Execution Flow

1. Client uses Target interface
2. Adapter implements Target and translates calls to Adaptee
3. Adaptee performs the work; Adapter returns results to client

Minimal Java Example

Without Pattern

```
// Example not provided.
```

With Pattern

```
// Example not provided.

public interface MediaPlayer { void play(String file); }
public class LegacyPlayer { public void start(String f){ /*...*/ } }
public class PlayerAdapter implements MediaPlayer {
  private LegacyPlayer adaptee;
  public PlayerAdapter(LegacyPlayer l){ this.adaptee = l; }
  public void play(String file){ adaptee.start(file); }
}
```

Advantages

- Reuse existing code without modification
- Keeps client code clean and stable

Disadvantages

- Can add extra layers and indirection

When NOT to Use

- When you can change both sides or when interfaces are simple to align

Common Mistakes

- Over-adapting: creating adapters for trivial interface differences
- Hiding performance or semantic differences behind adapter

Framework / Library Usage

- Adapters commonly appear as wrappers, facades, or compatibility layers in middleware

System Design Use Cases

- Integrating legacy systems, API version adapters, driver or protocol translation

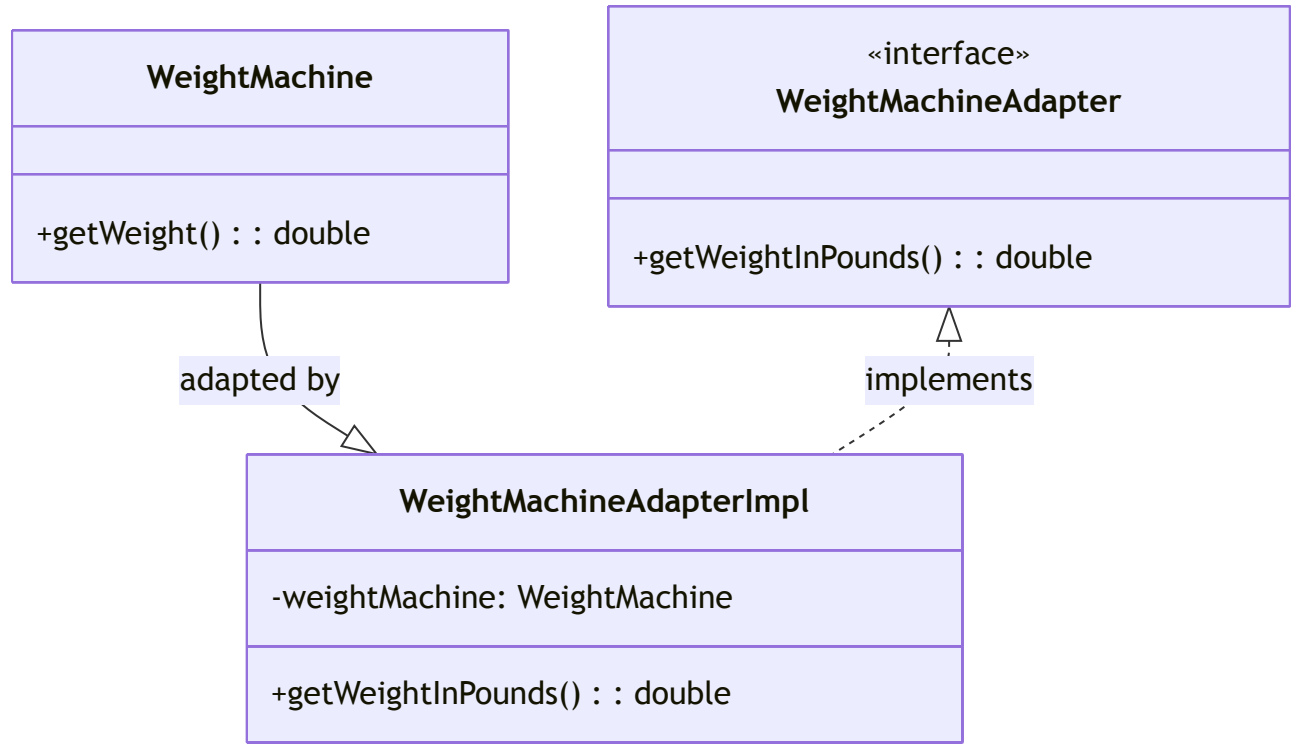
Interview One-Liner

Adapter wraps an incompatible interface to match the client's expected interface without changing original code.

Common Interview Questions

- Adapter vs Facade — differences?
- When to prefer object adapter over class adapter?

UML / Class Diagram



AdapterDesignPattern — UML Class Diagram

DecoratorDesign

DecoratorDesign — Interview Reference

Intent

Provide a concise intent for the DecoratorDesign pattern.

Problem Statement

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

Why Simple Code Fails

Often ad-hoc solutions (if/else, scattered constructors, tight coupling) make code hard to extend and test.

Solution Overview

Describe how the DecoratorDesign pattern solves the problem by providing structure and separation of concerns.

Minimal Java Example

Without Pattern

```
// Example not provided.
```

With Pattern

```
// Example not provided.

// Example not provided in original README.
```

Advantages

- Describe key advantages (decoupling, extensibility, reuse).

Disadvantages

- Describe trade-offs (complexity, indirection, overuse).

When NOT to Use

- Situations where the pattern is unnecessary.

Common Mistakes

- Frequent anti-patterns or pitfalls.

Framework / Library Usage

- Notes on common frameworks or language features.

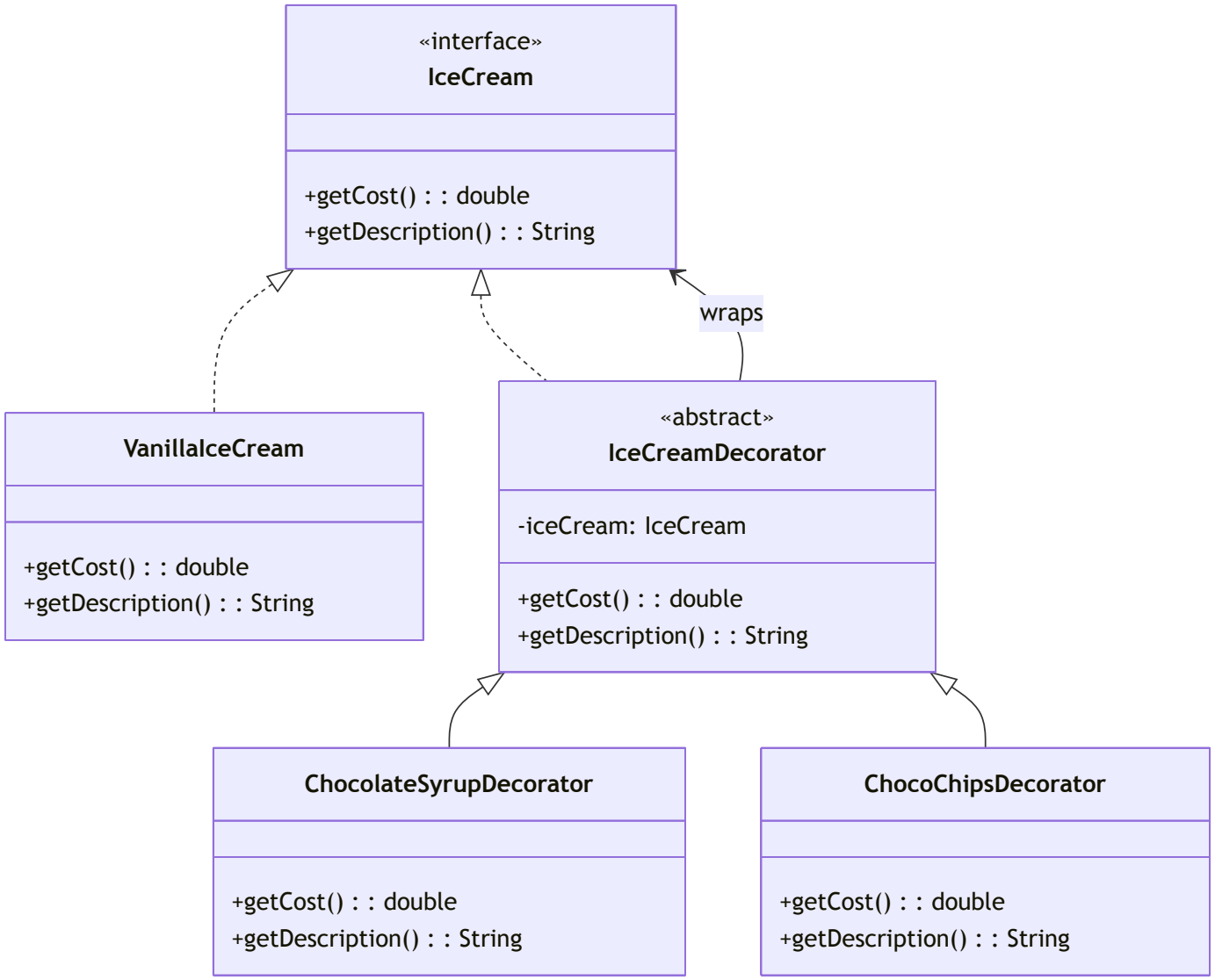
System Design Use Cases

- Real-world systems where this pattern helps.

Interview One-Liner

One-line summary of the pattern.

UML / Class Diagram



DecoratorDesign — UML Class Diagram

ProxyPattern

ProxyPattern — Interview Reference

Intent

Provide a concise intent for the ProxyPattern pattern.

Problem Statement

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.

Why Simple Code Fails

Often ad-hoc solutions (if/else, scattered constructors, tight coupling) make code hard to extend and test.

Solution Overview

Describe how the ProxyPattern pattern solves the problem by providing structure and separation of concerns.

Minimal Java Example

Without Pattern

```
// Example not provided.
```

With Pattern

```
// Example not provided.

// Example not provided in original README.
```

Advantages

- Describe key advantages (decoupling, extensibility, reuse).

Disadvantages

- Describe trade-offs (complexity, indirection, overuse).

When NOT to Use

- Situations where the pattern is unnecessary.

Common Mistakes

- Frequent anti-patterns or pitfalls.

Framework / Library Usage

- Notes on common frameworks or language features.

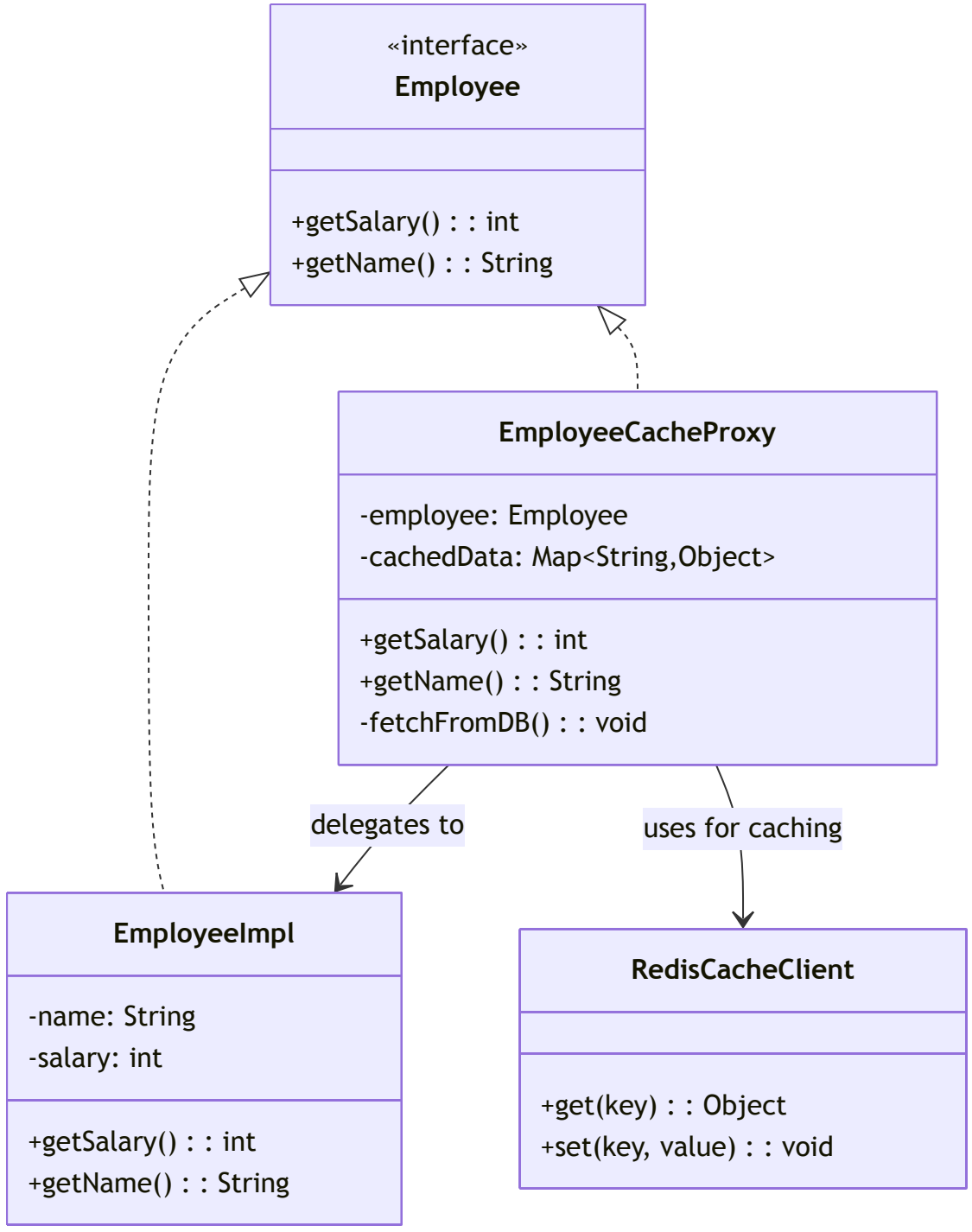
System Design Use Cases

- Real-world systems where this pattern helps.

Interview One-Liner

One-line summary of the pattern.

UML / Class Diagram



ProxyPattern — UML Class Diagram

---

bridgePattern

bridgePattern — Interview Reference

Intent

Provide a concise intent for the bridgePattern pattern.

Problem Statement

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.

Why Simple Code Fails

Often ad-hoc solutions (if/else, scattered constructors, tight coupling) make code hard to extend and test.

Solution Overview

Describe how the bridgePattern pattern solves the problem by providing structure and separation of concerns.

Minimal Java Example

Without Pattern

```
// Example not provided.
```

With Pattern

```
// Example not provided.

// Example not provided in original README.
```

Advantages

- Describe key advantages (decoupling, extensibility, reuse).

Disadvantages

- Describe trade-offs (complexity, indirection, overuse).

When NOT to Use

- Situations where the pattern is unnecessary.

Common Mistakes

- Frequent anti-patterns or pitfalls.

Framework / Library Usage

- Notes on common frameworks or language features.

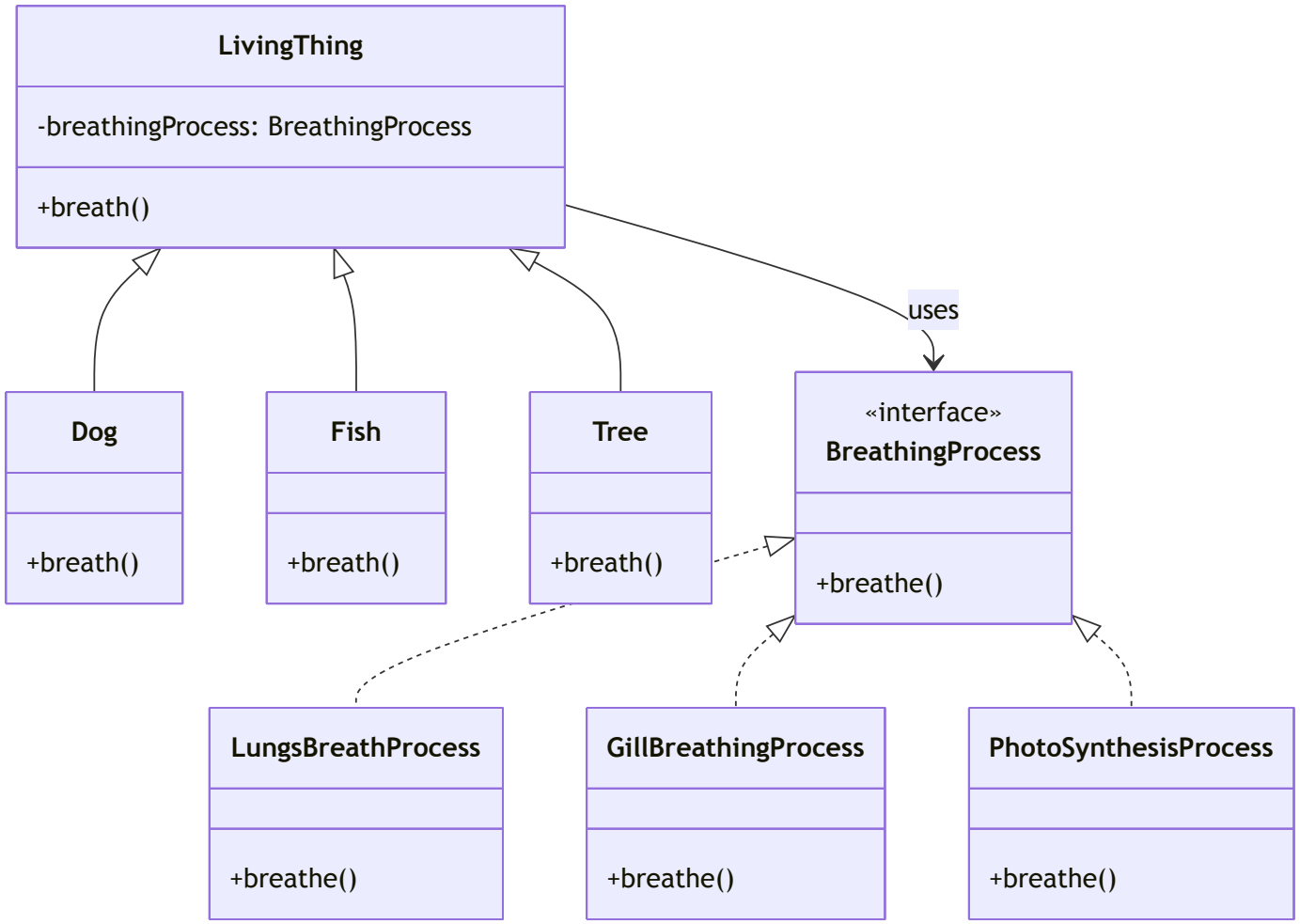
System Design Use Cases

- Real-world systems where this pattern helps.

Interview One-Liner

One-line summary of the pattern.

UML / Class Diagram



bridgePattern — UML Class Diagram

---

compositePattern

compositePattern — Interview Reference

Intent

Provide a concise intent for the compositePattern pattern.

Problem Statement

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.

Why Simple Code Fails

Often ad-hoc solutions (if/else, scattered constructors, tight coupling) make code hard to extend and test.

Solution Overview

Describe how the compositePattern pattern solves the problem by providing structure and separation of concerns.

Minimal Java Example

Without Pattern

```
// Example not provided.
```

With Pattern

```
// Example not provided.
```

```
// Example not provided in original README.
```

Advantages

- Describe key advantages (decoupling, extensibility, reuse).

Disadvantages

- Describe trade-offs (complexity, indirection, overuse).

When NOT to Use

- Situations where the pattern is unnecessary.

Common Mistakes

- Frequent anti-patterns or pitfalls.

Framework / Library Usage

- Notes on common frameworks or language features.

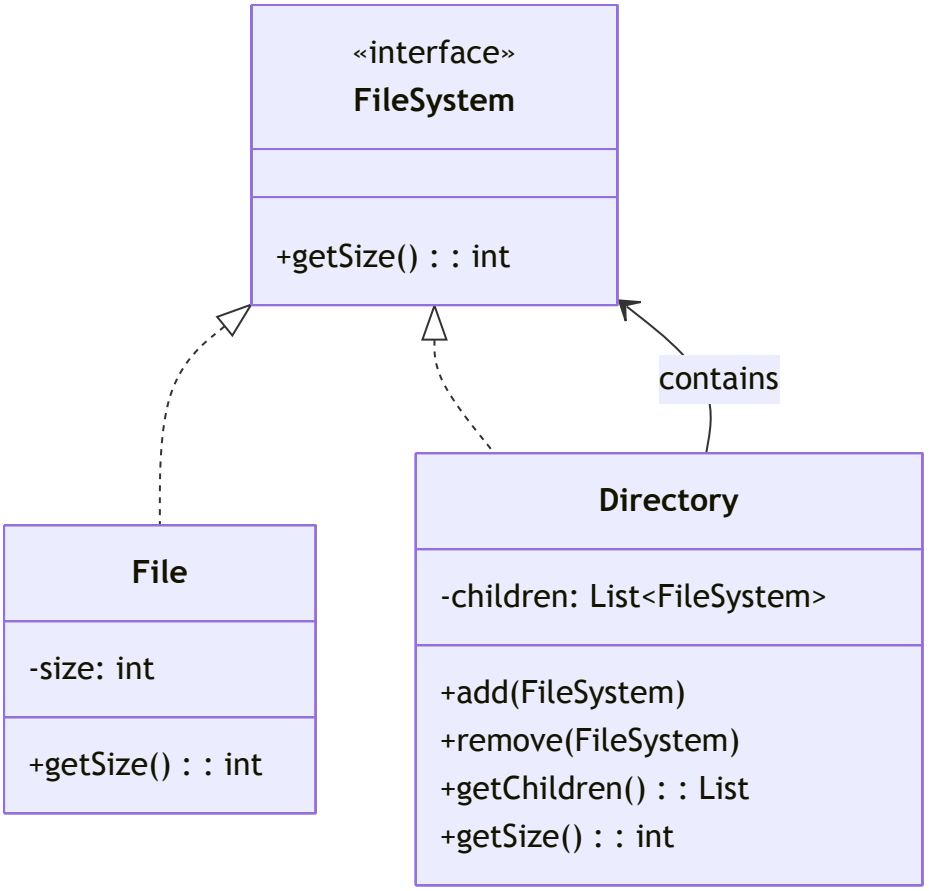
System Design Use Cases

- Real-world systems where this pattern helps.

Interview One-Liner

One-line summary of the pattern.

UML / Class Diagram



compositePattern — UML Class Diagram

---

flyweightPattern

flyweightPattern — Interview Reference

Intent

Provide a concise intent for the flyweightPattern pattern.

Problem Statement

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

Why Simple Code Fails

Often ad-hoc solutions (if/else, scattered constructors, tight coupling) make code hard to extend and test.

Solution Overview

Describe how the flyweightPattern pattern solves the problem by providing structure and separation of concerns.

Minimal Java Example

Without Pattern

```
// Example not provided.
```

With Pattern

```
// Example not provided.
```

```
// Example not provided in original README.
```

Advantages

- Describe key advantages (decoupling, extensibility, reuse).

Disadvantages

- Describe trade-offs (complexity, indirection, overuse).

When NOT to Use

- Situations where the pattern is unnecessary.

Common Mistakes

- Frequent anti-patterns or pitfalls.

Framework / Library Usage

- Notes on common frameworks or language features.

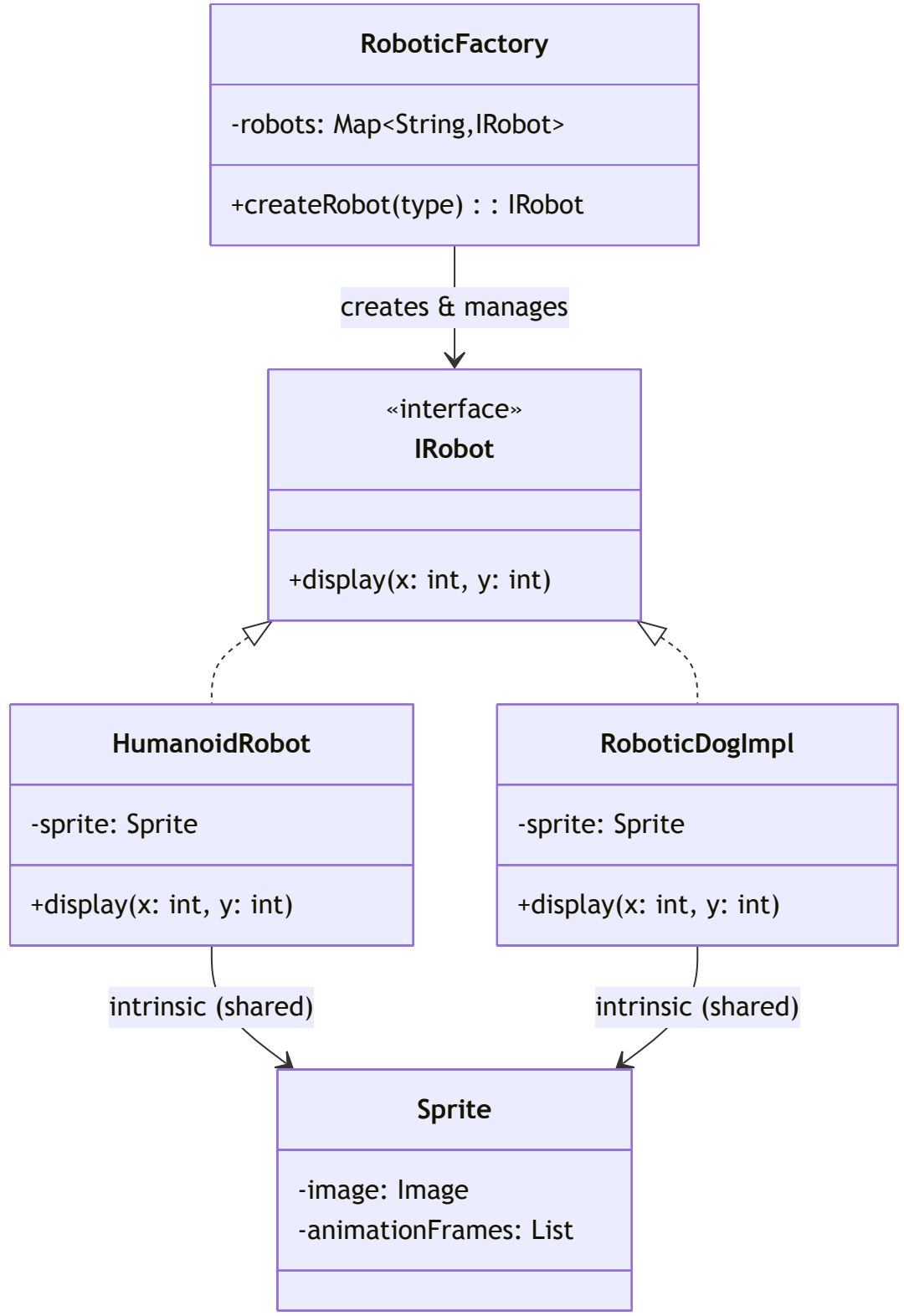
System Design Use Cases

- Real-world systems where this pattern helps.

Interview One-Liner

One-line summary of the pattern.

UML / Class Diagram



flyweightPattern — UML Class Diagram