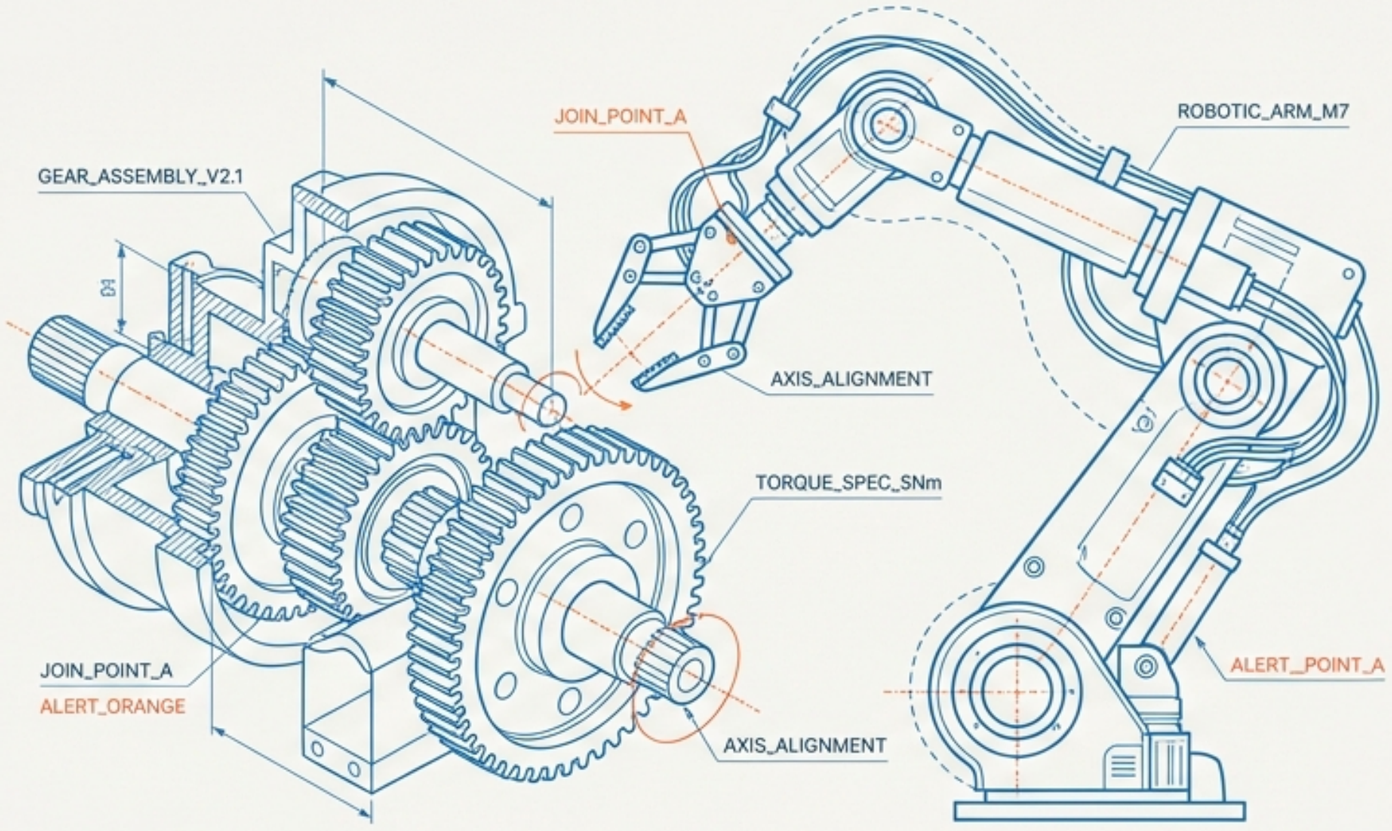


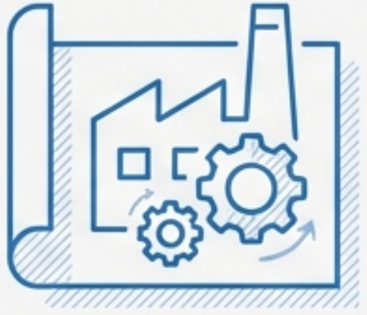
Factory, Abstract Factory, and Singleton Patterns

SWE 4743 Object-Oriented Design | Spring 2026

KSU SOFTWARE ENGINEERING

Learning Goals

Control Creation



Use **Factories** to decouple usage from instantiation. Stop gluing code to concrete classes.

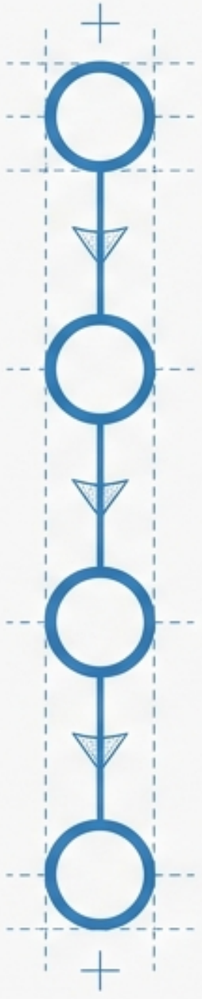
Control Quantity

Use **Singleton** to manage instance manage instance count (and understand the risks of global state). ⚠

Minimize Impact

Apply patterns to isolate change. Master C# thread safety C# thread safety and delegates.

Agenda



1. Why Object Creation Matters

The cost of the ‘new’ keyword.

2. The Factory Trio

Simple Factory vs. Factory Method vs. Abstract Factory.

3. Singleton & Thread Safety

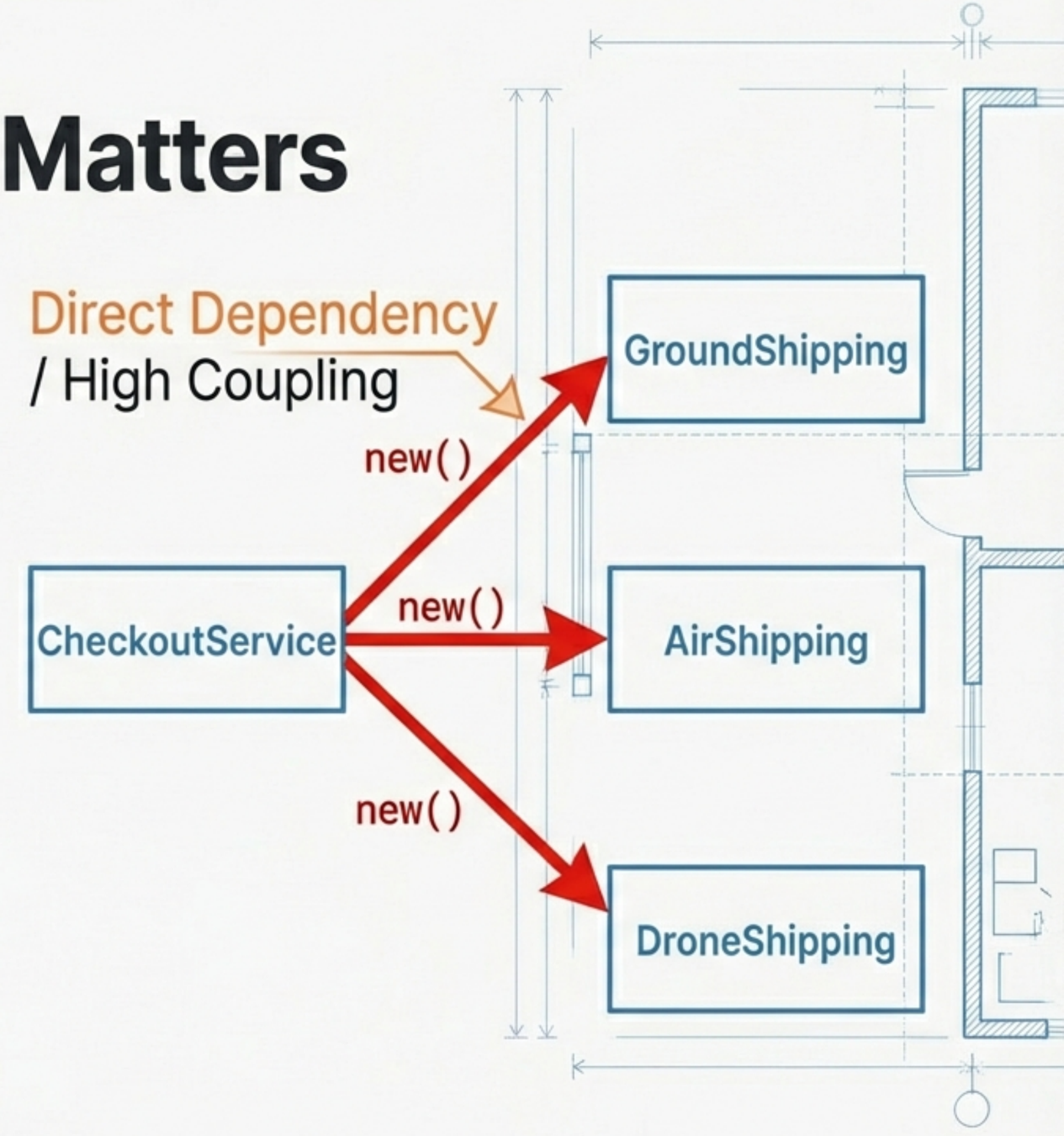
Managing global state and race conditions.

4. Review & Exam Guide

Tradeoffs and common pitfalls.

Why Object Creation Matters

- **The Problem:** "`new ClassName()`" glues code to a specific implementation.
- **The Goal:** Depend on abstractions (`IShippingStrategy`), not concrete classes.
- **The Context:** We will use a ShippingStrategy example throughout this lecture.



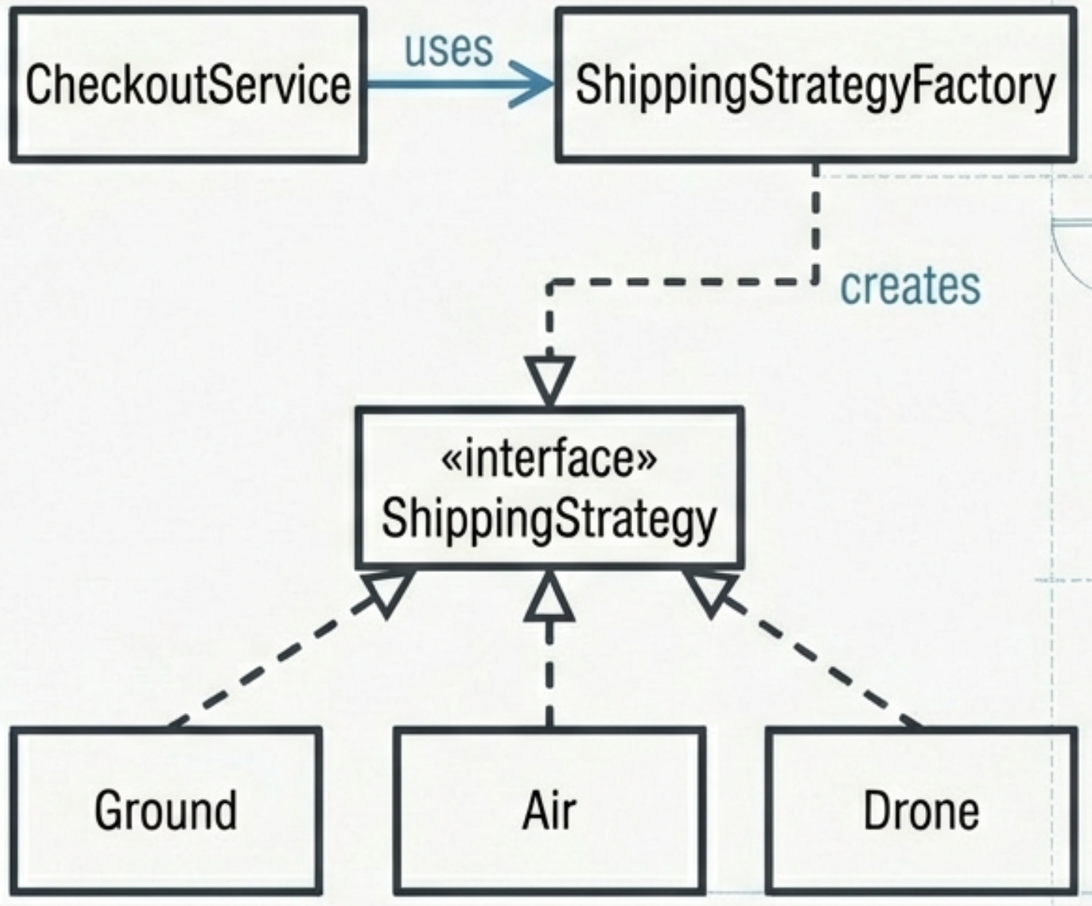
Factory Choice Matrix

The Question	The Pattern
Given a runtime key (string/enum), what do I create?	**Simple Factory** Centralizes branching logic.
Which subclass should decide what gets created?	**Factory Method Pattern** Creation varies by creator subtype.
How do I create a compatible family of objects?	**Abstract Factory** Guarantees consistency (e.g., Themes).

Simple Factory: Intent & Structure

Centralizing the decision of “What to create”.

- **Problem:** Business logic cluttered with `if/switch` statements.
- **Solution:** Move creation logic to one central method/class.
- **Result:** Client depends on “`ShippingStrategy`” (Interface), not concrete types.



C# Implementation: The Registry Pattern

```
public sealed class ShippingStrategyFactory
{
    // The Registry: Replaces switch statements
    private readonly Dictionary<string, Func<ShippingStrategy>> _registry =
        new(StringComparer.OrdinalIgnoreCase)
        {
            ["ground"] = () => new GroundShippingStrategy(),
            ["air"]    = () => new AirShippingStrategy(),
            ["drone"] = () => new DroneShippingStrategy()
        };

    public ShippingStrategy Create(string mode)
    {
        if (!_registry.TryGetValue(mode, out var creator))
            throw new NotSupportedException($"Unsupported: {mode}");

        return creator(); // Invoke the func
    }
}
```

Refactoring Win

Before: Large switch inside CheckoutService.
After: '_factory.Create(mode)'.

Simple Factory in Practice

Configuration

The Composition Root owns config loading. Business logic never parses CLI args or Env Vars.

Industry Example

Payment Providers (Stripe vs. Adyen vs. Mock). Switch providers based on environment (Dev/Prod).

Testing

Easy to mock. Pass a factory that returns a 'MockStrategy'.

SOLID Alignment

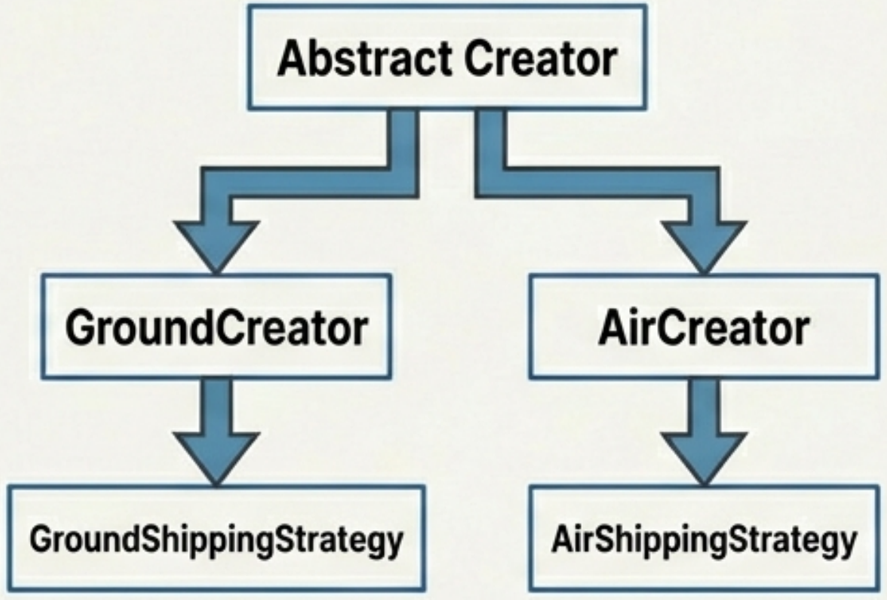
- **SRP:** Creation separated from usage policy.
- **OCP:** Register new types without touching the client.

The Delta: Simple Factory vs. Factory Method

Simple Factory

One selector class makes the decision.

Factory Method Pattern



A Class Hierarchy makes the decision.

Key Change: Logic moves from a central selector class to individual subclasses.

C# Implementation: Subclass Decision Making

The Abstraction

```
public abstract class ShippingStrategyCreator
{
    // The Extension Point
    protected abstract ShippingStrategy
       CreateStrategy();

    public decimal QuoteShipping(decimal weight)
    {
        // Base class doesn't know what it creates
        var strategy = CreateStrategy();
        return strategy.Calculate(weight);
    }
}
```

The Extension Point

Base class logic relies on abstract method

The Implementation

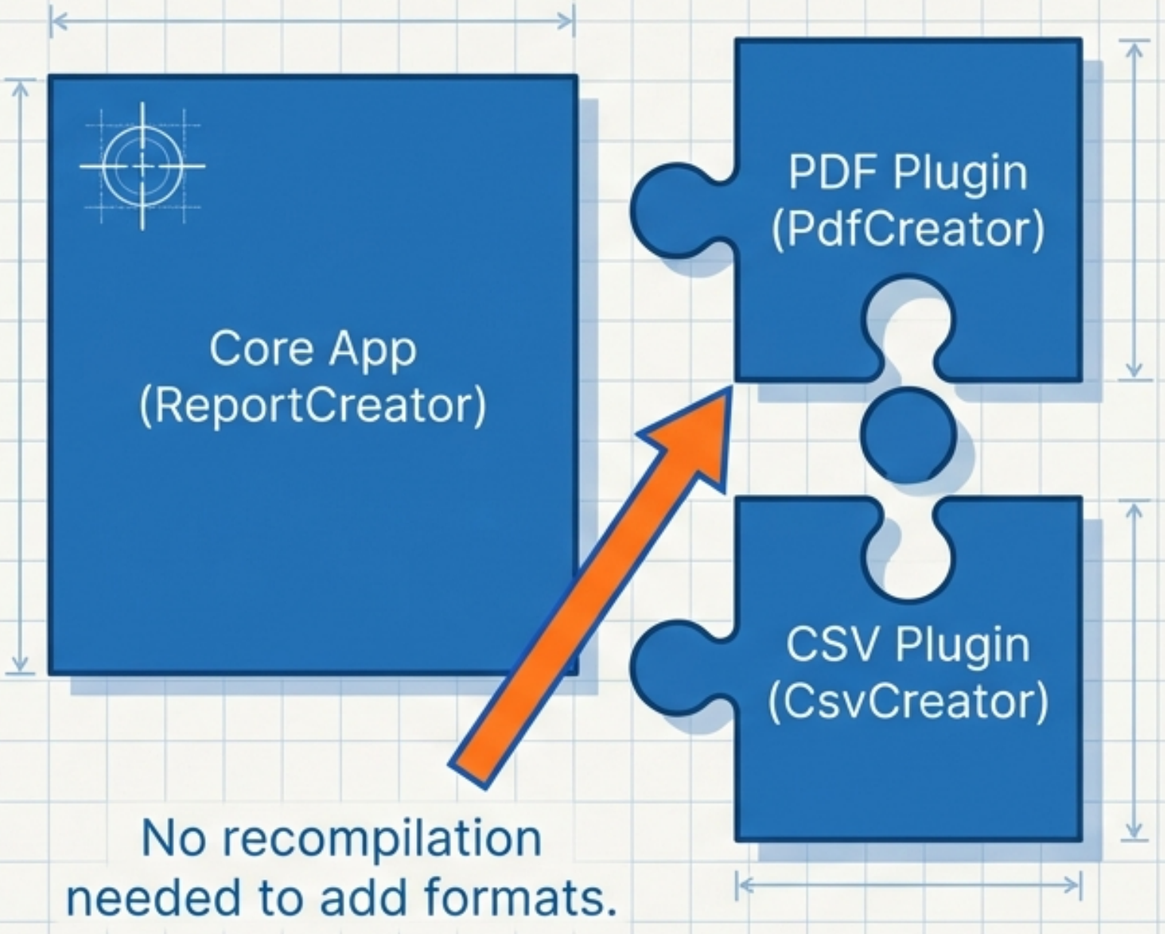
```
public sealed class GroundShippingCreator :
    ShippingStrategyCreator
{
    // The Decision
    protected override ShippingStrategy
       CreateStrategy()
    {
        return new GroundShippingStrategy();
    }
}
```

The Decision

Concrete implementation provided

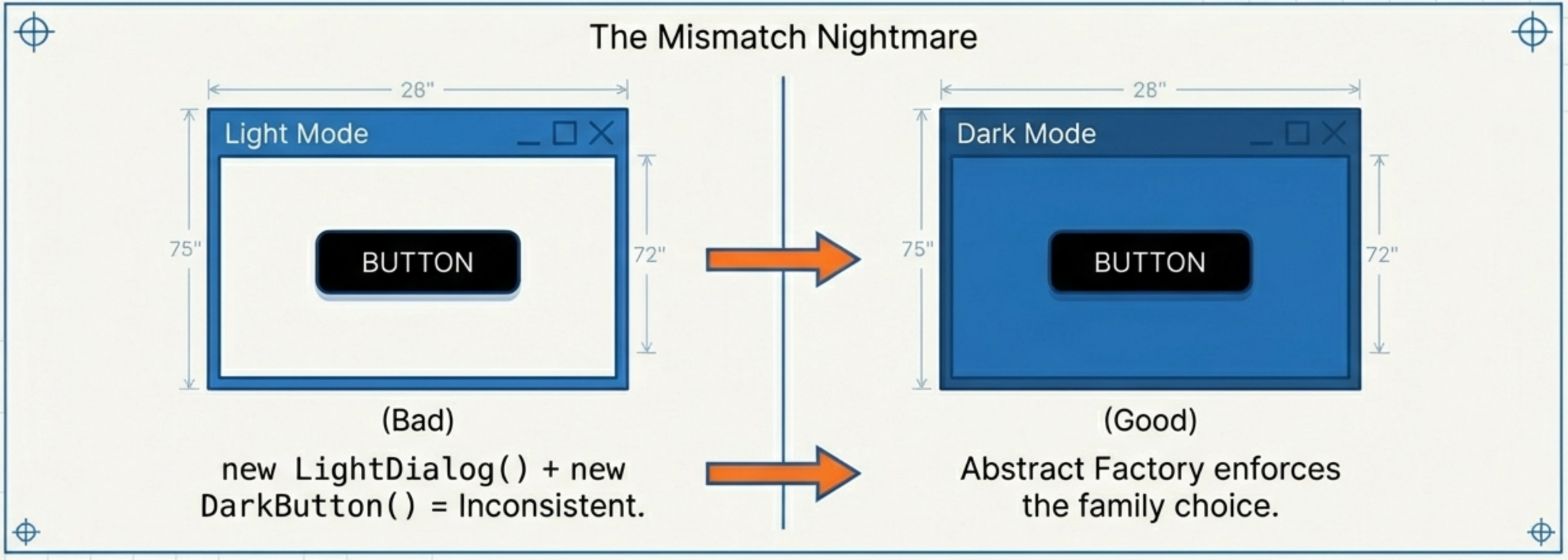
Use Case: Plugins & Extension Points

- The “Switch” didn’t disappear; it moved to the Composition Root.
- We now map inputs to Creators (e.g., `Dictionary<string, Func<Creator>>`).
- Ideal for Frameworks and Plugins.



Abstract Factory: The Problem of Families

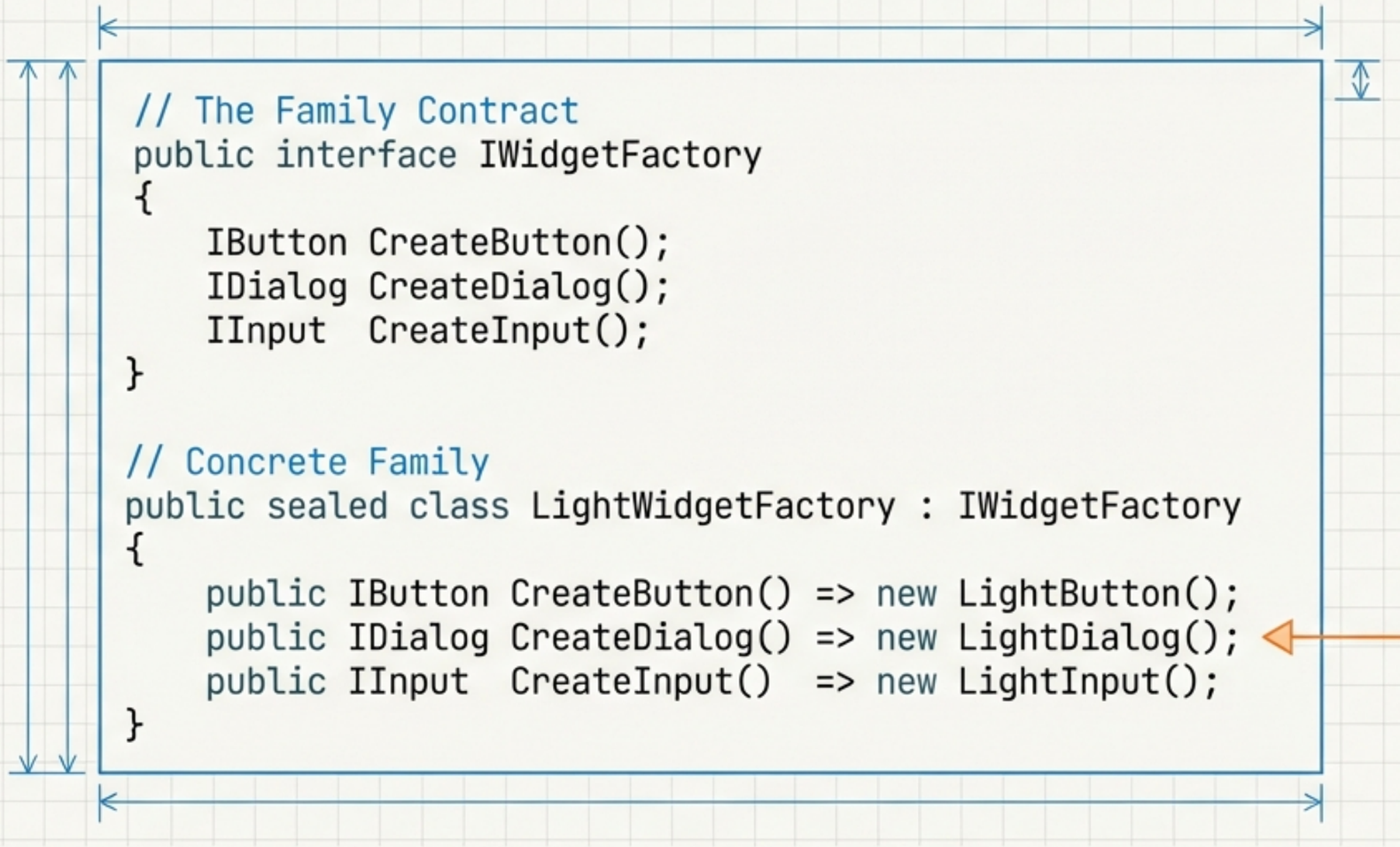
Ensuring compatibility across related objects.



Intent: Create families of related objects without specifying concrete classes.

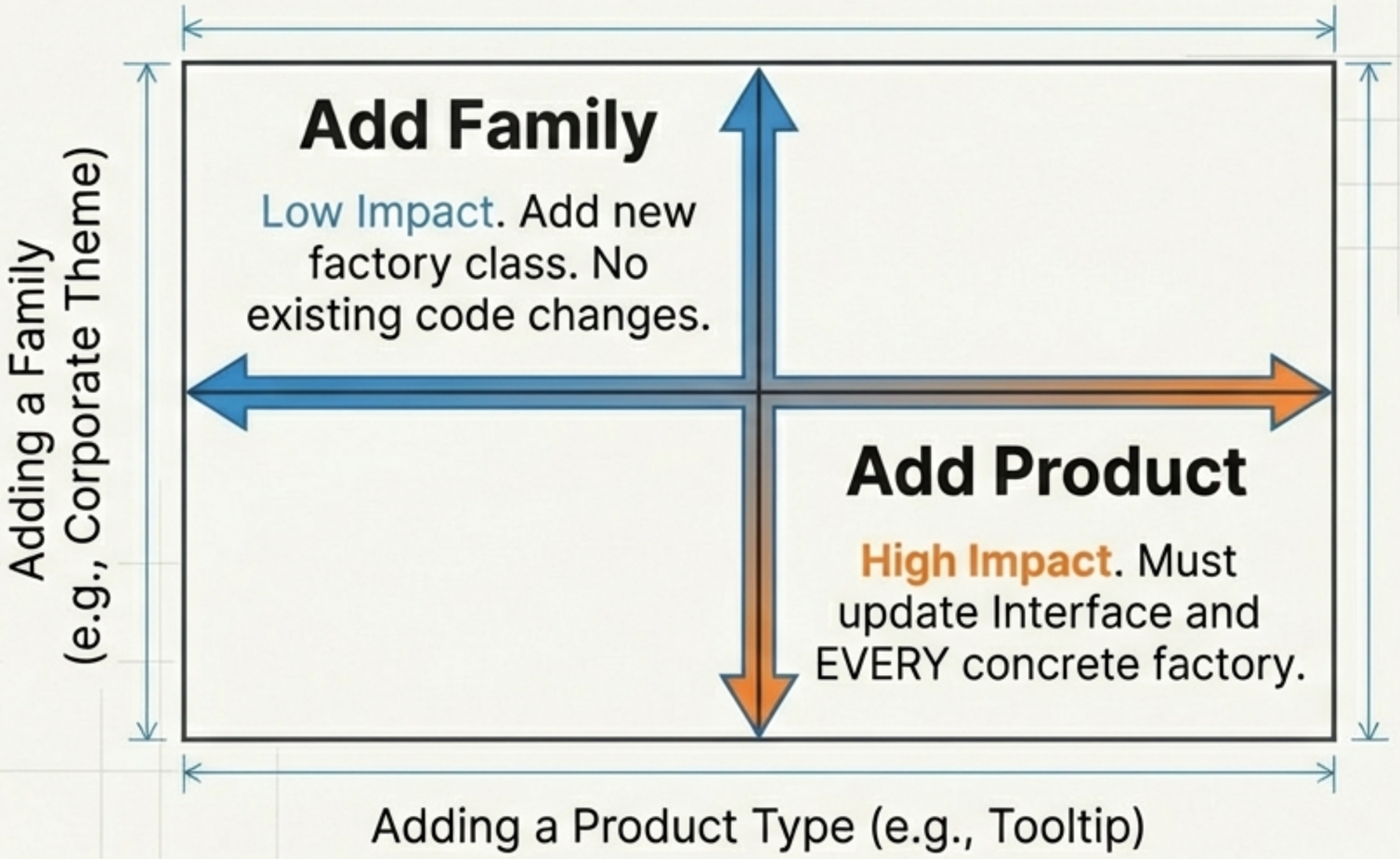
C# Implementation: Enforcing Consistency

```
// The Family Contract
public interface IWidgetFactory
{
    IButton CreateButton();
    IDialog CreateDialog();
    IInput  CreateInput();
}

// Concrete Family
public sealed class LightWidgetFactory : IWidgetFactory
{
    public IButton CreateButton() => new LightButton();
    public IDialog CreateDialog() => new LightDialog();
    public IInput  CreateInput()  => new LightInput();
}
```



Client Code: settingsScreen.Render(new LightWidgetFactory()) -> **Guaranteed consistency.**

Tradeoff Analysis: Families vs. Products



Key takeaway: Abstract Factory is great for new themes, painful for evolving widget sets.

Singleton Pattern: The Safety Warning

Intent

Ensure exactly one instance exists + Provide global access.

The Risk

Hidden Global State.

- Any code can change the state.
- Dependencies are hidden (not in the constructor).
- Makes testing difficult.

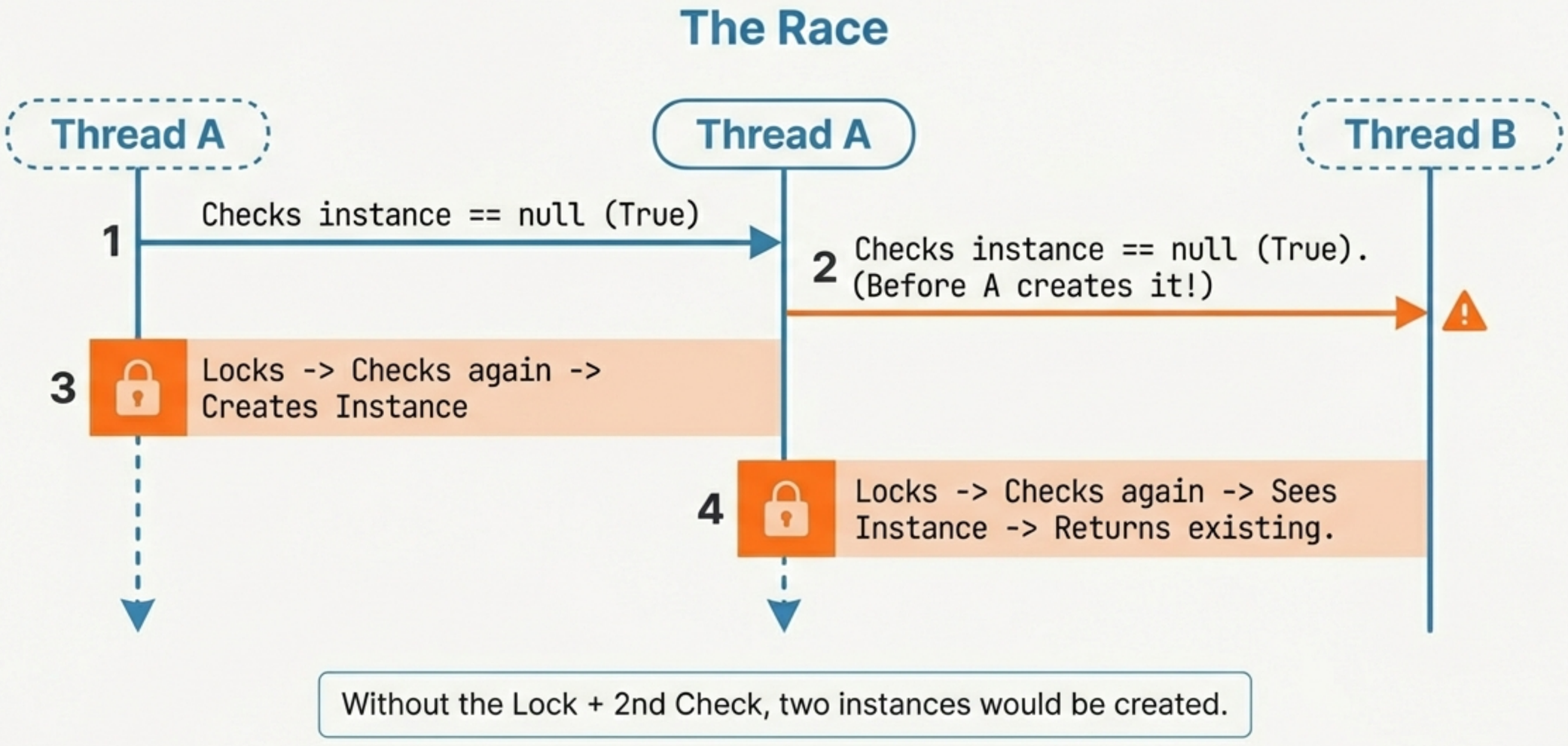
C# Implementation: Double-Checked Locking

```
public sealed class AppLogger
{
    private static volatile AppLogger? _instance;
    private static readonly object SyncRoot = new();

    private AppLogger() { } // Private Constructor

    public static AppLogger Instance
    {
        get
        {
            if (_instance is null) // 1st Check
            {
                lock (SyncRoot)
                {
                    if (_instance is null) // 2nd Check
                        _instance = new AppLogger();
                }
            }
            return _instance;
        }
    }
}
```

Java Differences

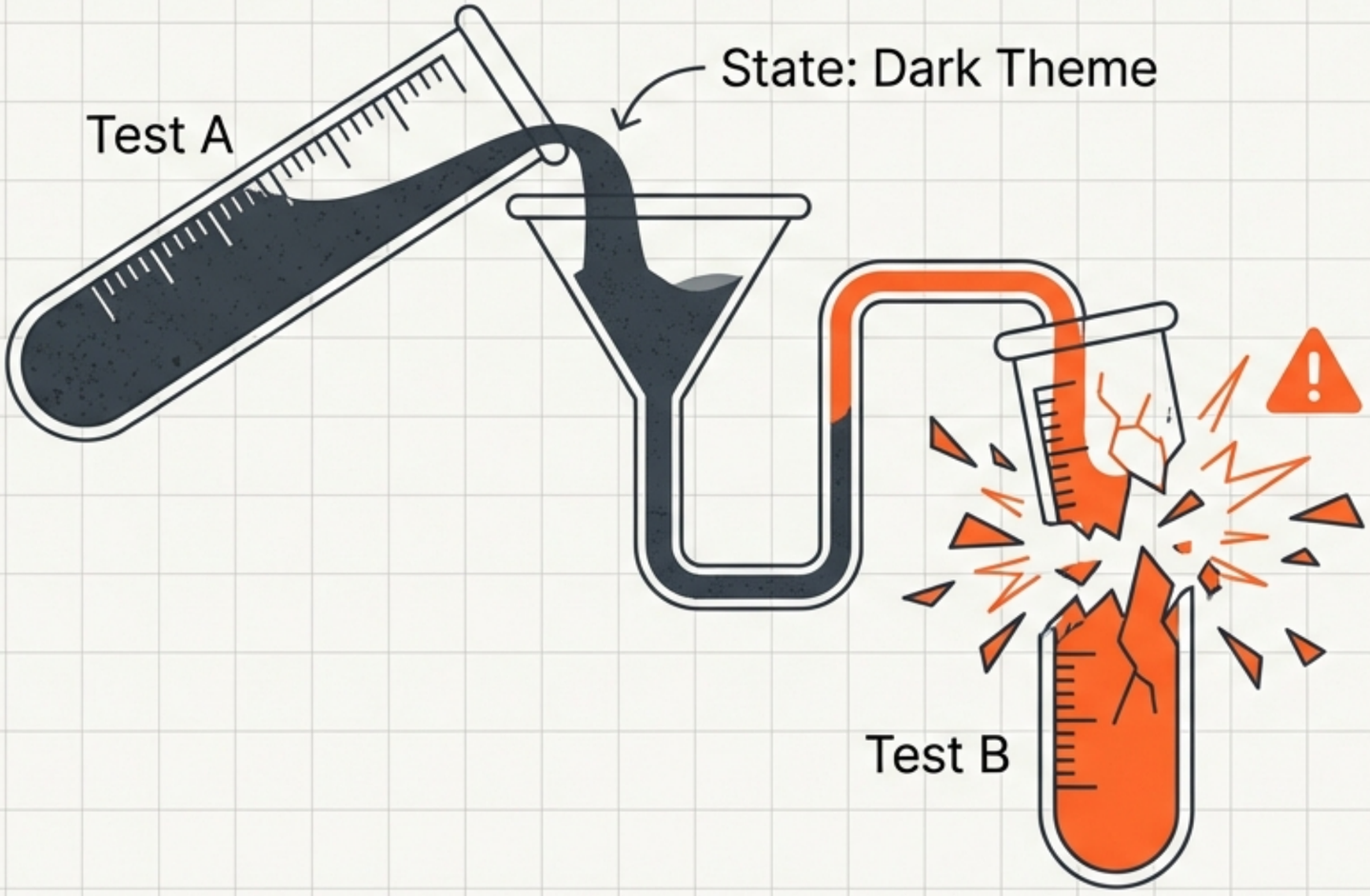
Uses 'synchronized' keyword or 'enum Singleton' (preferred).

Anatomy of a Race Condition



Without the Lock + 2nd Check, two instances would be created.

The Testing Nightmare



The Smell

Tests fail only when run in a specific order.

The Cause

Singleton keeps state across test cases.

The Fix

Use Dependency Injection (Container Managed Singleton) instead of static classes.

Pattern Comparison Cheat Sheet

Pattern	Structure	Best Fit
Simple Factory JetBrains Mono, Charcoal	One Central Selector Inter, Charcoal	Runtime Config Keys Inter, Charcoal
Factory Method JetBrains Mono, Charcoal	Subclass Decision Inter, Charcoal	Frameworks / Plugins Inter, Charcoal
Abstract Factory JetBrains Mono, Charcoal	Family Factories Inter, Charcoal	Themes / Cross-Platform Inter, Charcoal
Singleton JetBrains Mono, Charcoal	Static Instance Inter, Charcoal	Logging / Config (Immutable) Inter, Charcoal

Summary & Study Guide

Exam Questions

- Why is adding a product type hard in Abstract Factory?
  ↘ Answer: Breaks the interface.
- When to switch from Simple Factory to Factory Method?
  ↘ Answer: When variation needs subclass hooks.
- Why does Singleton break test isolation?
  ↘ Answer: Shared state persists across tests.

Prefer Dependency Injection over Singletons. Prefer Simple Factories over complex hierarchies until you need the power.