SWE 4743:
Object-Oriented Design

Jeff Adkisson

UML Class Diagramming

Agenda

- **Core Concepts and Layout**
  Understanding the three-zone structure of a UML class box (Name, Members, and Methods) and basic annotations like interfaces and enumerations.
- **Defining Relationships**
  Exploring inheritance (is-a), interface realization, and dependencies to represent how classes interact and utilize one another.
- **Structural Connections**
  Distinguishing between long-lived associations and the lifetime-specific differences of aggregation (shared) versus composition (owned).
- **Design Constraints and Multiplicity**
  Implementing access modifiers and multiplicity rules in code to enforce structural design decisions.
- **Practical Application**
  Best practices for using Mermaid to create versionable, AI-friendly diagrams that evolve alongside the codebase.

Lecture Material

See **03-uml-class-diagramming.md** in the Presentations folder for these topics:

- Core Concepts and Layout
- Design constrains (access modifiers)
- Multiplicity

UML Class Diagrams as a Design Tool

Class diagrams are valuable for thinking about design, communicating ideas, and defending decisions.

- Focus on understanding, not drawing mechanics or Mermaid syntax or choice of drawing tools.
- Use for discussion, design, and explanation – not creating giant diagrams no one uses.

What a Class Diagram Represents

Class diagrams are static, *not behavioral*. They do **not** show execution flow. Instead, they show what exists in the system and how pieces relate structurally.

- Static structure of a system
  - Classes, interfaces, and relationships
  - A snapshot of design intent

Why Diagrams Before Code

Diagrams help you decide who owns what, who depends on whom, and what abstractions exist *before code hardens*.

Code is harder to read later than it is to write now. Always be thinking about the next developer (which might be future-you wishing you had been more expressive the first time).

- Force explicit design decisions
  - Reveal responsibilities early
  - Cheaper to change in design than in code

Communication Over Implementation

Code contains history, edge cases, and implementation details.

Diagrams strip that away so a group can focus on structure and intent.

- Shared visual language
  - Less noise than code
  - Ideal for teaching and review

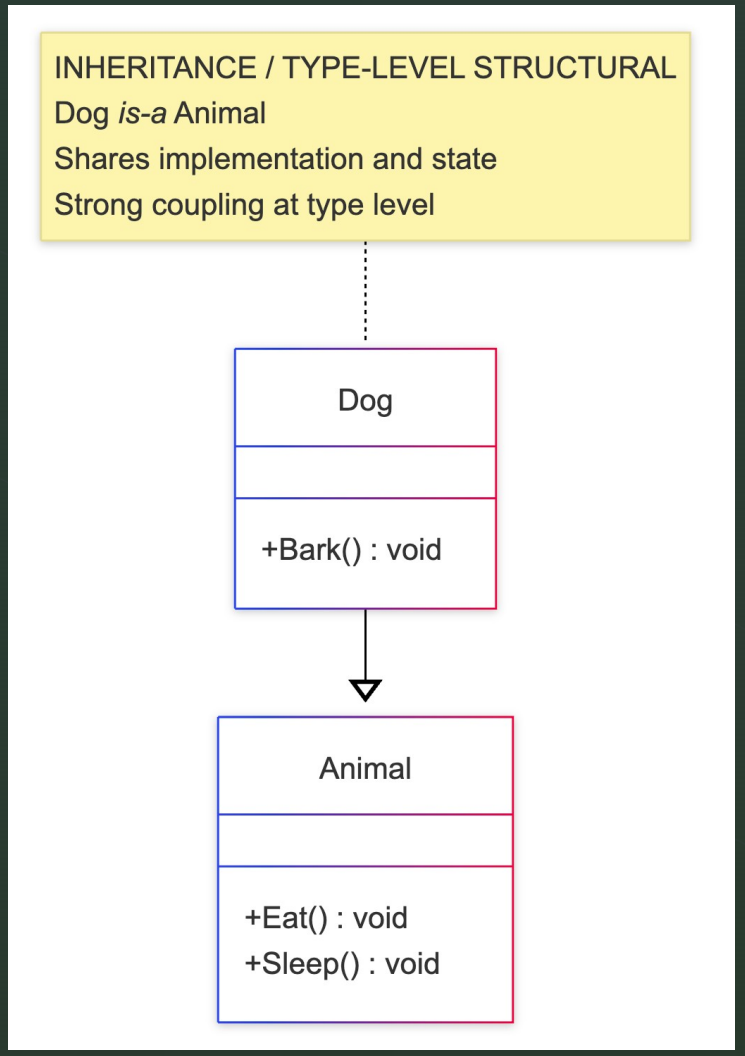
Type-Level Structural Dependencies

Structural dependencies represent long-lived relationships.

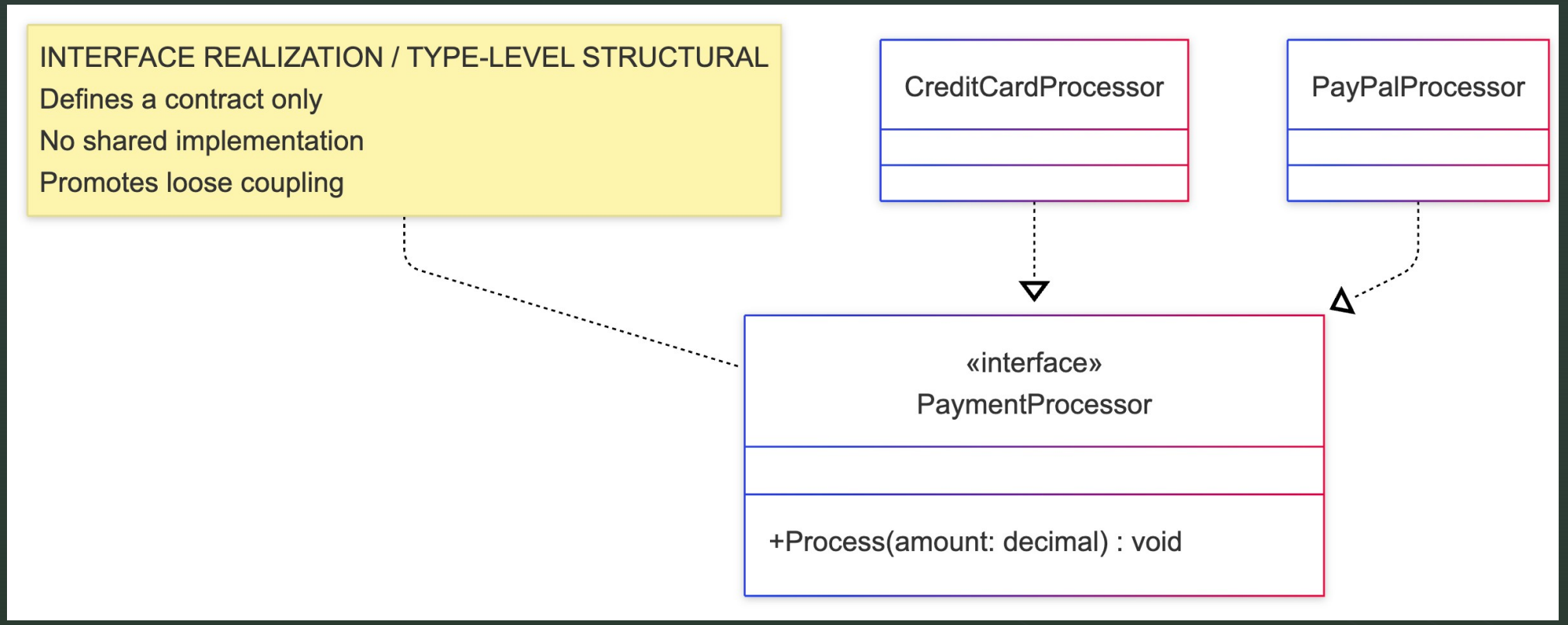
These show what objects know about and rely on over time, not just during a single method call.

- Associations and ownership
  - Inheritance relationships
  - Interface realizations

Inheritance / Type-Level Structural



Interface Realization / Type-Level Structural



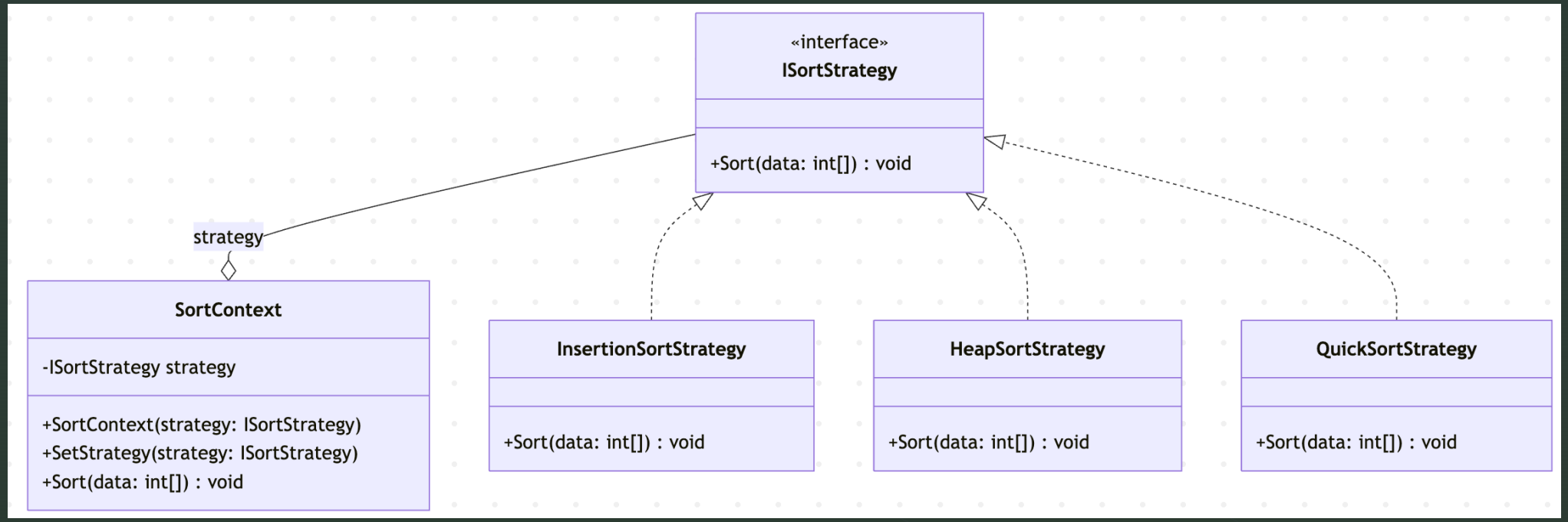
Interfaces vs Concrete Dependencies

Interfaces describe capability *without commitment*.

Diagrams make this distinction obvious and visible.

- Interfaces express contracts
  - Concrete classes express decisions
  - Prefer depending on abstractions

Interfaces vs Concrete Dependencies



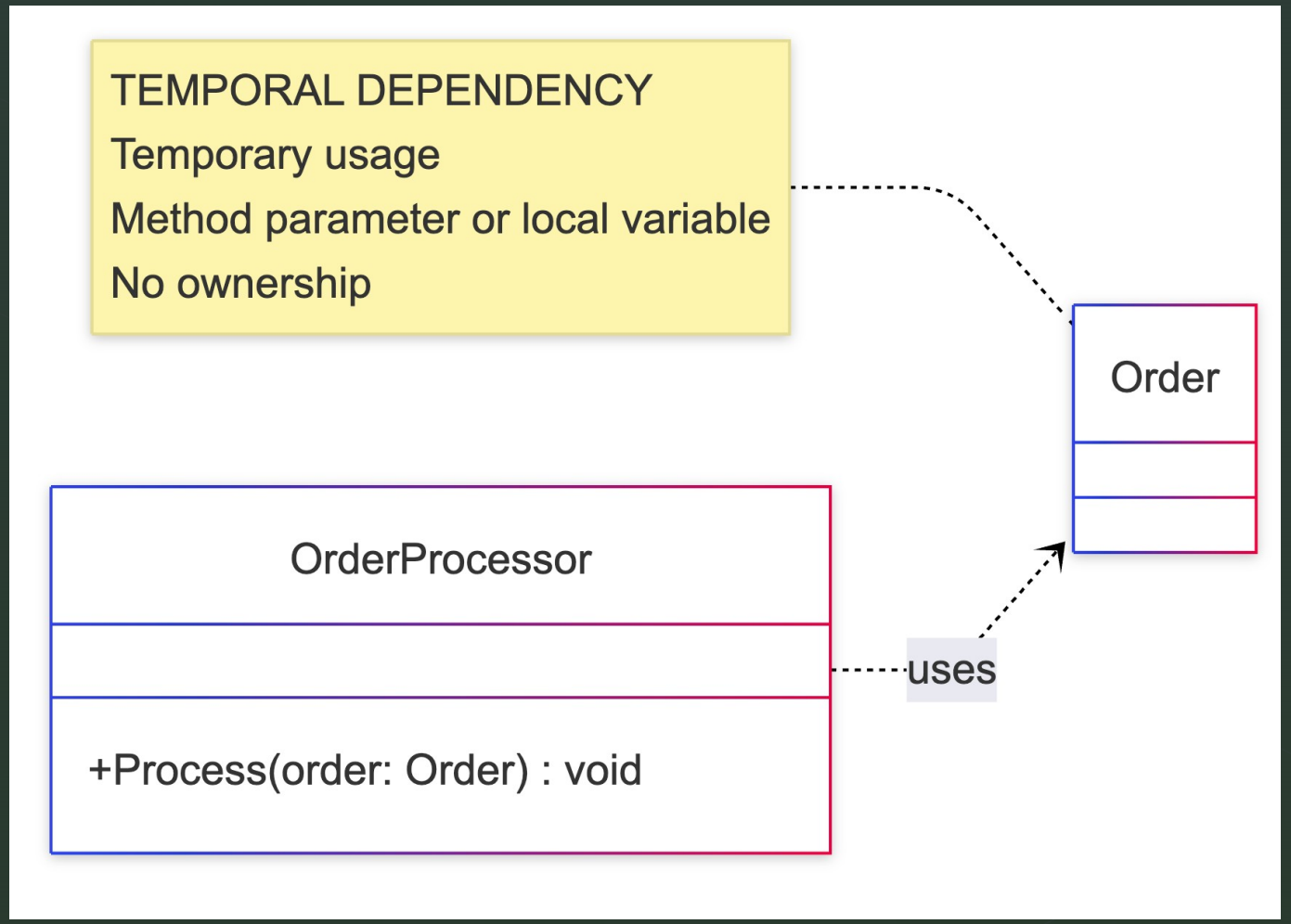
Temporal Usage Dependencies

Not all dependencies should be fields.

UML dependency arrows help distinguish temporary usage from structural ownership.

- Method parameters
  - Local variables
  - Short-lived interactions

Dependency / Temporal State Structural



Non-Temporal State Dependencies

Some dependencies exist because an object **stores state**, not because it needs something temporarily during execution.

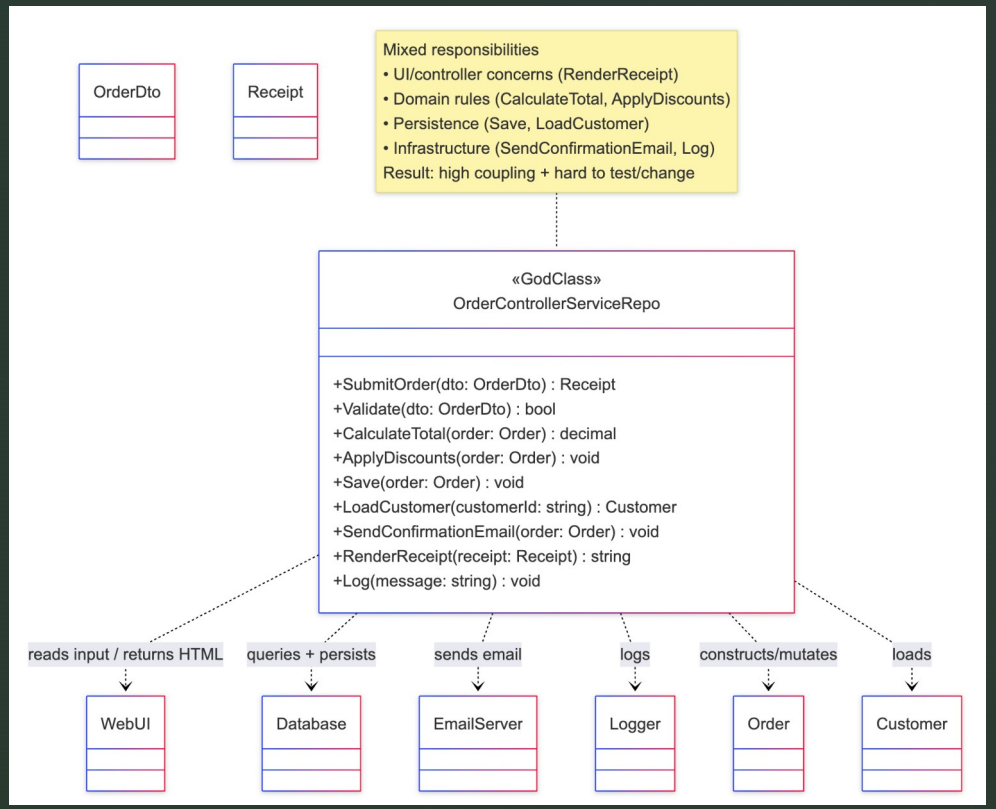
- **Represent persistent structure**
  These dependencies exist as **fields** and are part of the object's long-term state.
- **Express object relationships**
  Association, aggregation, and composition describe *how objects are connected*, not how methods execute.
- **Survive across method calls**
  These relationships exist beyond a single operation or call stack.
- **Communicate design intent**
  UML structural arrows show ownership, replaceability, and lifecycle expectations.
- **Form the object graph**
  These dependencies define the system's shape at runtime.

What Dependencies Reveal

Dense or tangled structures are easier to spot in diagrams than in code.

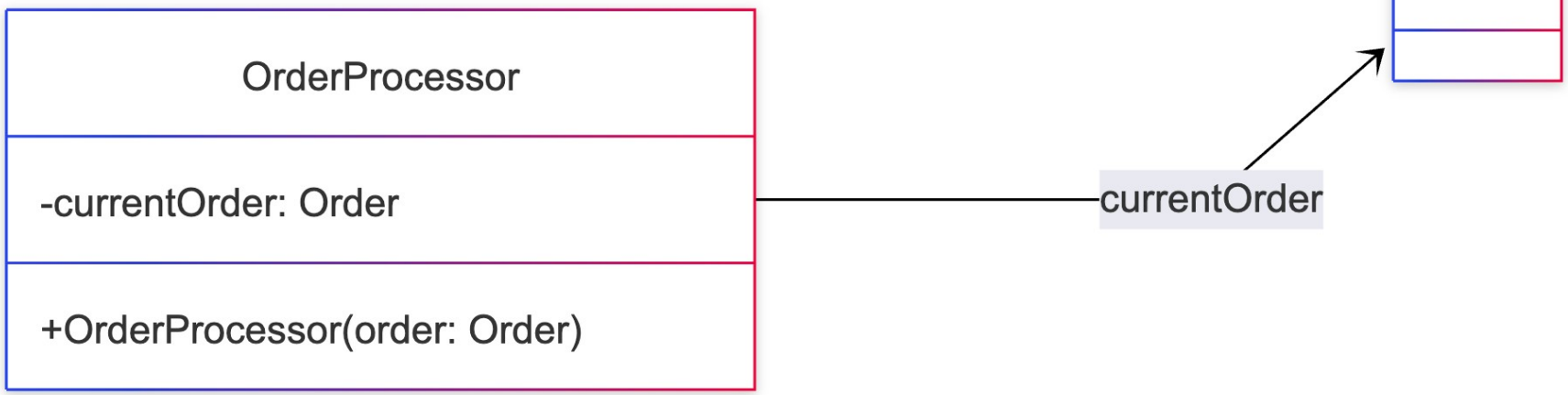
This is often where students first notice over-coupling or god classes.

- Object lifetime relationships
  - Tight vs loose coupling
  - Potential design smells

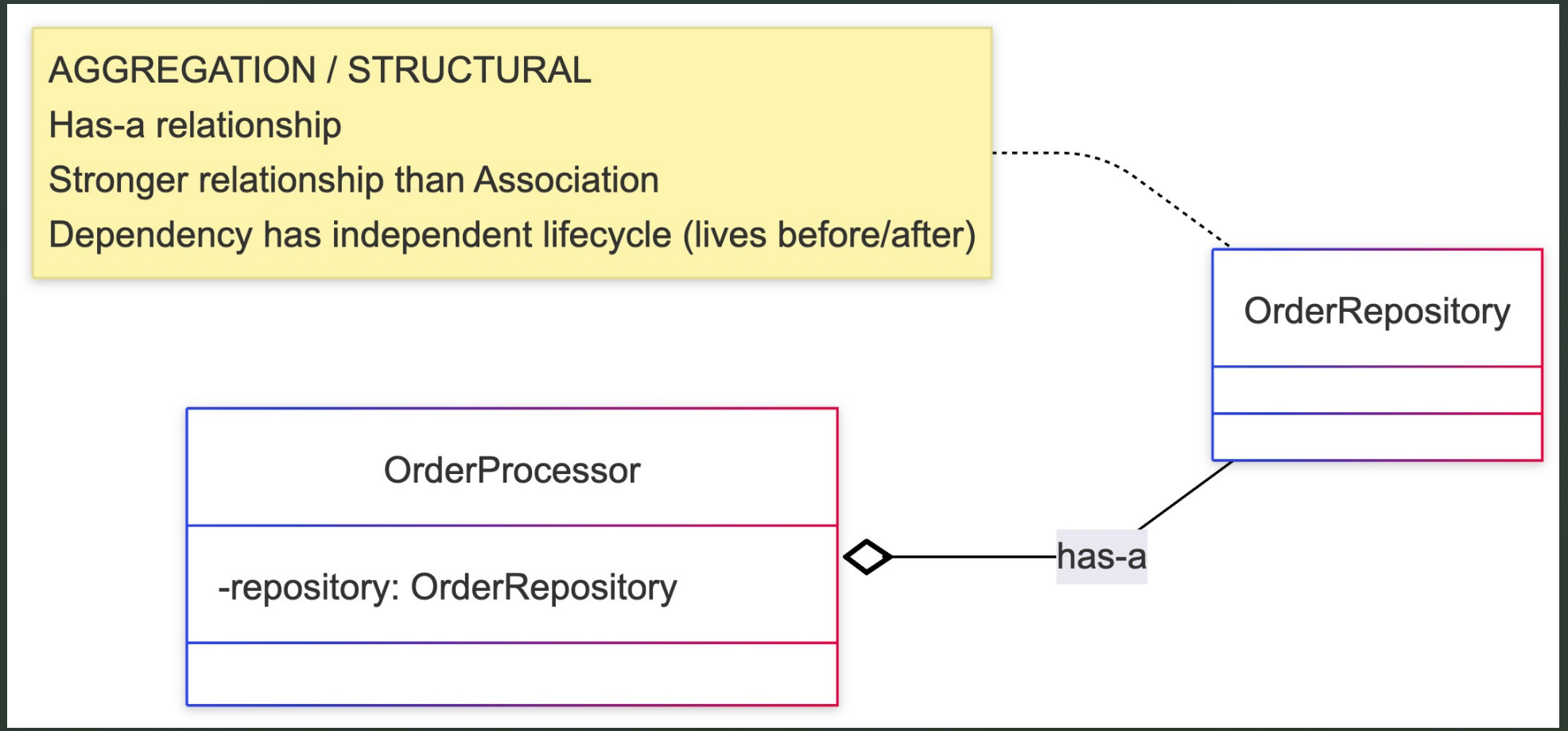


Association / State-Level Structural

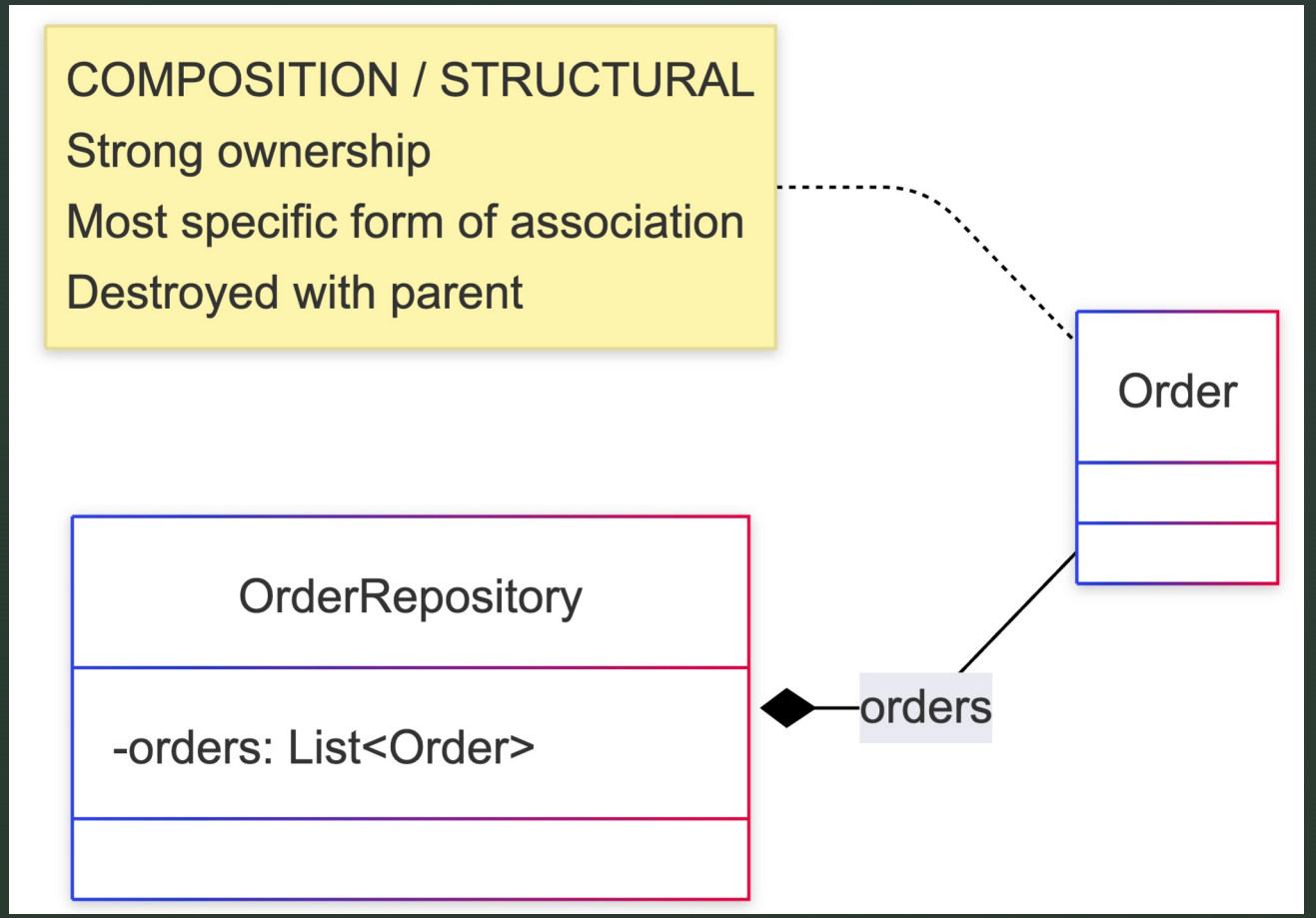
ASSOCATION / STRUCTURAL
Weakest structural relationship
Field populated numerous ways - constructor, method, setter, etc.
Preferred over Comp and Agg unless strict lifetime ownership is required



Aggregation / State-Level Structural



Composition / State-Level Structural



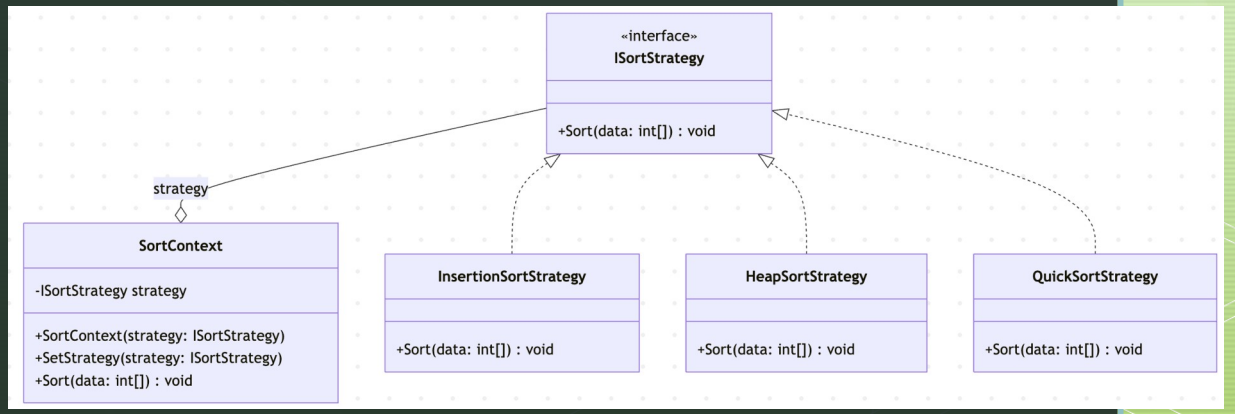
Why State-Level Dependencies Matter

- While performing your design, consider this: “Does this class really need to own this dependency, or does it only need it temporarily?”
- Reduce unnecessary coupling
  - Clarify required collaborators
  - Support cleaner APIs

Seeing Abstractions Visually

Patterns like Strategy and Observer become much clearer when interfaces are visible and not hidden behind concrete implementations.

- Interfaces as first-class elements
  - Multiple realizations
  - Pluggable behavior



Focused Design Discussions

Good diagrams are selective! Diagram what matters for the decision you are considering, not everything that exists.

- Limit diagrams to key classes
  - Avoid giant system diagrams
  - Design is contextual

Why Diagrams Beat Code to Promote Discussion

Reading code is serial and slow.

Diagrams allow parallel understanding and quicker agreement or disagreement during reviews.

- Easier to reason about structure
  - Less cognitive load
  - Supports group conversation

Pull Requests & Design Defense

Consider using diagrams in pull requests (PR: code you are submitting to your teammates, technical lead, open source project, etc.) for inclusion in a project or design reviews to explain *why* something is built a certain way instead of arguing line-by-line in code.

- Explain intent visually
  - Justify abstractions
  - Highlight dependency choices

Responsibility Boundaries

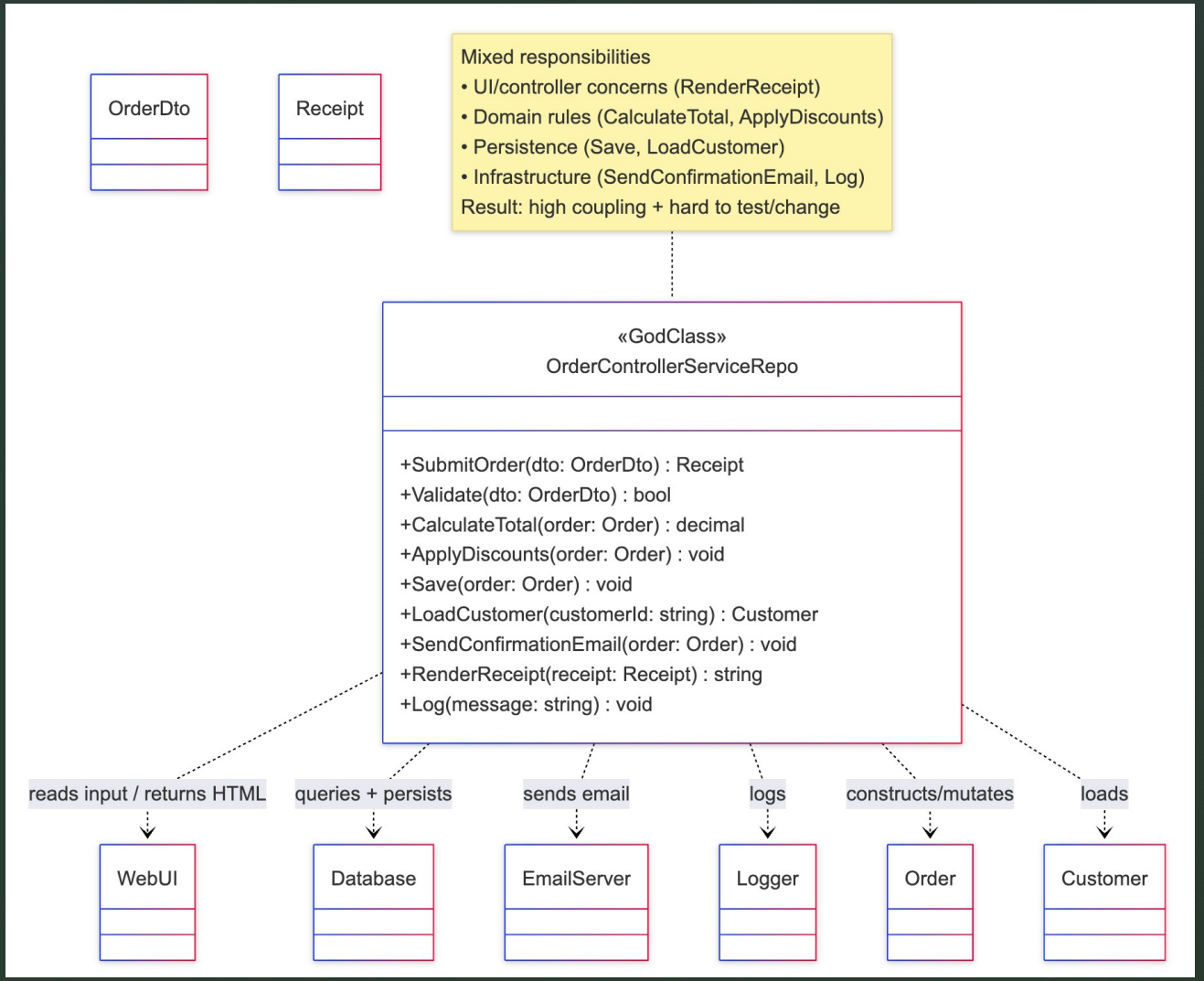
Each class box invites questions.

Too many relationships often indicate too many responsibilities.

Try to keep each class limited to a Single Responsibility. Diagrams help spot responsibility problems that might be less visible when reading code.

- Who does what?
  - Who knows about whom?
  - When to split classes

Responsibility Boundaries



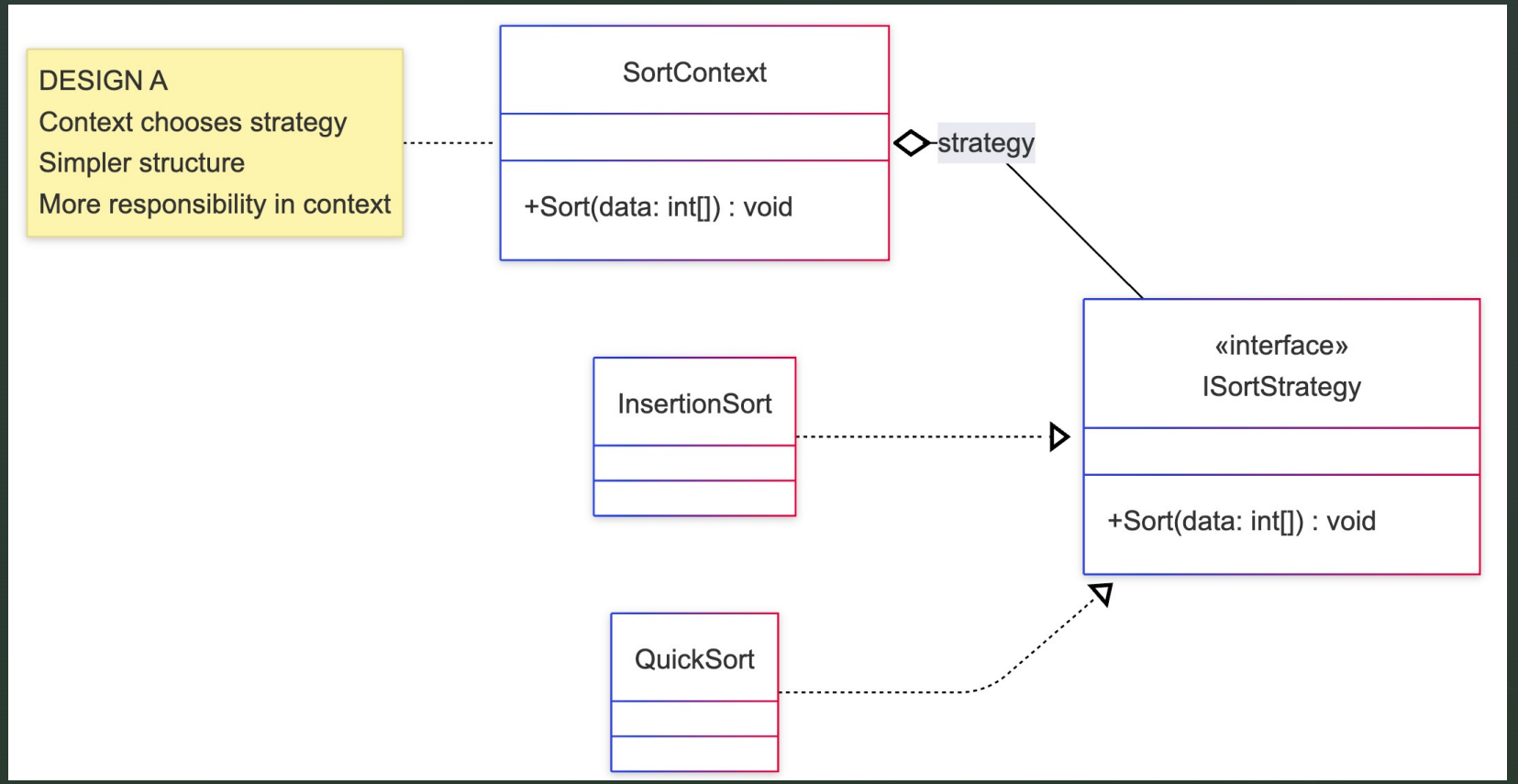
Exploring Alternatives

Whiteboarding diagrams and diagram tool like Mermaid.js allow experimentation without refactoring code.

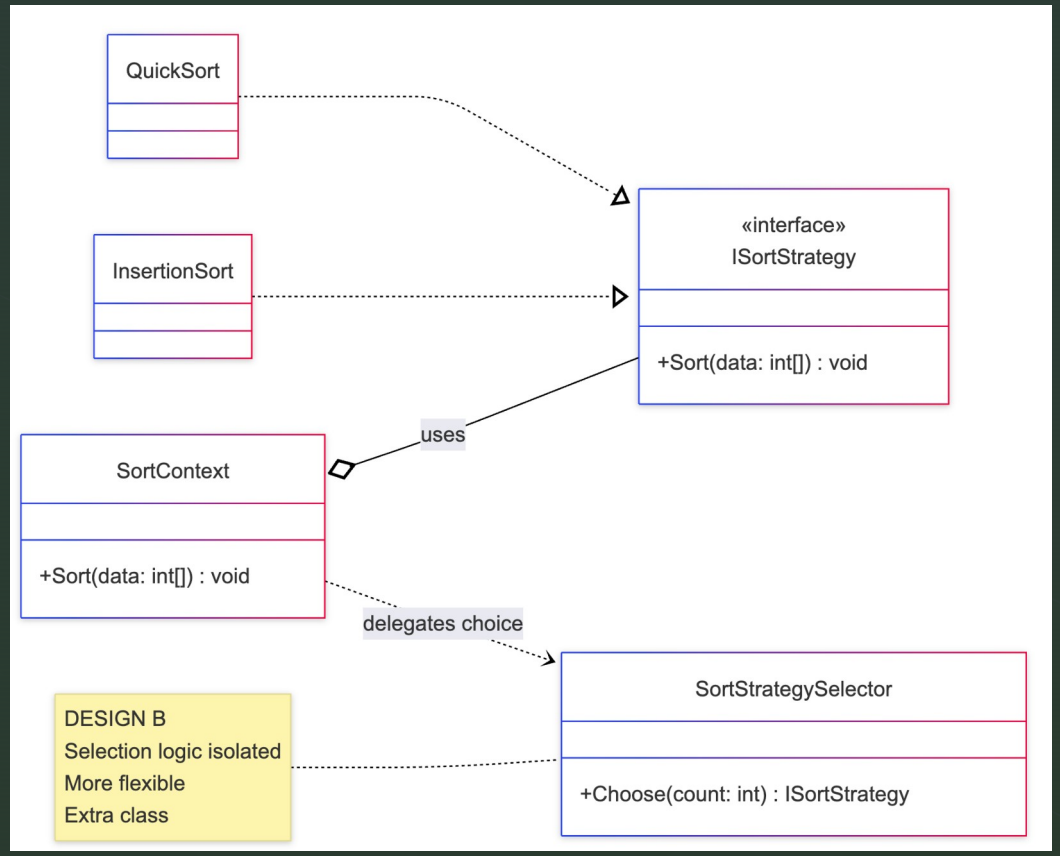
This encourages exploration instead of premature commitment.

- Try multiple designs quickly
  - Compare structures
  - Evaluate tradeoffs

Exploring Alternatives: Example A



Exploring Alternatives: Example B



Diagrams as Living Documentation

Even if diagrams drift from code, they often preserve the original intent, which is incredibly valuable context later.

- Capture design intent
  - Aid future maintainers
  - Support architectural memory

Why Code-Based Diagrams (Mermaid) Work Especially Well with AI

Even when diagrams drift from the code, they preserve **design intent**, which provides critical context for both humans and AI.

- **Improved AI reasoning**
  AI can reason over the structured text in code-based diagrams (such as Mermaid), enabling better code generation, refactoring suggestions, pattern recognition, and automated design critiques.
- **Shared context for humans and AI**
  Mermaid diagrams are versionable, diff-able, and colocated with code—making them a common language for developers *and* tools.

Key Takeaways

Used lightly and intentionally, UML class diagramming is a powerful tool for designers and developers.

- Class diagrams support design thinking and critical decision making
  - They reveal dependencies and abstractions hidden by large code submissions
  - They focus and improve design conversations