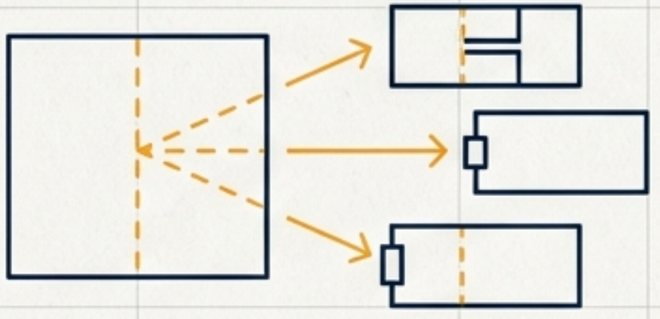
The Interface Segregation Principle

CS 4743 Object-Oriented Design

Principles of Scalable Software Architecture

Lecture Agenda

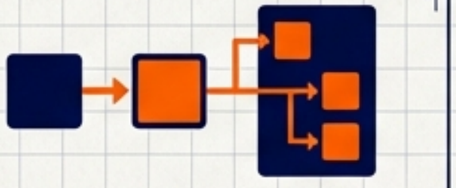
- Canonical Definition & Technical Interpretation
- Refactoring 'Fat' Interfaces (Case Study)
- ISP vs. Other SOLID Principles
- Design Heuristics & Smell Detection
- Integration with Adapter & Facade Patterns



Defining the Principle

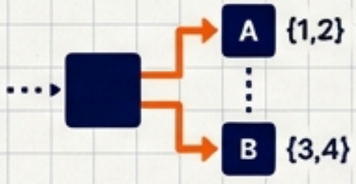
> **“Clients should not be forced to depend upon interfaces that they do not use.”**
>
> **— Robert C. Martin**

Precise Technical Interpretation



Dependency Scope

A client should depend only on the specific operations it requires.

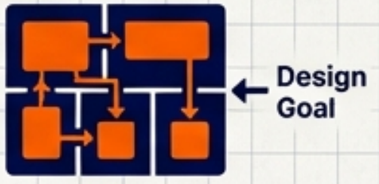


Disjoint Usage

If Client A uses methods (1,2) and Client B uses (3,4), the interface is too broad.

Coupling Risk

Broad interfaces couple unrelated change axes (e.g., Warehouse changes breaking Checkout).



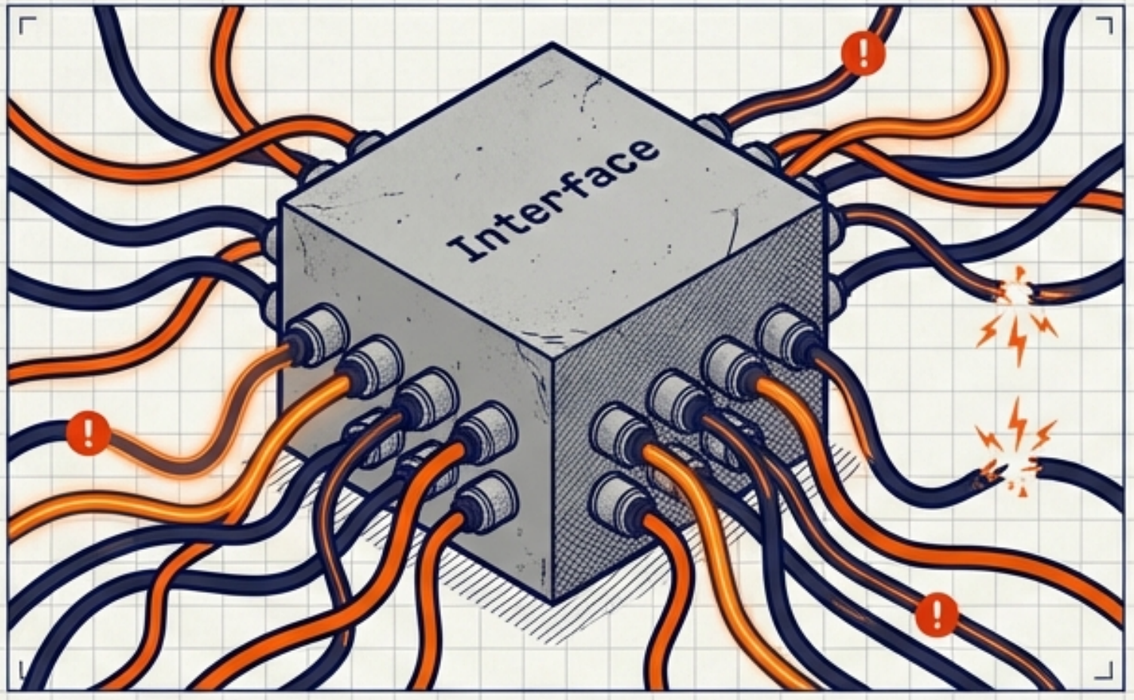
Design Goal

Interface design must follow client usage boundaries, not implementation convenience.

ENGINEERING LEGEND	PROJECT: INTERFACE SEGREGATION
	SCALE: 1:1

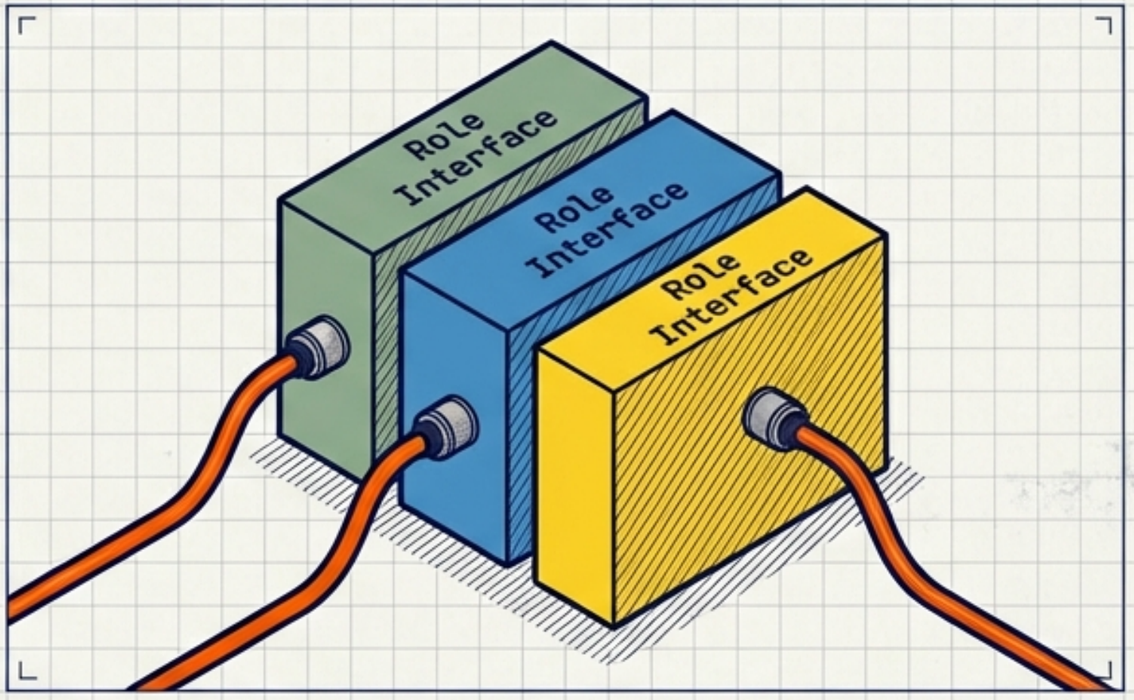
The 'Fat Interface' Problem

The Fat Interface



- When an interface tries to represent too many jobs, every client pays the price.
- Exposes a massive surface area, most of which is irrelevant to any single caller.

The Role Interface

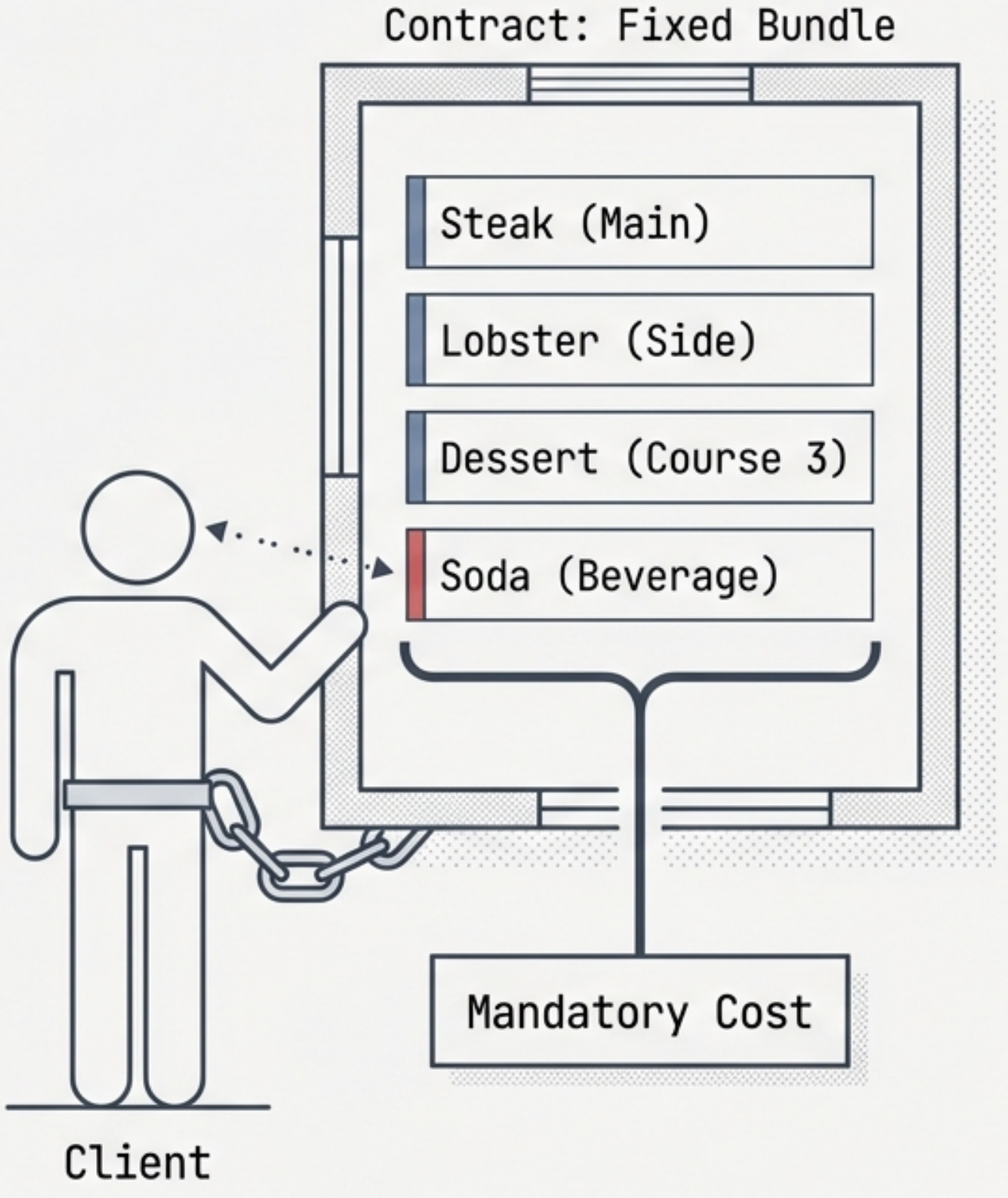


- Expose focused contracts (Role Interfaces).
- Each client sees a small, relevant surface area.
- Changes to 'Admin' do not force recompilation in 'User View'.

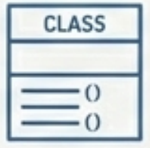
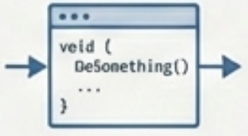
ENGINEERING LEGEND	PROJECT: INTERFACE SEGREGATION
	SCALE: 1:1

The Mental Model: The Fixed-Menu Constraint

- **Scenario:** A customer enters a restaurant where the menu is a single, indivisible bundle.
- **The Constraint:** To order a soda, the customer is contractually required to order, pay for, and receive a steak, lobster, and dessert.
- **The Friction:** The customer cannot opt-out; they must manage the waste and cost of the unwanted items.
- **Architectural Waste:** Resources are consumed preparing items the customer will discard.
- **Rigidity:** If the kitchen changes the lobster preparation, the soda-drinker is impacted.



Mapping the Analogy to Architecture

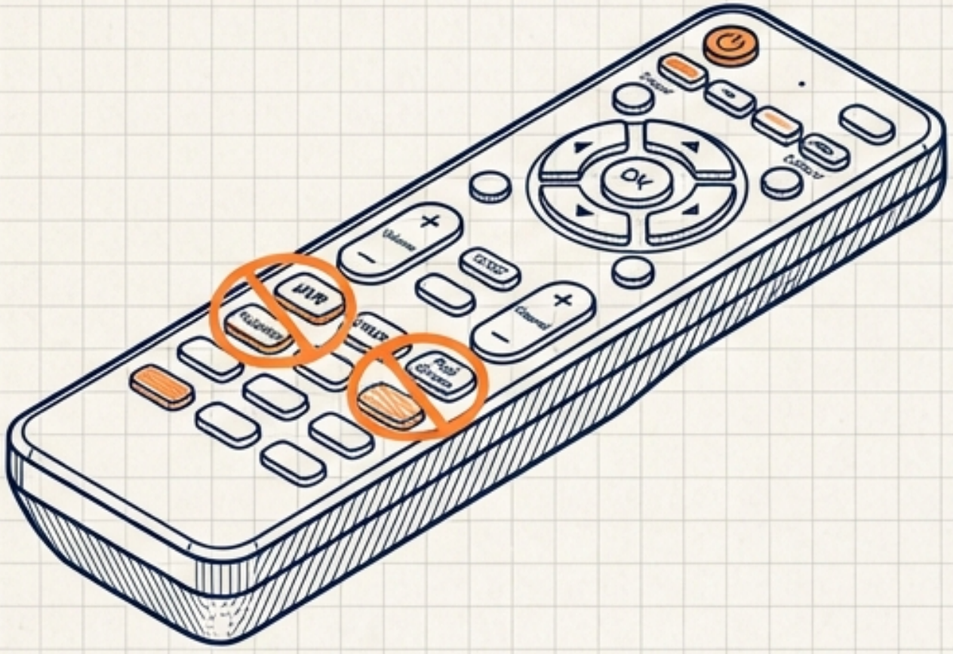
The Analogy	The Architecture
 The Menu	 **The Interface** (Definition of operations)
 The Customer	 **The Client Class** (The consumer)
 Unwanted Food	 **Unused Methods** (Forced dependency)

Technical Consequences:

- **Client-Forced Dependency:** Clients depend on operations they do not use, creating artificial coupling.
- **Regression Risk:** Changes to the complex method (Lobster) force recompilation of the simple client (Soda Drinker).
- **Implementation Noise:** Client code polluted with 'NotImplementedException'.

Real-World Analogies

The Hotel Remote (Everyday Friction)



- **Context:** A remote with buttons for features (DVR, Projector) not present in the room.
- **The Flaw:** Guests see all buttons; pressing them triggers errors.
- **The Risk:** Changing the DVR mechanism requires replacing remotes in rooms that don't even use DVR.

Air Inter Flight 148 (Safety-Critical)

- **Context:** Cockpit control mixed semantically different modes.
- **The Flaw:** Ambiguous, overloaded interface increased cognitive load.
- **The Lesson:** In safety-critical code, poor segregation increases error likelihood.

ENGINEERING LEGEND	
PROJECT: INTERFACE SEGREGATION	
SCALE: 1:1	DATE: 2024

Bad Example: The Fat Interface

Scenario: An Enterprise Order Platform handling Checkout, Warehouse, and Compliance.

```
// FAT INTERFACE: Forces clients to see methods they don't own
public interface IOrderPlatformService {
    Order PlaceOrder(Cart cart);           // Checkout needs this
    void PrintPickList(DateOnly date);     // Warehouse needs this
    void ExportTaxAuditCsv(int year);      // Compliance needs this
}
public sealed class WebCheckoutService : IOrderPlatformService {
    public Order PlaceOrder(Cart cart) { return new Order(); }

    // VIOLATION: Checkout forces compile-time dependency but runtime failure
    public void PrintPickList(DateOnly date)
        => throw new NotImplementedException("Checkout doesn't do picking.");

    public void ExportTaxAuditCsv(int year)
        => throw new NotImplementedException("Checkout doesn't do tax.");
}
```

Compiles fine, but crashes or confuses developers.

ENGINEERING LEGEND	
PROJECT: INTERFACE SEGREGATION	
SCALE: 1:1	DATE: 2024

Why It Violates ISP

1 Forced Implementation

The class must implement `PrintPickList` just to satisfy the compiler.

2 LSP Violation ⚠

The subtype throws exceptions for valid interface methods (`NotImplementedException`), breaking substitutability.

3 Fragile Dependency

A change to the `ExportTaxAuditCsv` signature forces `WebCheckoutService` to recompile/deploy, even though it doesn't use that method.

4 Cognitive Load

Developers working on Checkout must mentally filter out Warehouse methods from IntelliSense.

ENGINEERING LEGEND	
PROJECT: INTERFACE SEGREGATION	
SCALE: 1:1	DATE: 2024

Refactored: Segregated Interfaces

The Fix: Split the ‘Fat’ interface into Role Interfaces based on client needs.

```
public interface
IOrderPlacementService {
  Order PlaceOrder(Cart cart);
}
```

```
public interface
IWarehouseOperationsService {
  void PrintPickList(DateOnly
date);
}
```

```
public interface
IAuditExportService {
  void ExportTaxAuditCsv(int
year);
}
```

```
// IMPLEMENTATION: Can implement multiple interfaces, but clients only see one
public sealed class WarehouseBackOfficeService
    : IWarehouseOperationsService, IAuditExportService {
    public void PrintPickList(DateOnly date) { /* Logic */ }
    public void ExportTaxAuditCsv(int year) { /* Logic */ }
}
```

ENGINEERING LEGEND	
PROJECT: INTERFACE SEGREGATION	
SCALE: 1:1	DATE: 2024

What Improved

No Placeholder Exceptions

Classes only implement behavior they actually support. No more `throw new NotImplementedException()`.

Decoupled Clients

The `CheckoutController` depends *only* on `IOrderPlacementService`. It doesn't know Warehouse logic exists.

Localized Changes

Changing the Tax Export signature only affects Compliance clients. Checkout is untouched and requires no recompile.

Cleaner Tests

Mocks for Checkout don't need to stub out irrelevant Warehouse methods. Test setup is 3 lines, not 30.

ENGINEERING LEGEND	
PROJECT: INTERFACE SEGREGATION	
SCALE: 1:1	DATE: 2024

ISP and Other SOLID Principles

ISP + SRP (Single Responsibility Principle)

SRP is about **internal cohesion** (one reason to change *inside*). **ISP** is about **boundary cohesion** (one client group requiring dependencies).

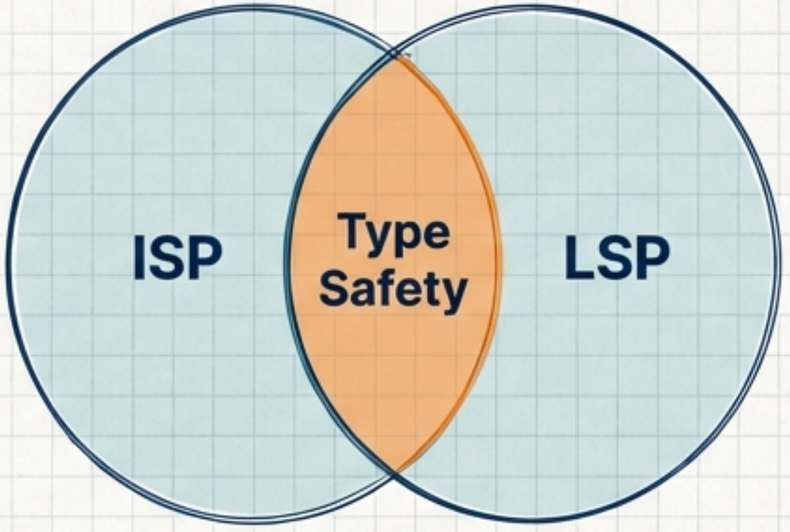
Rule of Thumb: A fat interface usually indicates an **SRP violation** inside the implementing class.

ISP + OCP (Open/Closed Principle)

Fat interfaces are unstable. ISP supports OCP by allowing you to add *new* interfaces for *new* features without touching existing ones.

ISP + LSP (Liskov Substitution Principle)

`NotImplementedException` is a direct violation of LSP. ISP prevents this by ensuring the interface matches the implementation capabilities.



ENGINEERING LEGEND	
PROJECT: INTERFACE SEGREGATION	
SCALE: 1:1	**DATE:** 2024

ISP and Ousterhout

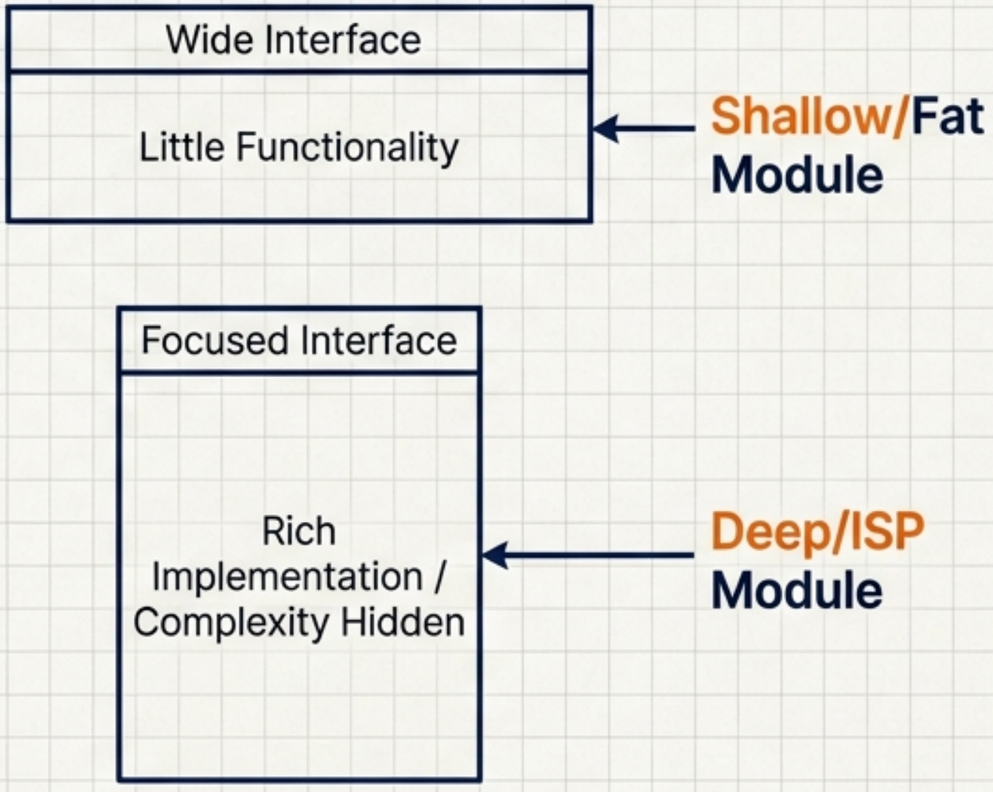
Reference: “A Philosophy of Software Design” by John Ousterhout

Deep Modules

- **Goal:** Simple interfaces, rich/complex hidden implementations.
- **ISP Connection:** ISP promotes “Deep Modules” by preventing shallow, “do-everything” interfaces.

Cognitive Load

- A 20-method interface creates high friction.
- Focused interfaces (2-4 methods) reduce the “surface area” a developer must understand.



ENGINEERING LEGEND	
PROJECT: INTERFACE SEGREGATION	
SCALE: 1:1	DATE: 2024

Practical Application & Heuristics

- **Group by Client:** Align methods with the actor calling them (e.g., `IAdminActions` vs `IUserActions`).
- **Role-Based Naming:** Prefer `IOrderReader`, `IOrderWriter` over generic `IOrderManager`.
- **Change Frequency:** If method A changes weekly and method B changes yearly, they beeong on different interfaces.
- **Testability:** If your mock setup requires stubbing 10 unrelated methods, the interface is too broad.

ENGINEERING LEGEND	
Details:	Inter
Snippet:	JetBrains Mono
Type:	iF4F4F1

To Split or Not to Split?

When to Split (YES)

- Different clients use disjoint subsets of methods.
- You frequently type `` `throw new NotImplementedException()` ``
- You see ‘God Classes’ implementing a single massive interface.

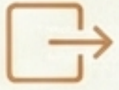
When Not to Split (NO)

- **Atomization:** Don't create an interface for every single method.
- **Cohesion:** If clients **always** need methods A and B together, keep them together.
- **Over-abstraction:** Do not split if it drastically complicates dependency injection without clear benefit.

ENGINEERING LEGEND	
PROJECT: INTERFACE SEGREGATION	
SCALE: 1:1	DATE: 2024

Detection Signals & Smells

Code Review Red Flags

	**Partial Implementation:** Classes that leave methods empty or throwing exceptions.
	**Unused Imports:** Clients importing libraries they don't use, forced by a fat interface.
	`if (service is WarehouseService) ...` Indicates the abstraction failed.
	`process(includeTax: false, rebuildIndex: false)` Method doing too much.
	**Ripple Effects:** A change in 'Search' breaks the build for 'Checkout'.

FONTS:		ENGINEERING LEGEND	
Headlines:	Helvetica Now Display	PROJECT: INTERFACE SEGREGATION	
Body:	Inter		
Code:	JetBrains Mono	SCALE: 1:1	DATE: 2024

Diagnostic Tool: The Client-Method Matrix

Method	Authorize()	Refund()	ExportTax()	RebuildIndex()
Checkout UI	X	-	-	-
Support App	-	X	-	-
Compliance Job	-	-	X	-
Search Ops	-	-	-	X

Interpretation:

Sparse Matrix:
Lots of empty cells indicates disjoint usage. Strong evidence for Interface Segregation.

Dense Matrix:
Mostly X's would indicate a cohesive interface.

FONTS:		ENGINEERING LEGEND	
Headlines:	Helvetica Now Display	PROJECT: INTERFACE SEGREGATION	
Body:	Inter		
Code:	JetBrains Mono	SCALE: 1:1	DATE: 2024

Naming as an Architectural Enforcer

The Semantic Bucket

Risky

```
IOrderManager
IProcessor
IHandler
```

Vague names invite unrelated methods.
“Manager” implies no specific boundary.

The Semantic Contract

Safe

```
IOrderPlacement
ICsvReader
ITaxCalculator
```

Precise names enforce boundaries.
`ICsvReader` implies a strict reading contract.

The ‘And’ Test: If describing the interface requires the word ‘and’ (e.g., “It validates users *and* exports logs”), it likely violates SRP and ISP.

Structural Naming Guidelines

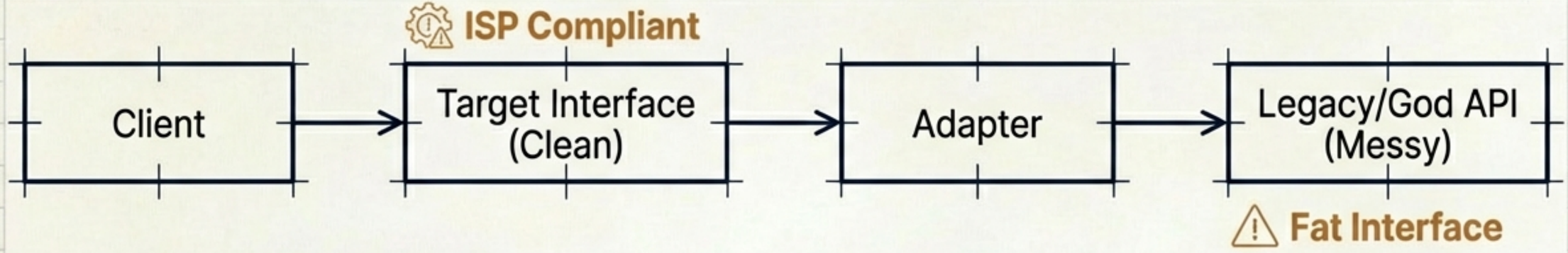
Category	Guideline	Example
Interface Names	Name based on Client Role or Capability.	`IOrderPlacement` (Role) `IAuditable` (Capability)

Method Verbs	Avoid generic `Process()`. Use side-effect specific verbs.	`Calculate()`
		`Persist()`
		`Publish()`

Parameter Precision	Accept the smallest possible interface.	`Print(IReadOnlyList<T>)` instead of `Print(List<T>)`.

Pattern Integration: Adapter

How Adapter Supports ISP (Enabler Pattern)



The **Clean Interface**:

```
public interface IShippingLabelGateway {
    Label Create(Request r);
}
```

The Adapter:

```
public class LegacyAdapter : IShippingLabelGateway {
    private LegacyBigApi _api;
    public Label Create(Request r) =>
        _api.ComplexShipmentWorkflow(r);
}
```

FONTS:		ENGINEERING LEGEND	
Headlines:	Helvetica Now Display	PROJECT: INTERFACE SEGREGATION	
Body:	Inter		
Code:	JetBrains Mono	SCALE: 1:1	DATE: 2024

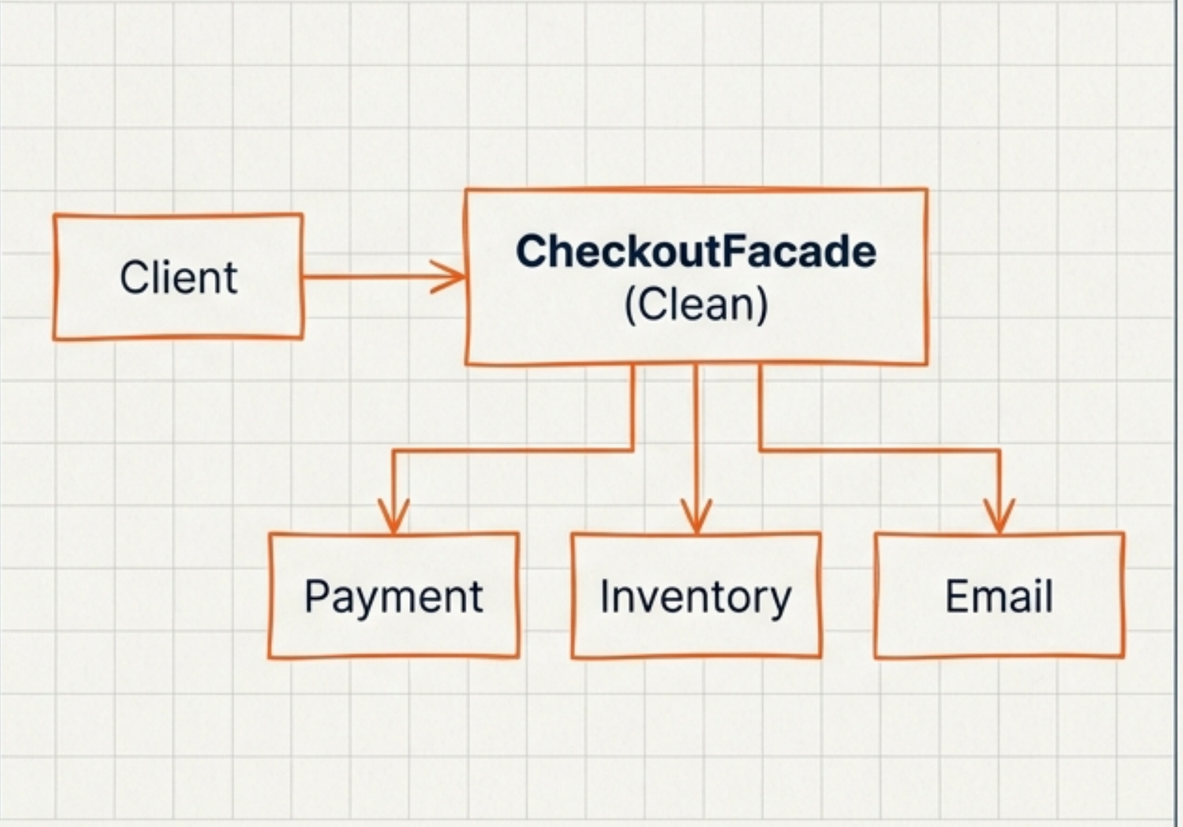
Pattern Integration: Facade

How Facade Supports ISP

Scenario: Client needs to coordinate multiple subsystems (Payment, Inventory, Email).

Solution: Create a Facade exposing *only* the workflow that specific client needs.

Crucial Warning: Do not create a 'God Facade'. Even Facades should be segregated by actor (e.g., `ICheckoutFacade` vs. `IAdminFacade`).



Adapter: Fixes interface mismatch.

Facade: Simplifies complex interactions.

Real-World Summary

1. Client-First Design

Interfaces are defined by who ***uses*** them, not who ***implements*** them.

2. Safety

Segregation prevents runtime errors (`NotImplementedException`) and accidental misuse.

3. Maintenance

Reduced blast radius for changes; unrelated clients remain unaffected.

4. Testability

Smaller interfaces lead to simpler mocks and more focused unit tests.

ENGINEERING LEGEND		
Project: Swiss Engineering	Scale: 1/60	Date: 14/4/2022