SE 350 - Object-Oriented Software Development

Software Design Patterns: Anti-Patterns and Refactoring

Instructor: Stefan Mitsch

Learning Objectives

- Understand "solutions" that create more challenges than benefits
- Understand "code smells": symptoms of poor code quality
- Understand how to improve code quality by refactoring

Introduction

Anti-Patterns

- Describe commonly occurring solutions to problems that generate negative consequences
- Summarize real-world exerpience in recognizing recurring problems
  - Software Development Anti-Patterns
  - Software Architecture Anti-Patterns
  - Software Project Management Anti-Patterns

SW Development: The Blob

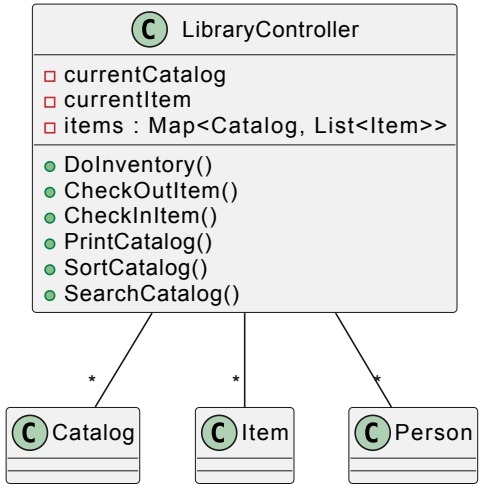
Problem

A small number of classes monopolize processing while the rest provides data

Causes

- Lack of (object-oriented) architecture or enforcement
- Iterative growth without refactoring
- Ported legacy (procedural) solution

Illustration

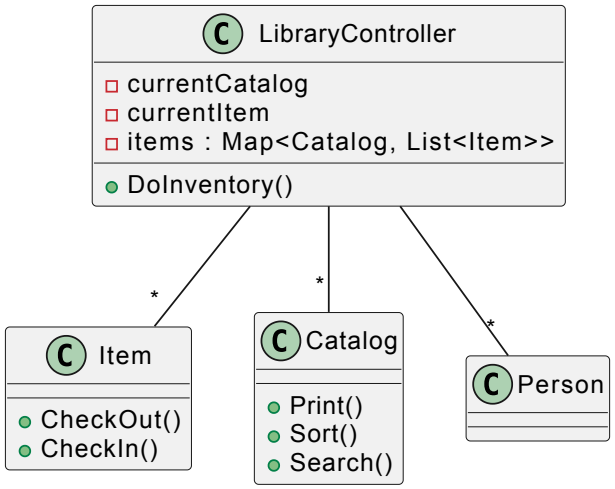


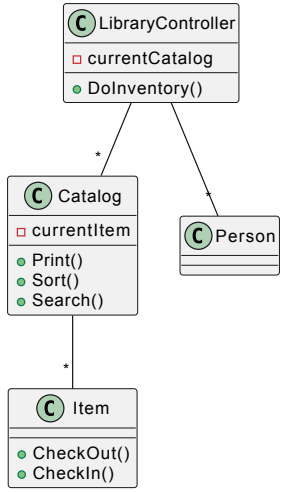
Consequences

- Too complex to test and reuse
- Expensive to load

The Blob Refactoring

- Move methods closer to data
- Identify far-coupling (e.g., `Catalog` and `Item`)
- Restructure associations





SW Development: Functional Decomposition

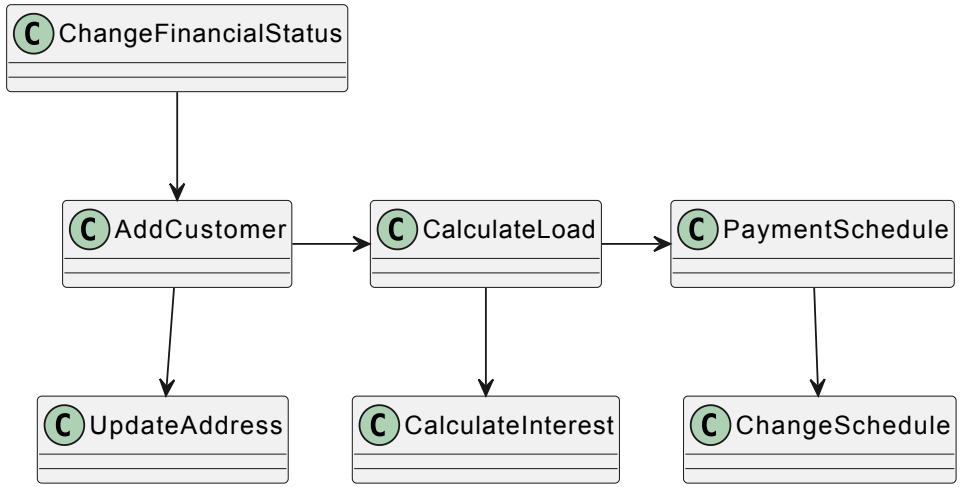
Problem

Functional decomposition does not translate into a class structure

Causes

- Procedural design mapped directly into an OO language
- Overuse of the Command pattern
- Absence of inheritance and polymorphism

Illustration

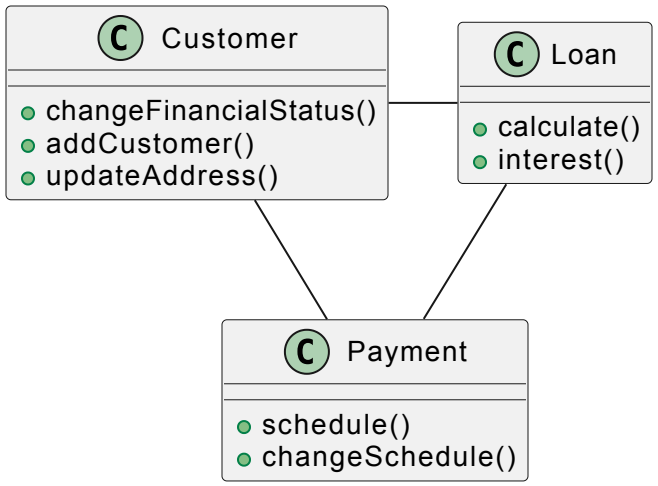


Consequences

- Difficult to reuse
- Difficult class interaction

Functional Decomposition Refactoring

- Create a design model (UML) to explain the existing functionality
- Combine related functionality into classes



SW Architecture: Stovepipe System

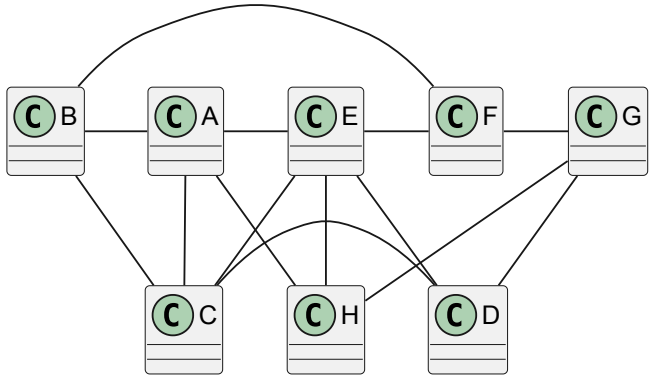
Problem

Point-to-point integration between many subsystems

Causes

- Lack of abstraction: unique interfaces for each subsystem
- Tight coupling between client code and service

Illustration

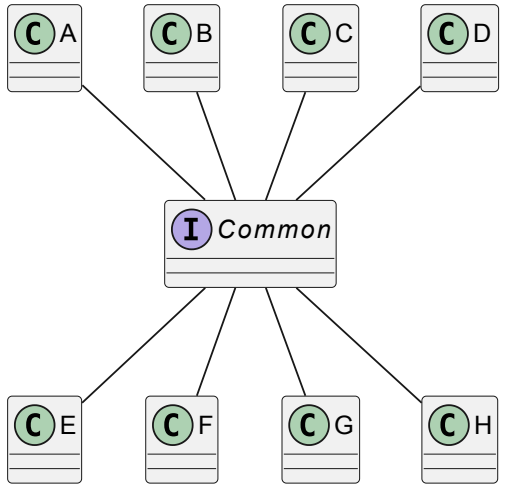


Consequences

- Requirement changes are costly
- Integration of other subsystems is difficult

Stovepipe System Refactoring

- Clarify roles: clients vs. services
- Increase abstraction
  - Introduce common interfaces for services
  - Reduce number of interfaces



SW Architecture: Swiss Army Knify

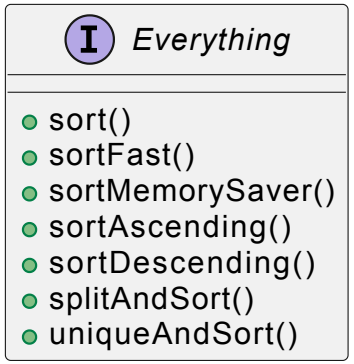
Problem

Complex (class) interface or development standard

Causes

- Attempt to make product apply to all possible situations
- Attempt to address all future foreseeable needs
- Absence of abstraction and encapsulation

Illustration



Consequences

- Hard to implement and use
- Ignores complexity management
- Difficult to test and debug

Swiss Army Knife Refactoring

- Use the Facade pattern to separate interface into smaller profiles to access the Swiss Army Knife
- Then break Swiss Army Knife into separate classes
- Introduce functionality when needed

Anti-Patterns Summary

Software Development, Software Architecture, and Project Management Anti-Patterns

Anti-Patterns

- Describe common defective processes or software development attempts
- Seem solutions-like, but their negative consequences outweigh their benefits

Refactoring

Transform messy code into a simple design and clean code

🛏 Clean Code

- Is obvious to other programmers
- Uses easy-to-understand naming
- Avoids duplication
- Avoids unnecessary/speculative complexity
- Passes all tests

🕓 When to refactor

- **Rule of three**: first time, just get it done; second time, cringe but repeat; third time, start refactoring
- **New feature**: refactor existing code when adding a new feature is difficult
- **Bug fixing**: clean up code when fixing a bug
- **Code review**: use reviews to find difficult code and fix it

Refactoring Techniques

- **Compose methods**: change how methods are composed can improve code
- **Move features** between objects: restructure how functionality is divided between classes can improve clarity
- **Organize data**: decouple classes to make them more portable and reusable
- **Simplify conditionals**: change complicated conditional program flow
- **Simplify method calls**: make method signatures easier to understand
- **Abstraction**: move functionality along class hierarchies

Code Smells

- **Bloaters**: methods or classes that have grown too large
- **Object-Orientation Abusers**: incorrect application of object-oriented principles
- **Change Preventers**: need to modify code in lots of different places to implement a change or new feature
- **Dispensables**: dead code or incorrect comments where removing entirely would make the code cleaner
- **Couplers**: tight coupling between objects

Bloater: Long Parameter List

⚠ Code Smell

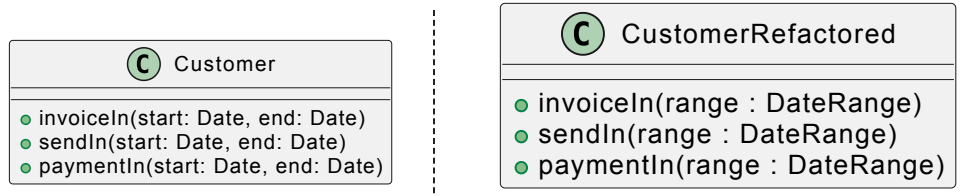
Long list of parameters

Causes

- Attempt to merge multiple algorithms into a single method
- Attempt to separate classes: need to pass lots of data

Treatment

- Replace parameter with method call (get data from somewhere else)
- Introduce parameter object



Object-Orientation Abuser: Temporary Field

⚠ Code Smell

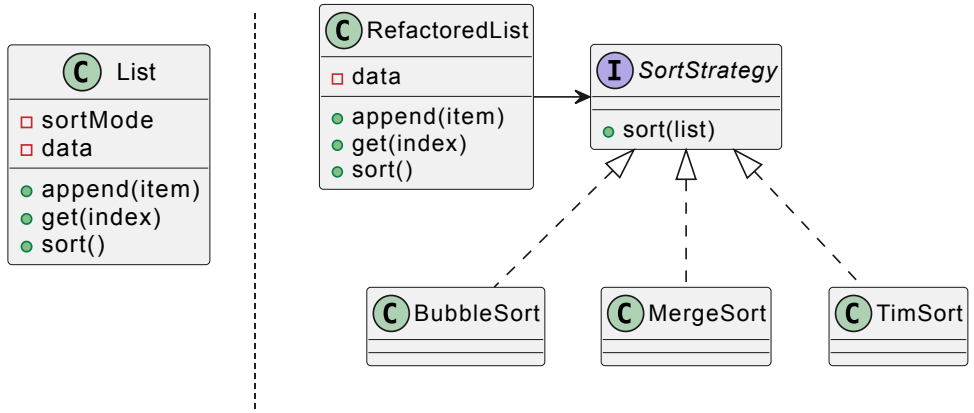
A field gets its value only under certain circumstances

Causes

- Attempt to replace method parameters with fields
- Attempt to implement switchable algorithm strategies in single class

Treatment

- Separate temporary field and methods using it into their own class
- Use `Option` or introduce a Null object (instead of using `null`)



Change Preventer: Parallel Hierarchies

⚠ Code Smell

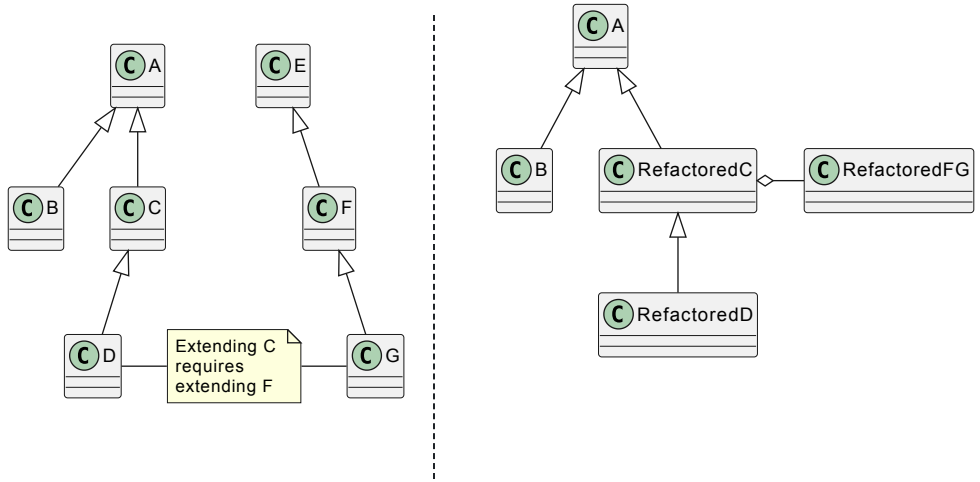
Whenever creating a subclass for one class, we find ourselves needing to create a subclass for another class

Causes

- Attempt to separate concerns into different class hierarchies

Treatment

- Favor composition over inheritance



Dispensable: Speculative Generality

⚠ Code Smell

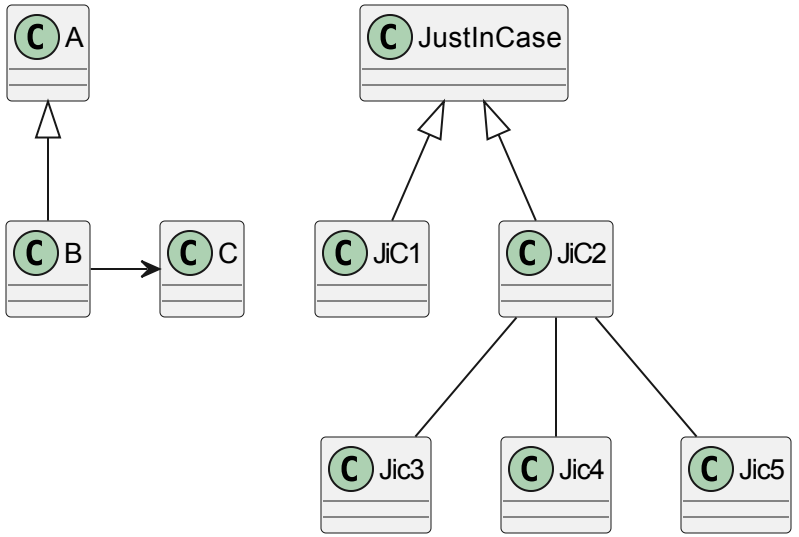
Unused class, method, field, or parameter.

Causes

- Attempt to support anticipated future features
- Attempt to support runtime flexibility that ends up never being used
- Speculative overuse of design patterns

Treatment

- Collapse class hierarchies
- Inline class or method
- Remove fields or parameters



Coupler: Feature Envy

⚠ Code Smell

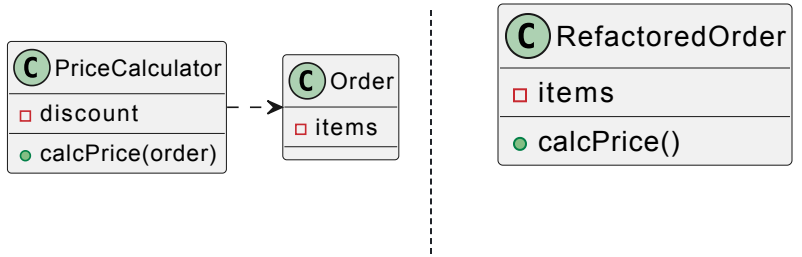
A method accesses data of other objects more than its own class data

Causes

- Fields are moved to a data class
- Parts of a method belong elsewhere

Treatment

- Determine where most of the needed data resides
- Separate method into smaller ones to move only parts
- Move parts or entire method



Incomplete Library

⚠ Code Smell

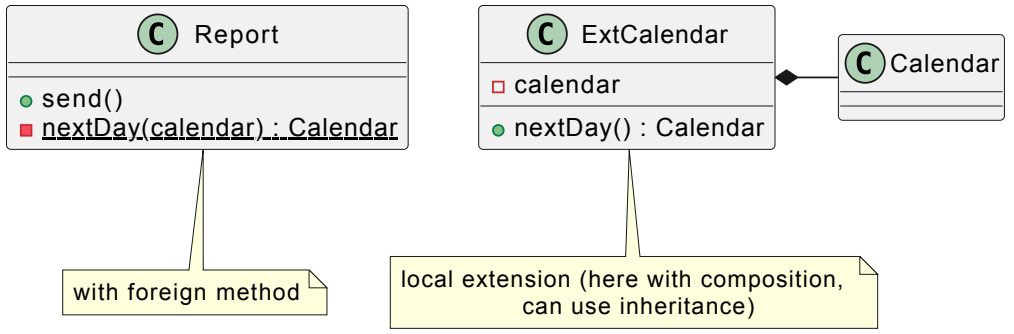
A library does not quite meet our needs, but changing it is impossible

Causes

- Third-party library does not provide enough features
- Legacy code no longer updated

Treatment

- Small changes: introduce foreign method
- Big changes: introduce local extension



Refactoring Summary

Refactoring is a **technique to improve code clarity**

- Improves maintainability, lowers cost of enhancements
- Part of day-to-day programming
- Done when necessity arises
- Supported by many IDEs

Refactoring catalogs

- Refactoring Guru
- Martin Fowler's Refactoring Catalog