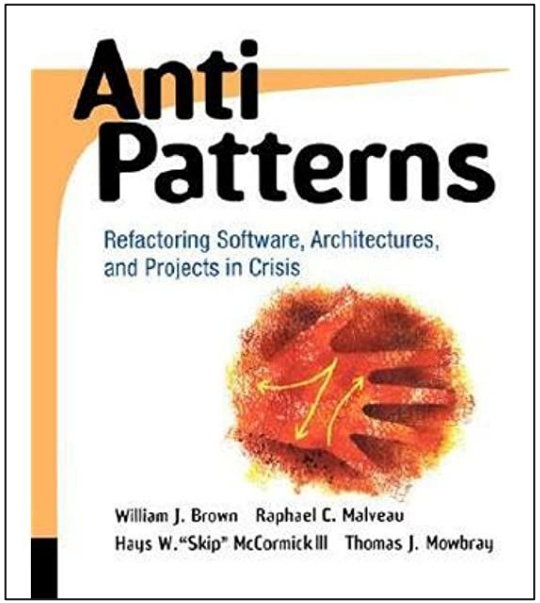
AntiPatterns

History

- Coined in 1995 by Andrew Koenig of Bell Labs
  - Inspired by the Gang-of-four's *Design Patterns*

- Outlined in 1998's *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*
  - Written by The "Upstart Gang of Four"
  - Extended concept into managerial domain



Patterns

Software Pattern: A set of design decisions employed to solve a common problem.

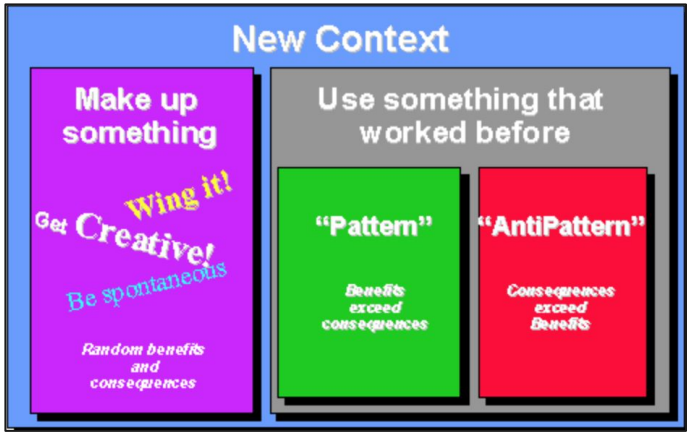
- Good Patterns:
  - Shown to be effective through trial and error.
  - Benefits > Consequences
- Anti Patterns
  - Observed over many failed software products.
  - Consequences > Benefits

Types:

- Development
- Architectural
- Management

Informal Definitions

- Any contextually-inappropriate pattern
- Any pattern where another solution exists to the problem the AP is trying to address, and “this solution is documented, repeatable, and proven to be effective where the anti-pattern is not.” tldr any relatively non-optimal solution
- A solution that will work, but will probably require refactoring.



Why study them?

- “Patterns are an abstraction of experience”
- Learn from other’s (costly) mistakes
- Provide a roadmap of the most common pitfalls in the software industry.
- Steer us towards good practices.

Spaghetti Code

- Convoluted code, possibly with subpar documentation.
- Vacation heuristic
- Symptoms in OOP:
  - Many object methods with no parameters
  - Intertwined/unexpected relationships between objects
  - Process-oriented methods
- Advantages of OOP are lost: inheritance can’t be used to extend, and polymorphism is ineffective.
- Cost of maintenance > Cost of rebuilding from scratch

Spaghetti Code Solutions

Reactive:

- Refactor to generalize: Create an abstract superclass
- Refactor to specialize: Simplify conditionals
- Refactor to combine: capture aggregations and components
    - Move members between aggregate and component classes
    - Convert inheritance to aggregation

Proactive:

- Refactor as you program
- Use good practices (SOLID, patterns, etc)

God Object / Blob

- Symptoms
  - A single class with many attributes and operations.
  - A controller class with simple, data-object classes.
  - Lack of OO design (Violating Single Responsibility Principle)
  - A migrated legacy design
- Problems: difficult to understand, maintain, and test. Promotes tight coupling, losing OO advantages
- Context: Data master in game development

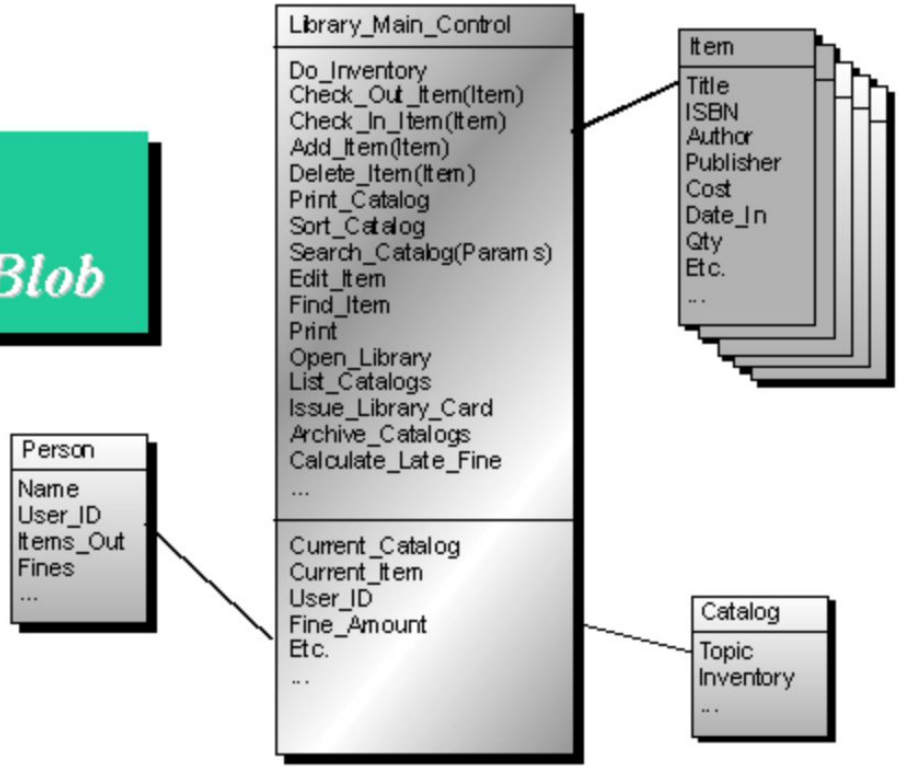
Refactoring

Development AntiPattern:

The Blob

Example:

The Library Blob



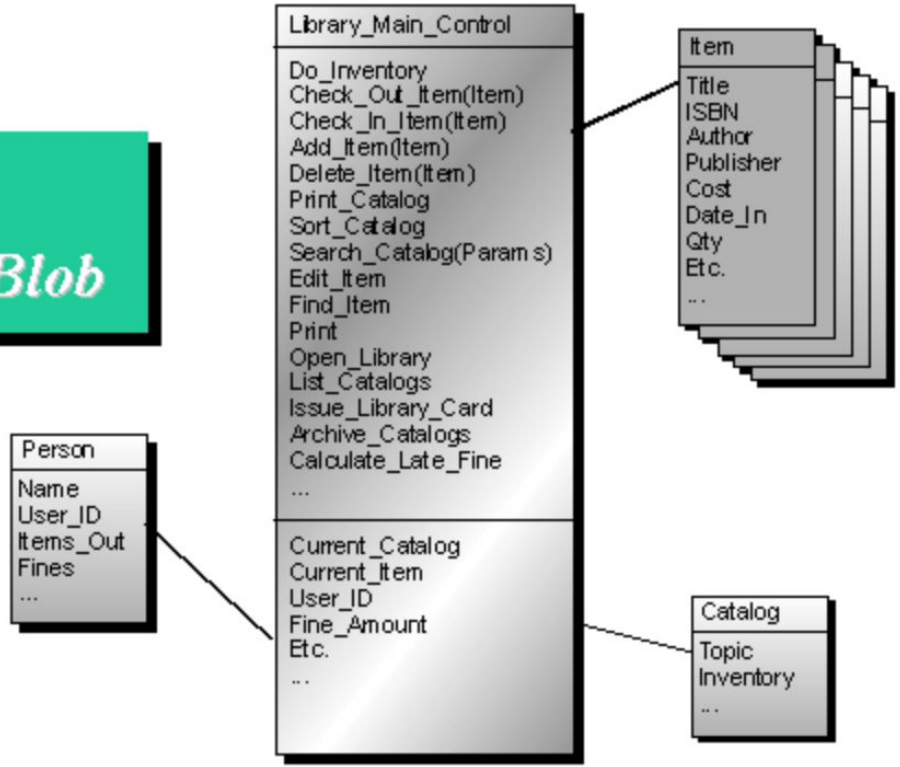
Refactoring

Development AntiPattern:

The Blob

Example:

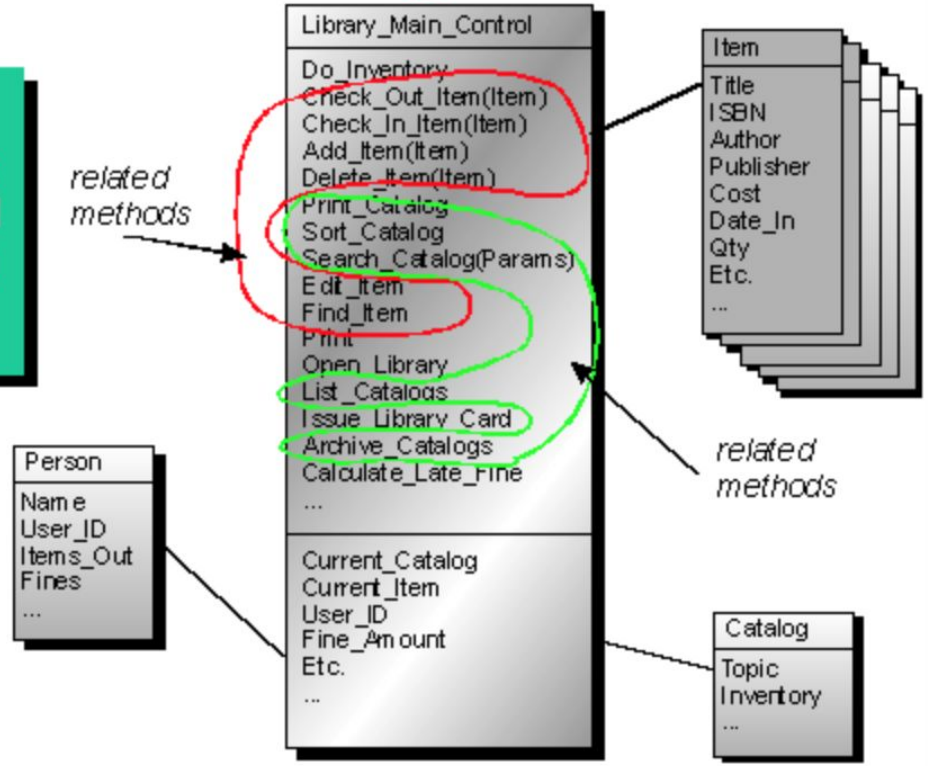
The Library Blob



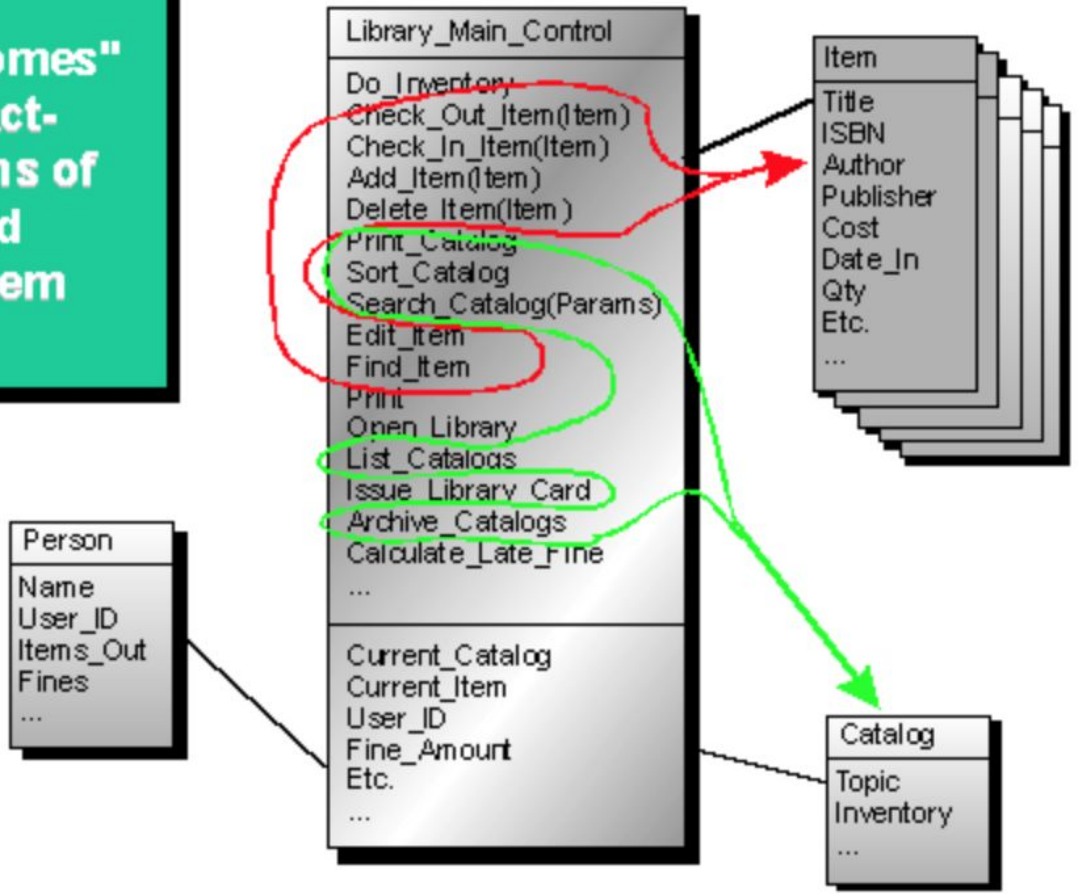
Development AntiPattern:

The Blob - Refactoring

Step 1:
Identify or categorize related attributes and operations according to contracts.



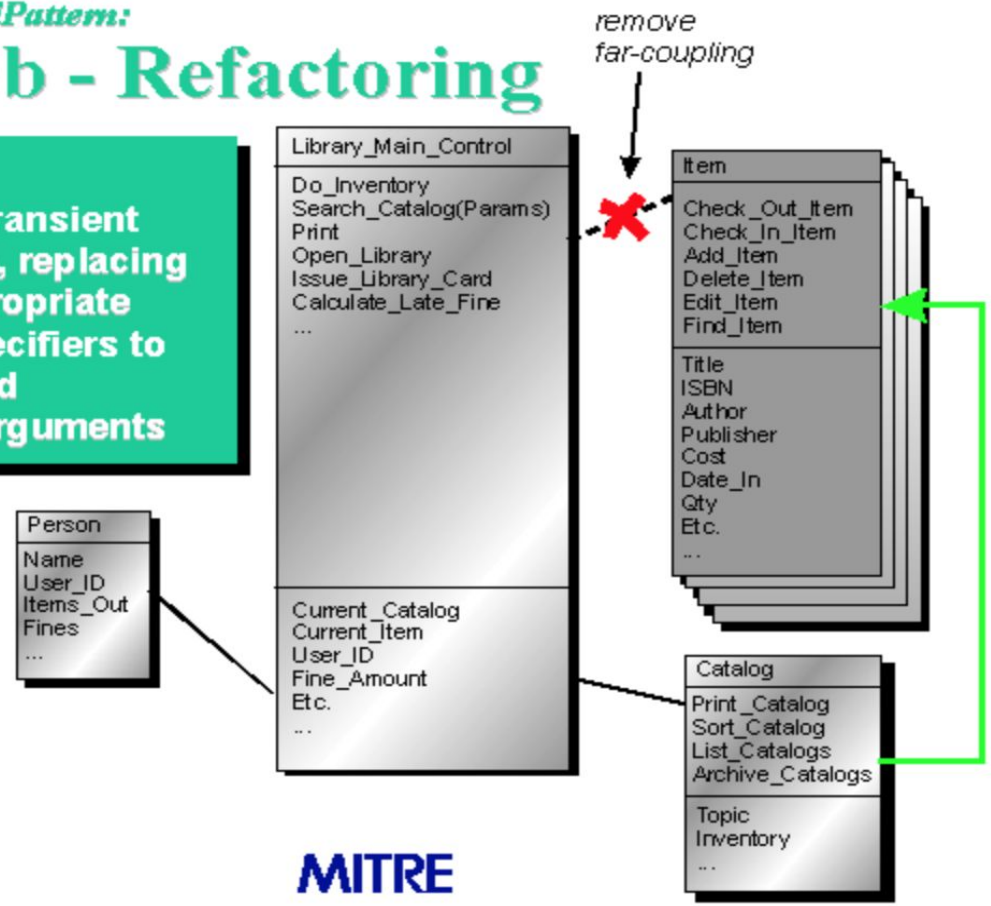
Step 2:
Find "natural homes" for these contract-based collections of functionality and them migrate them there



Development AntiPattern:

The Blob - Refactoring

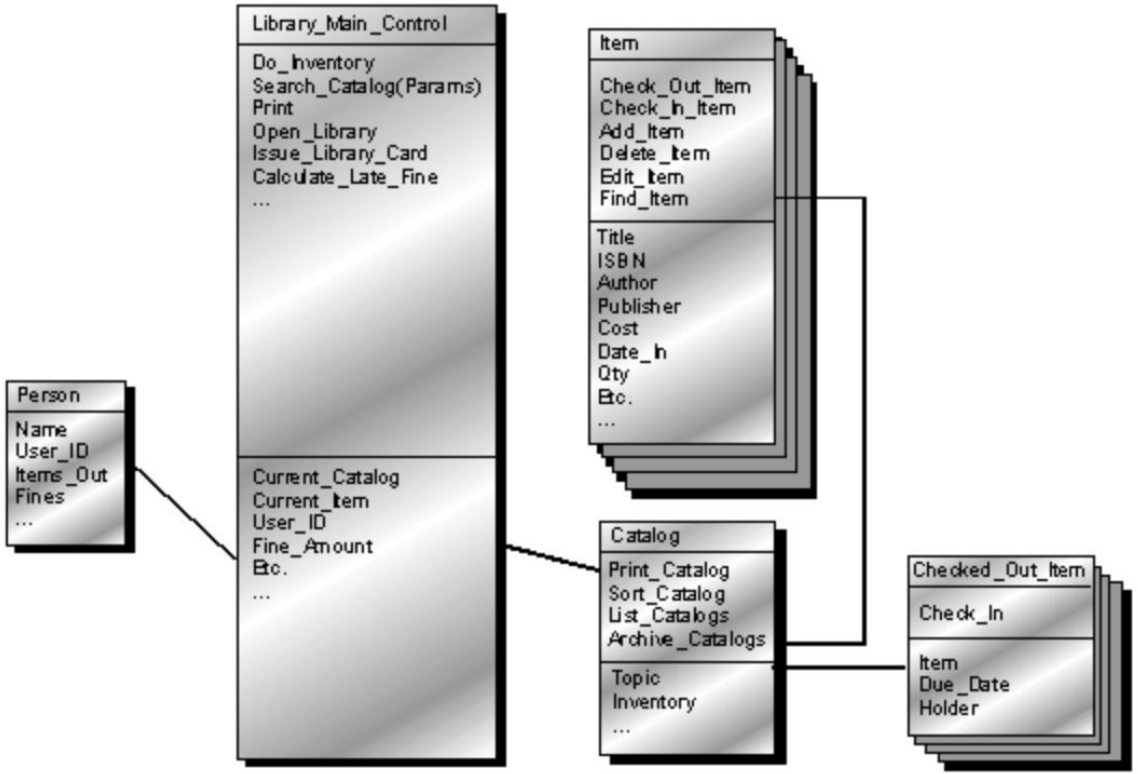
Final Step:
Remove all transient associations, replacing them as appropriate with type specifiers to attributes and operations arguments



MITRE

Development AntiPattern:

The Blob Refactored



Poltergeist

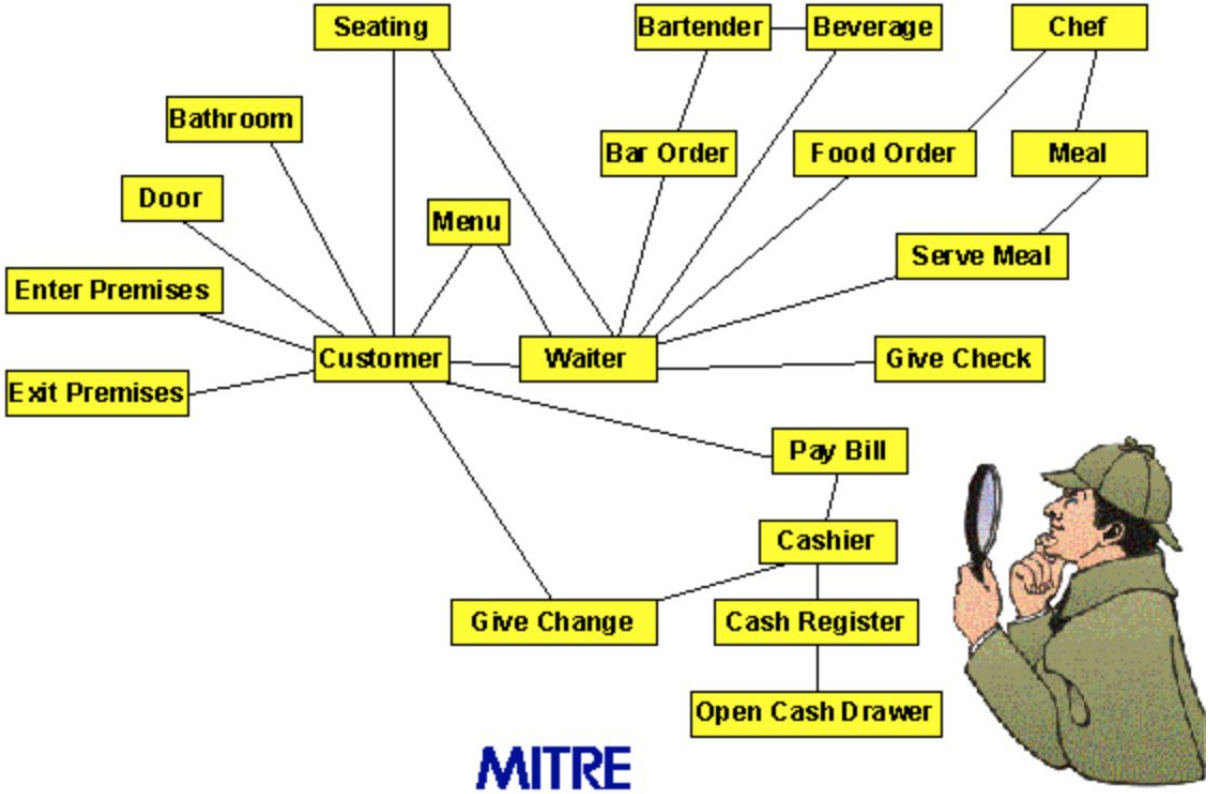
- Stateless object used to perform initialization or invoke methods of another more permanent class.
- Leads to lack of extensibility, poor performance, excessive complexity.
- Often labelled “_manager”, “_controller”, “start_process”
- Not to be confused with long-lived, state-bearing objects of a pattern, such as model-view-controller

Poltergeist Refactoring

- Eliminate irrelevant classes
  - Delete external classes outside the system
  - Delete classes with no domain relevance
- Eliminate transient data classes and operation classes
- Refactor other classes with short life cycles or few responsibilities
  - Move their behavior into collaborating classes
  - Regroup into cohesive larger classes

Poltergeist Refactoring Exercise

Development AntiPattern:
Poltergeists Example



Stovepipe System

- An ad-hoc collection of poorly-related components, leading to a brittle architecture.
- Often the result of:
  - Architecture by implication
  - Lack of agreement between software architects
  - Having no architect, or an architect disconnected from the source code

Architecture AntiPattern:

Stovepipe System - Example

Stovepipe System Refactoring

- Refactoring a stovepipe system:
  - Identify and isolate components
  - Establish interfaces and APIs
  - Incremental refactoring
  - Encapsulate legacy code
  - Use design patterns
  - Continuous integration and deployment
- Practices to prevent stovepipe systems:
  - Plan ahead! Invest in documentation
  - Design for modularity
  - DRY
  - Agile development

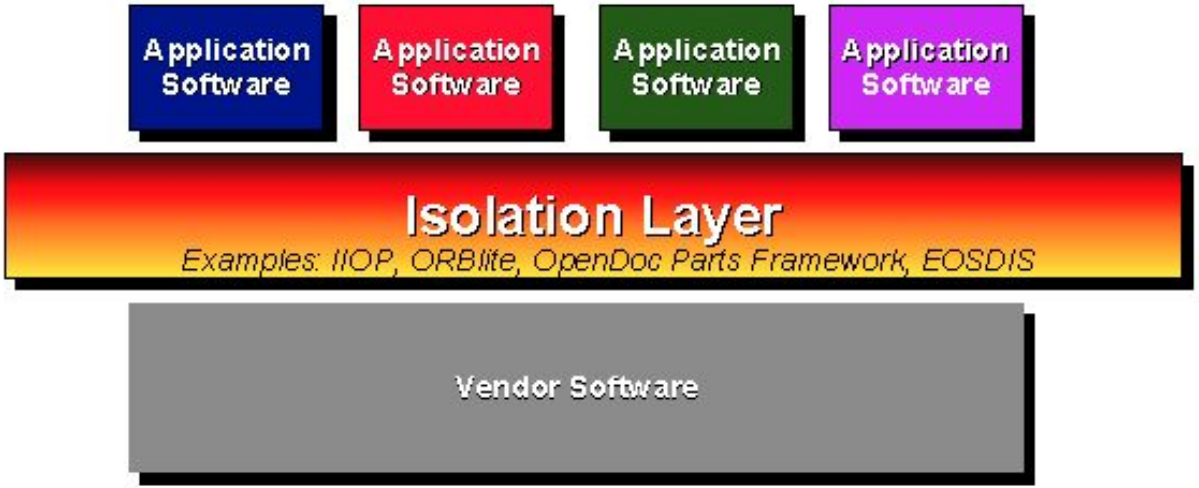
Vendor Lock-In

- When you start building on top of a vendor’s proprietary software, you face:
  - Loss of control:
    - the product may not meet your requirements,
    - updates are always on a separate timeline,
    - vendor changes the product, and your software is broken
  - High refactoring costs if you want to pivot to another vendor

Architecture AntiPattern:

Vendor Lock-In - Refactored Solution

Solution Strategy: Isolation Layer



Lava Flow

- Dead or obsolete code that remains in the codebase, making it harder to understand and maintain

- How to prevent:
  - Leverage git. Ensure that you're isolating new features to their own branches, and optimize the code within them before merging them with your main branch.
  - Regular code reviews
  - Continuous refactoring

Code Smells

- Indicate violation of fundamental design principles, negatively impacting design quality.
- They don't prevent the program from functioning
- Hint at the presence of other antipatterns
- Code where Consequences > Benefits
- Common code smells: long methods, large classes, duplicated code, excessive parameters
- Refactoring:
  - Engineer for optimal solutions.
  - DRY

Magic Numbers and Strings

- Hard-coded values without clear meaning or context
- Problems caused by Magic Numbers and Strings: reduced readability, maintainability, and potential for errors
- Techniques to avoid and refactor: use named constants, configuration files, or enumerations

Anemic Domain Model

- Classes only hold data without encapsulating behavior
- Consequences: poor cohesion, violates Object-Oriented principles
- Improve by adding behavior and encapsulating logic within domain classes

Data Clumps

- Groups of variables that are consistently used together, indicating a missing abstraction
- Problems caused by Data Clumps: reduced readability, code duplication, lack of cohesion
- Strategies to avoid and refactor: extract classes, encapsulate related data into a single entity

Speculative Generality

- Adding code or abstractions to support anticipated future needs that may never arise
- Consequences: unnecessary complexity, increased maintenance effort, decreased readability
- How to prevent and refactor: follow YAGNI principle (You Aren't Gonna Need It), remove unused abstractions, focus on current requirements

Copy-Paste Coding

- Nothing explicitly wrong with using code found online, libraries, etc.
  - Becomes problematic if you implement without understanding
- Works until it doesn't (something gets refactored)
- Black box trade off

Questions?

Demo Time

Improvements to the The Coug's backend?

References

- Brown, W. J. (1998). *AntiPatterns: Refactoring software, Architectures, and projects in crisis*. John Wiley & Sons.
- Jimenez, E. (n.d.). Antipatterns. "AntiPatterns". Link
- Koenig, Andrew (March–April 1995). "Patterns and Antipatterns".

  Journal of Object-Oriented Programming.
- *What are software Anti-Patterns?* Lucidchart. (n.d.). Link

Parallel Inheritance Hierarchies

- Two or more class hierarchies that evolve together, mirroring each other's changes
- Problems caused by Parallel Inheritance Hierarchies: increased complexity, tight coupling, code duplication
- Strategies to avoid and refactor: merge hierarchies, use composition over inheritance, apply Bridge pattern

Divergent Change

- A class is frequently modified for different reasons, violating Single Responsibility Principle
- Issues associated with Divergent Change: increased maintenance effort, reduced cohesion, instability
- Techniques to avoid or refactor: apply Single Responsibility Principle, extract classes, modularize code

Singleton Abuse

- Recall: Singleton is a design pattern that restricts instantiation of a class to one object.
- Drawbacks: global state, tight coupling, hard to test
- Alternatives: Dependency Injection, Factory Pattern, Service Locator

Shotgun Surgery

- A single change requires multiple updates across the codebase
- Issues caused: high maintenance cost, error-prone, poor encapsulation
- How to prevent and refactor: consolidate responsibilities, apply Single Responsibility Principle