

Refactoring

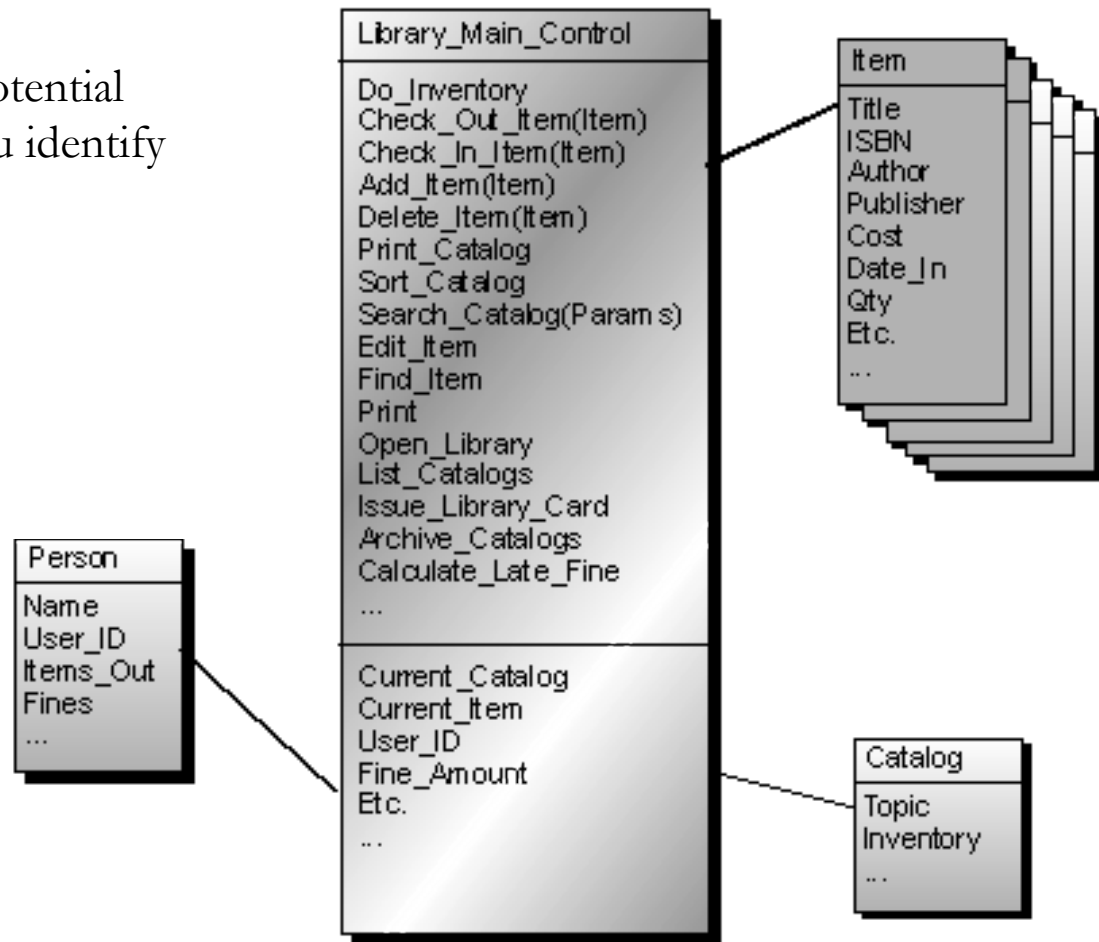
(Some material adapted from Martin Fowler's book)

Lehman & Belady: Laws of Software Evolution (1974)

- **Continuing Change** - Systems must be continually adapted else they become progressively less satisfactory.
 - **Increasing Complexity** - As a system evolves its complexity increases **unless work is done to maintain or reduce it.**
-

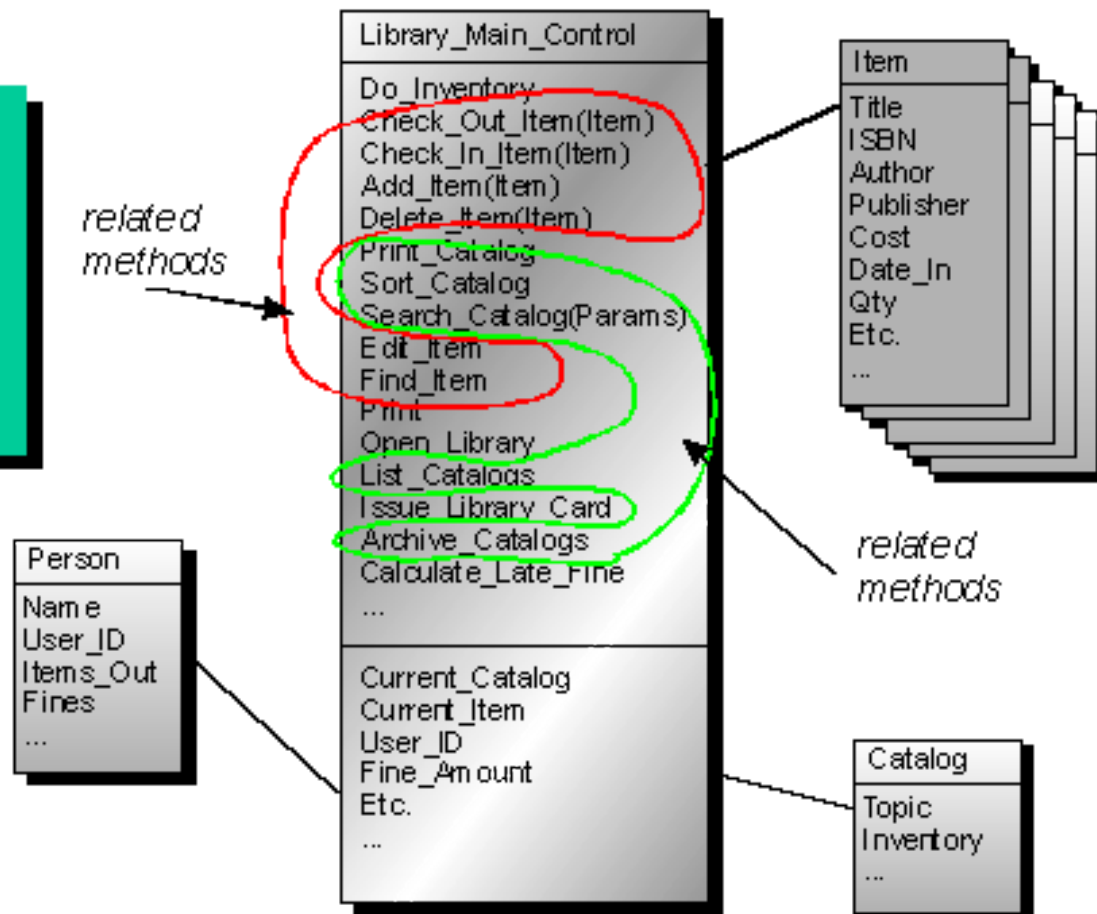
Library system – Existing design

What areas do you see as potential problem areas? Why did you identify each of those areas?



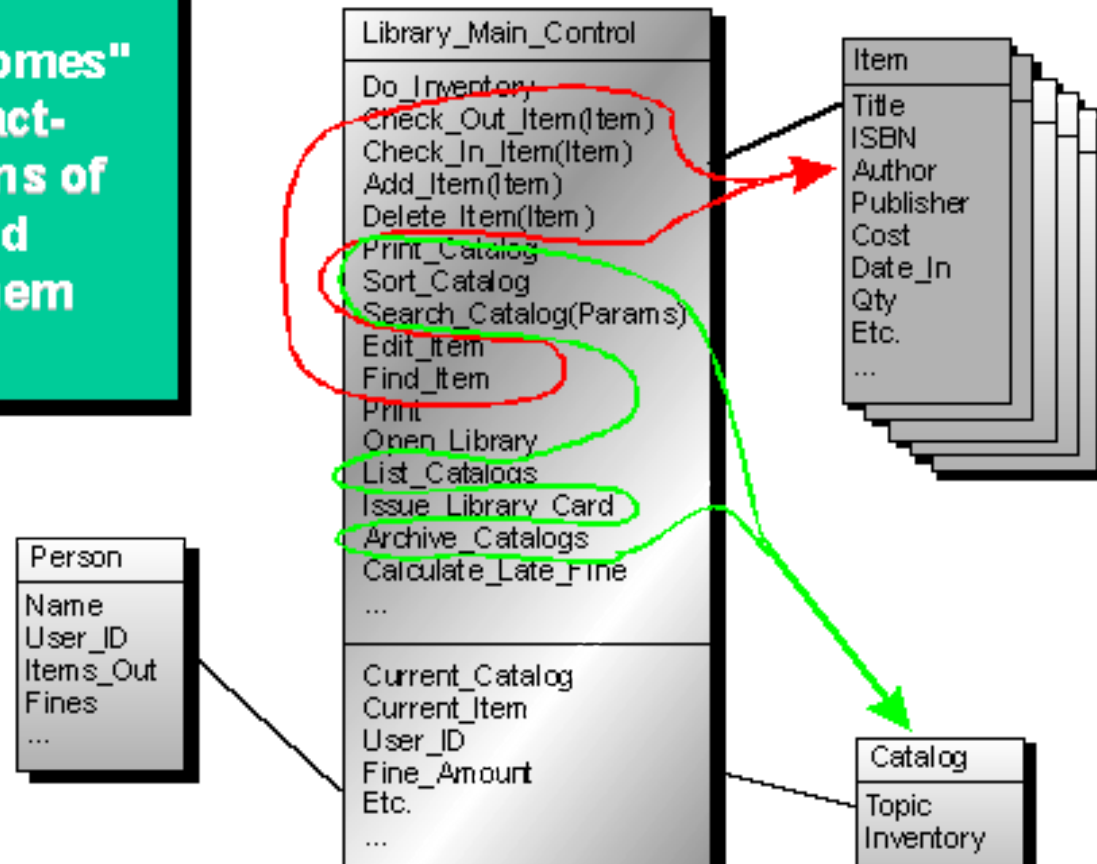
Library system – Changing the design

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



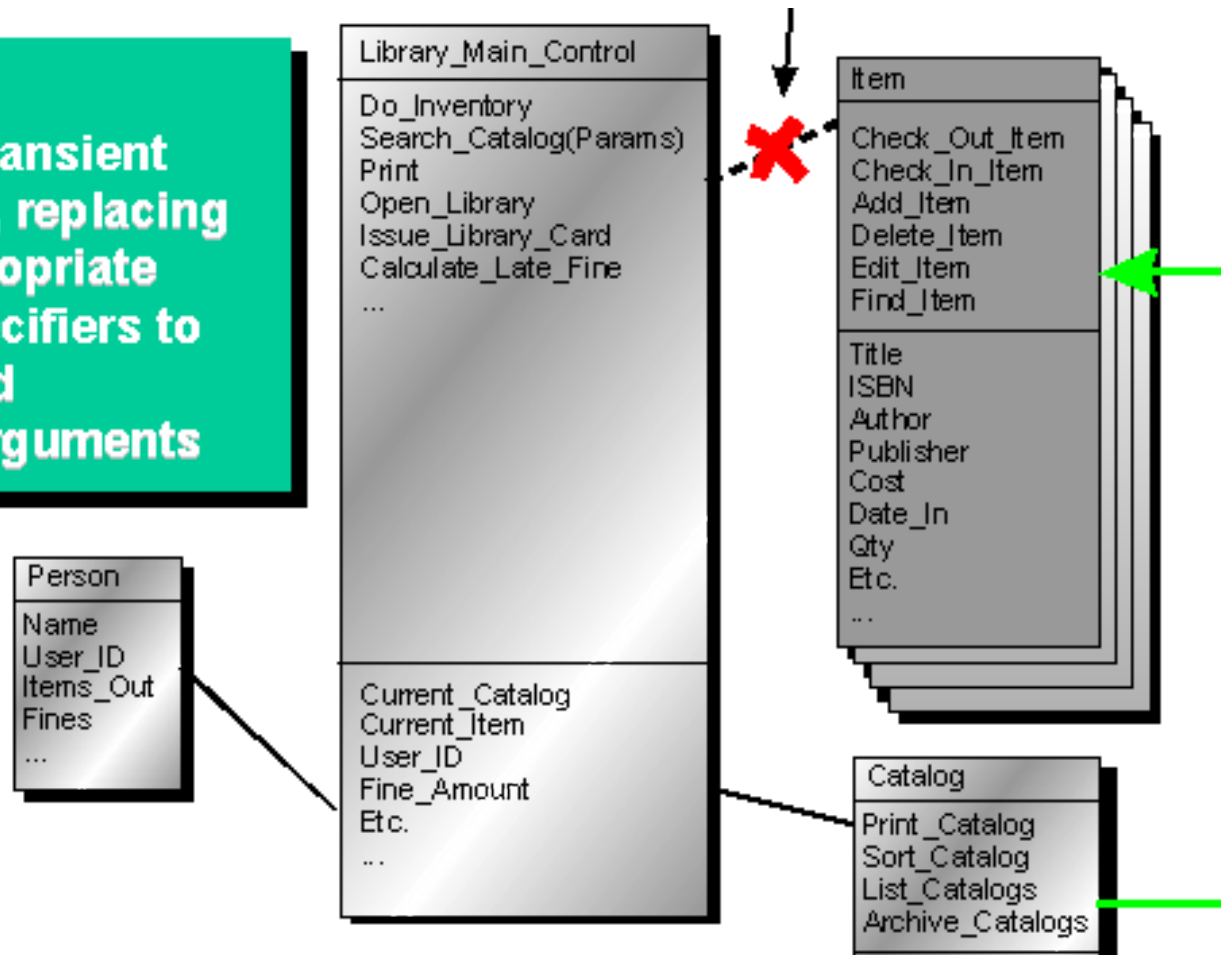
Library system – Changing the design

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

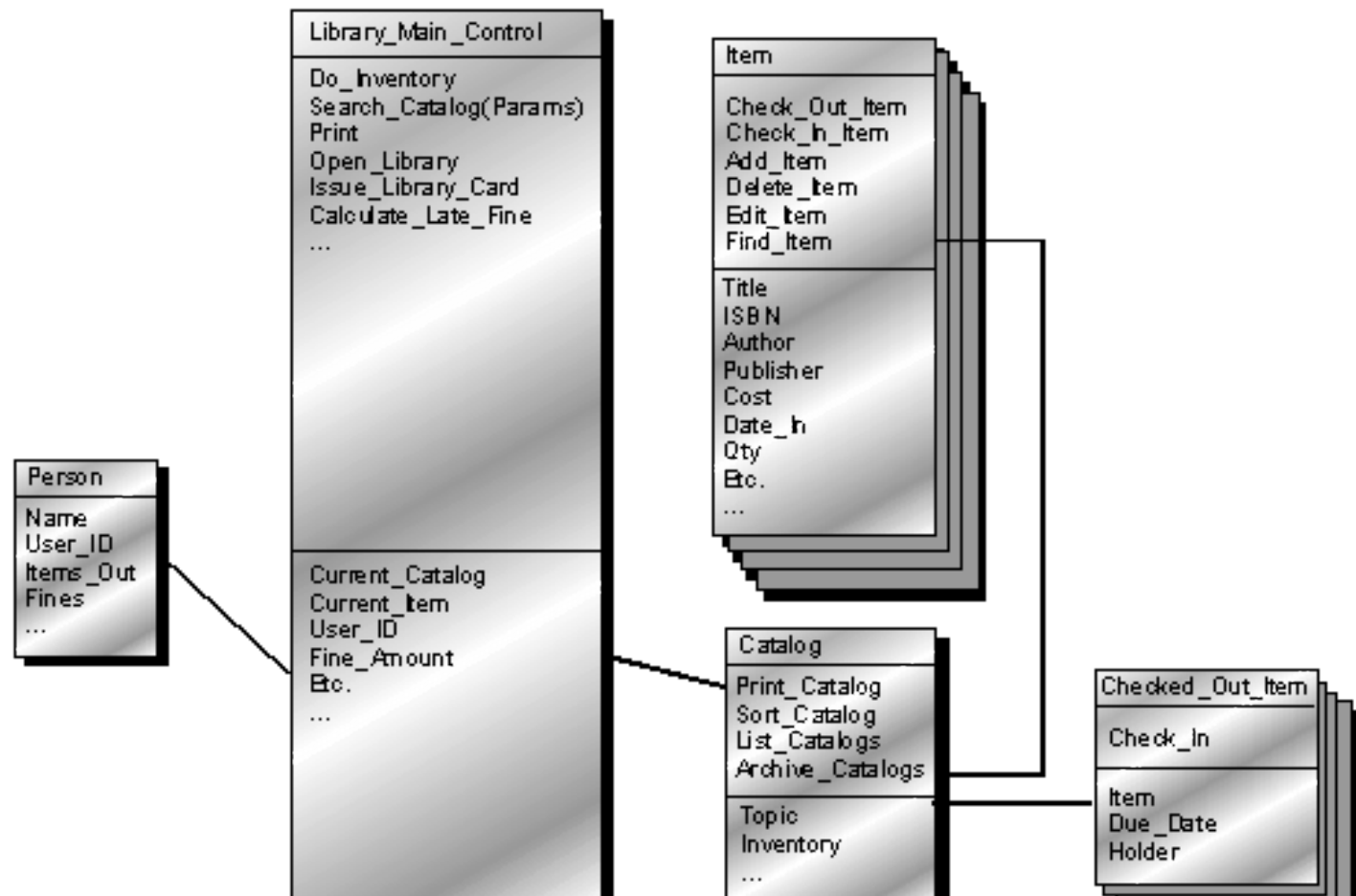


Library system – Changing the design

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



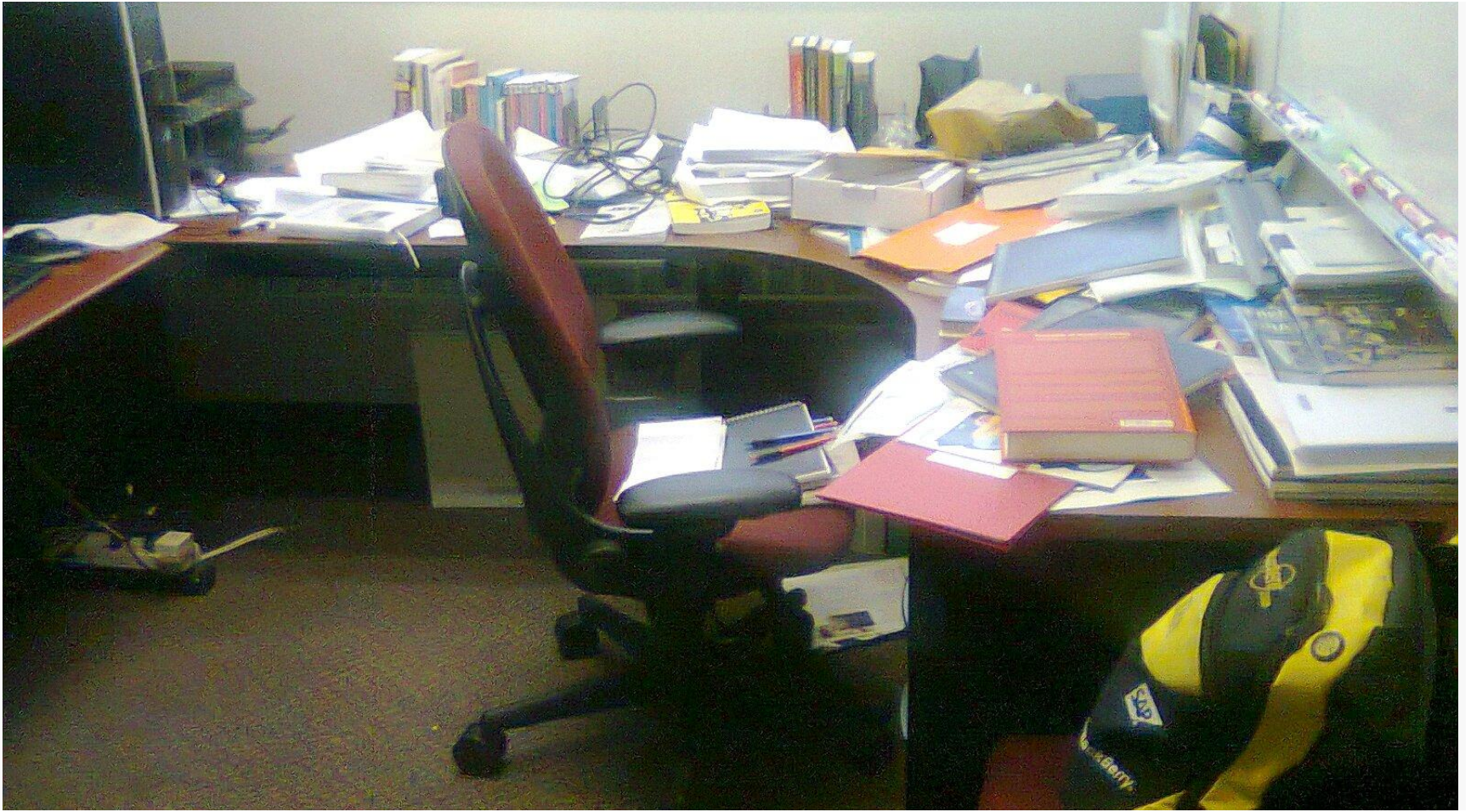
Library system – Changing the design



It is usually hard to counter, “If it ain’t broke, don’t fix it.”

- Generally improves product quality
 - Pay today to ease work tomorrow
 - May actually accelerate today’s work
-

From this...



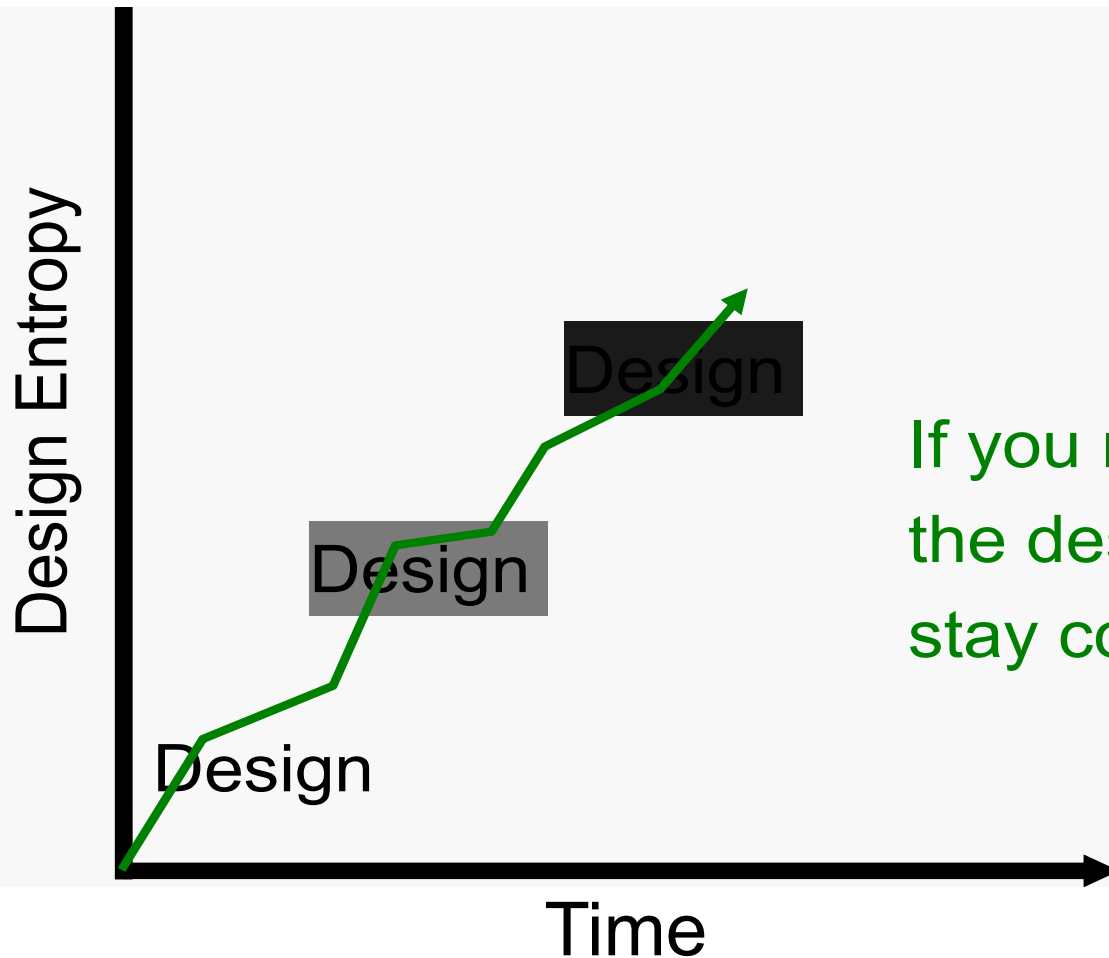
Source: Mike Lutz, RIT

... To this



Source: Mike Lutz, RIT

Design Entropy Vs Time



If you no longer can see the design, how can you stay consistent to it?

What causes Design Smells?

- Violation of design principles
 - `java.util.Calendar`
 - Violates Single responsibility principle (overloaded with date functionality and time functionality)
 - `java.util.Stack`
 - extends `java.util.Vector`
 - Do Stack and Vector share a IS-A relationship?
 - Violates principle of hierarchy (Broken Hierarchy – substitutability is broken)

What causes Design Smells?

- Inappropriate use of patterns
 - Language limitations
 - Procedural thinking in OO
 - Viscosity
 - Software viscosity
 - Environment viscosity
 - Non-adherence to best practices and processes
-

Design Smell classification

Abstraction

- Missing Abstraction
- Imperative Abstraction
- Incomplete Abstraction
- Multifaceted Abstraction
- Unnecessary Abstraction
- Unutilized Abstraction
- Duplicate Abstraction

Modularization

- Broken Modularization
- Insufficient Modularization
- Cyclically-dependent Modularization
- Hub-like Modularization

Design Smell classification

Encapsulation

- Deficient Encapsulation
- Leaky Encapsulation
- Missing Encapsulation
- Unexploited Encapsulation

Modularization

- Missing Hierarchy
- Unnecessary Hierarchy
- Unfactored Hierarchy
- Wide Hierarchy
- Speculative Hierarchy
- Deep Hierarchy
- Rebellious Hierarchy
- Broken Hierarchy
- Multipath Hierarchy
- Cyclic Hierarchy

Code Smells Within Classes

■ Comments

- ❑ Are the comments necessary?
- ❑ Do they explain "why" and not "what"?
- ❑ Can you refactor the code so the comments aren't required?
- ❑ Remember, you're writing comments for people, not machines.

■ Long Method

- ❑ Shorter method is easier to read, easier to understand, and easier to troubleshoot.
- ❑ Refactor long methods into smaller methods if you can

■ Long Parameter List

- ❑ The more parameters a method has, the more complex it is.
- ❑ Limit the number of parameters you need in a given method, or use an object to combine the parameters.

■ Duplicated code

- ❑ Stamp out duplication whenever possible.
- ❑ Don't Repeat Yourself!

Code Smells Within Classes

■ Conditional Complexity

- ❑ large conditional logic blocks, particularly blocks that tend to grow larger or change significantly over time.
- ❑ Consider alternative object-oriented approaches such as decorator, strategy, or state.

■ Combinatorial Explosion

- ❑ Lots of code that does *almost* the same thing.. but with tiny variations in data or behavior.
- ❑ This can be difficult to refactor-- perhaps using generics or an interpreter?

■ Large Class

- ❑ Large classes, like long methods, are difficult to read, understand, and troubleshoot.
- ❑ Large class can be restructured or broken into smaller

Code Smells Within Classes

■ Uncommunicative Name

- ❑ Does the name of the method succinctly describe what that method does? Could you read the method's name to another developer and have them explain to you what it does?

■ Inconsistent Names

- ❑ set of standard terminology and stick to it throughout your methods.

■ Dead Code

- ❑ Ruthlessly delete code that isn't being used

■ Speculative Generality

- ❑ Write code to solve today's problems, and worry about tomorrow's problems when they actually materialize.
- ❑ Everyone loses in the "what if.." school of design.

Code Smells Between Classes

■ Alternative Classes with Different Interfaces

- If two classes are similar on the inside, but different on the outside, perhaps they can be modified to share a common interface.

■ Primitive Obsession

- If data type is sufficiently complex, write a class to represent it.

■ Data Class

- Avoid classes that passively store data.
- Classes should contain data *and* methods to operate on that data, too.

■ Data Clumps

- If you always see the same data hanging around together, maybe it belongs together.
- Consider rolling the related data up into a larger class.

Code Smells Between Classes

■ **Refused Bequest**

- ❑ Inherit from a class but never use any of the inherited functionality

■ **Inappropriate Intimacy**

- ❑ Classes that spend too much time together, or classes that interface in inappropriate ways.
- ❑ Classes should know as little as possible about each other

■ **Indecent Exposure**

- ❑ Classes that unnecessarily expose their internals.
- ❑ Aggressively refactor classes to minimize their public surface.
- ❑ You should have a compelling reason for every item you make public. If you don't, hide it.

■ **Feature Envy**

- ❑ Methods that make extensive use of another class may belong in another class.
- ❑ Move the method to the class it is so envious

Code Smells between Classes

■ Lazy Class

- ❑ Classes should pull their weight.
- ❑ If a class isn't doing enough to pay for itself, it should be collapsed or combined into another class.

■ Message Chains

- ❑ Long sequences of method calls or temporary variables to get routine data.
- ❑ Intermediaries are dependencies in disguise.

■ Middle Man

- ❑ If a class is delegating all its work., then cut out the middleman.
- ❑ Beware classes that are merely wrappers over other classes or existing functionality in the framework.

■ Divergent Change

- ❑ If changes to a class touch completely different parts of the class, it may contain too much unrelated functionality.
- ❑ Isolate the parts that changed in another class.

Code Smells between Classes

■ Shotgun Surgery

- ❑ If a change in one class requires cascading changes in several related classes

■ Parallel Inheritance Hierarchies

- ❑ Every time you make a subclass of one class, you must also make a subclass of another.
- ❑ Consider folding the hierarchy into a single class.

■ Solution Sprawl

- ❑ If it takes five classes to do anything useful, you might have solution sprawl.
 - ❑ Consider simplifying and consolidating your design.
-

Refactoring

- As a software system grows, the overall design often suffers
- In the short term, working in the existing design is cheaper than doing a redesign
- In the long term, the redesign decreases total costs
 - Extensions
 - Maintenance
 - Understanding
- Refactoring is a set of techniques that reduce the short-term pain of redesigning
 - Not adding functionality
 - Changing structure to make it easier to understand and extend

The Scope of Refactoring

- Small steps:
 - Rename a method
 - Move a field from one class to another
 - Merge two similar methods in different classes into one common method in a base class
 - Each individual step is small, and easily verified/
tested
 - The composite effect can be a complete
transformation of a system
-

Principles

- Don't refactor and extend a system at the same time
 - Make a clear separation between the two activities
- Have good tests in place before you begin refactoring
 - Run the tests often
 - Catch defects immediately
- Take small steps
 - Many localized changes result in a larger-scale change
 - Test after each small step

When Should You Refactor?

- You're extending a system, and realize it could be done better by changing the original structure
 - Stop and refactor first
- The code is hard to understand
 - Refactor to gain understanding, and leave the code better than it was

Refactoring and OOD

- The refactoring literature is written from a coding perspective
 - Many of the operations still apply at design time
 - It helps if you have an appropriate level of detail in the design
 - Too much, and you may as well code
 - Too little, and you can't tell what's happening
-

A decorative L-shaped line in a golden-brown color, consisting of a horizontal segment and a vertical segment meeting at a right angle.

AntiPatterns

A horizontal line in a golden-brown color, positioned below the title.

AntiPatterns

- A pattern of practice that is commonly found in use
 - A pattern which when practiced usually results in *negative* consequences
 - Patterns defined in several categories of software development
 - Design
 - Architecture
 - Project Management
-

Purpose for AntiPatterns

- Identify problems
 - Develop and implement strategies to fix
 - Work incrementally
 - Many alternatives to consider
 - Beware of the cure being worse than the disease
-

Pattern Vs AntiPattern

■ Patterns

- ❑ Usually bottom up
- ❑ Begin with recurring solution
- ❑ Then the forces and context
- ❑ Usually leads to one solution

■ AntiPatterns

- ❑ Top down
- ❑ Begin with commonly recurring practice
- ❑ Obvious negative consequences
- ❑ Symptoms are past and present; consequences go into the future

Software Design AntiPatterns

■ AntiPatterns

- ❑ The Blob
- ❑ Lava Flow
- ❑ Functional Decomposition
- ❑ Poltergeists
- ❑ Golden Hammer
- ❑ Spaghetti Code
- ❑ Cut-and-Paste Programming

■ Mini-AntiPatterns

- ❑ Continuous Obsolescence
- ❑ Ambiguous Viewpoint
- ❑ Boat Anchor
- ❑ Dead End
- ❑ Input Kludge
- ❑ Walking through a Minefield
- ❑ Mushroom Management

The Blob

- AKA
 - Winnebago, The God Class, Kitchen Sink Class
- Causes
 - Sloth, haste
- Unbalanced Forces:
 - Management of Functionality, Performance, Complexity
- Anecdotal Evidence:
 - “This is the class that is really the *heart* of our architecture.”

The Blob (2)

- Like the blob in the movie can consume entire object-oriented architectures
- Symptoms
 - ❑ Single controller class, multiple simple data classes
 - ❑ No object-oriented design, i.e. all in main
 - ❑ Start with a legacy design
- Problems
 - ❑ Too complex to test or reuse
 - ❑ Expensive to load into system

Causes

- Lack of OO architecture
- Lack of any architecture
- Lack of architecture enforcement
- Limited refactoring intervention
- Iterative development
 - Proof-of-concept to prototype to production
 - Allocation of responsibilities not repartitioned

Solution

- Identify or categorize related attributes and operations
 - Migrate functionality to data classes
 - Remove far couplings and migrate to data classes
-

Lava Flow

- AKA
 - Dead Code
 - Causes
 - Avarice, Greed, Sloth
 - Unbalanced Forces
 - Management of Functionality, Performance, Complexity
-

Symptoms and Consequences

- Unjustifiable variables and code fragments
 - Undocumented complex, important-looking functions, classes
 - Large commented-out code with no explanations
 - Lot's of “to be replaced” code
 - Obsolete interfaces in header files
 - Proliferates as code is reused
-

Causes

- Research code moved into production
 - Uncontrolled distribution of unfinished code
 - No configuration management in place
 - Repetitive development cycle
-

Solution

- Don't get to that point
 - Have stable, well-defined interfaces
 - Slowly remove dead code; gain a full understanding of any bugs introduced
 - Strong architecture moving forward
-

Functional Decomposition

- AKA
 - No OO
 - Root Causes
 - Avarice, Greed, Sloth
 - Unbalanced Forces
 - Management of Complexity, Change
 - Anecdotal Evidence
 - “This is our ‘main’ routine, here in the class called Listener.”
-

Symptoms and Consequences

- Non-OO programmers make each subroutine a class
 - Classes with functional names
 - ❑ Calculate_Interest
 - ❑ Display_Table
 - Classes with single method
 - No leveraging of OO principles
 - No hope of reuse
-

Causes

- Lack of OO understanding
 - Lack of architecture enforcement
 - Specified disaster
-

Solution

- Perform analysis
 - Develop design model that incorporates as much of the system as possible
 - For classes outside model:
 - Single method: find home in existing class
 - Combine classes
-

Poltergeists

- AKA
 - Gypsy, Proliferation of Classes
 - Root Causes
 - Sloth, Ignorance
 - Unbalanced Forces
 - Management of Functionality, Complexity
 - Anecdotal Evidence
 - “I’ m not exactly sure what this class does, but it sure is important.”
-

Symptoms and Consequences

- Transient associations that go “bump-in-the-night”
- Stateless classes
- Short-lived classes that begin operations
- Classes with control-like names or suffixed with *manager* or *controller*. Only invoke methods in other classes.

Causes

- Lack of OO experience
- Maybe OO is incorrect tool for the job. “There is no right way to do the wrong thing.”

Solution

- Remove Poltergeist altogether
- Move controlling actions to related classes

Cut-and-Paste Programming

- AKA
 - Clipboard Coding
- Root Causes
 - Sloth
- Unbalanced Forces
 - Management of Resources, Technology Transfer
- Anecdotal Evidence
 - “Hey, I thought you fixed that bug already, so why is it doing this again?” “Man, you guys work fast. Over 400,000 lines of code in three weeks is outstanding progress!”

Symptoms and Consequences

- Same software bug reoccurs
 - Code can be reused with a minimum of effort
 - Causes excessive maintenance costs
 - Multiple unique bug fixes develop
 - Inflates LOC without reducing maintenance costs
-

Causes

- Requires effort to create reusable code; must reward for long-term investment
- Context or intent of module not preserved
- Development speed overshadows all other factors
- “Not-invented-here” reduces reuse
- People unfamiliar with new technology or tools just modify a working example

Solution

- Code mining to find duplicate sections of code
 - Refactoring to develop standard version
 - Configuration management to assist in prevention of future occurrence
-

Golden Hammer

- AKA
 - Old Yeller
- Root Causes
 - Ignorance, Pride, Narrow-Mindedness
- Unbalanced Forces
 - Management of Technology Transfer
- Anecdotal Evidence
 - “Our database is our architecture” “Maybe we shouldn’ t have used Excel macros for this job after all.”

Symptoms and Consequences

- Identical tools for conceptually diverse problems.
“When your only tool is a hammer everything looks like a nail.”
- Solutions have inferior performance, scalability and other ‘ilities’ compared to other solutions in the industry.
- Architecture is described by the tool set.
- Requirements tailored to what tool set does well.

Causes

- Development team is highly proficient with one toolset.
 - Several successes with tool set.
 - Large investment in tool set.
 - Development team is out of touch with industry.
-

Solution

- Organization must commit to exploration of new technologies
 - Commitment to professional development of staff
 - Defined software boundaries to ease replacement of subsystems
 - Staff hired with different backgrounds and from different areas
 - Use open systems and architectures
-