
Anti-Patterns

CS356 Object-Oriented Design and Programming

<http://cs356.yusun.io>

December 1, 2014

Yu Sun, Ph.D.

<http://yusun.io>

yusun@csupomona.edu



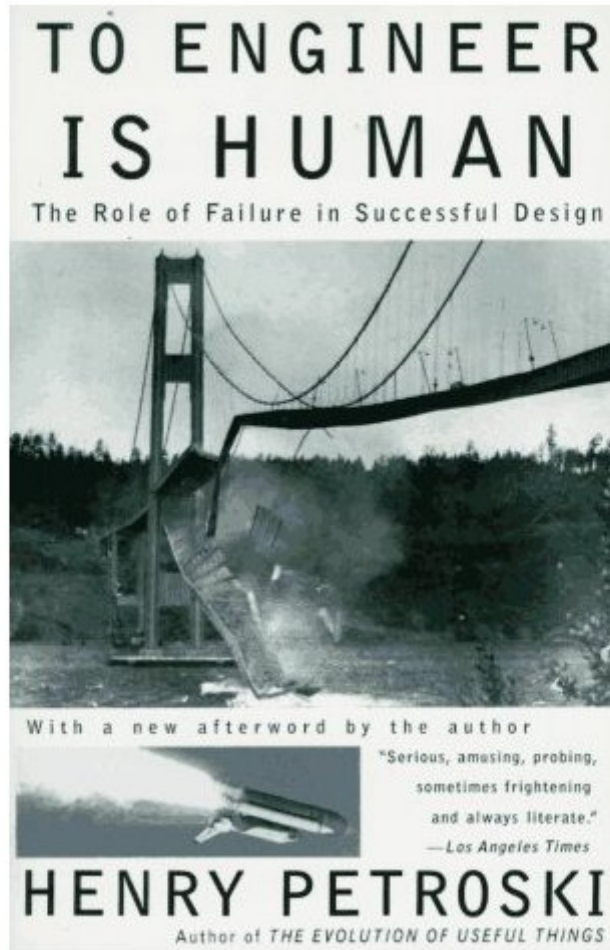
CAL POLY POMONA

Recap: Refactoring

- ◆ Improve the structure of code
 - ◆ No value gain at the moment, but
 - ◆ Easier to add features later
 - ◆ Less chances of errors in maintenance tasks
- ◆ Key is to preserve semantics (behavior)
 - ◆ Imprecisely ensure by developing tests
 - ◆ Also, by code inspection
- ◆ Often automated support for common refactorings
 - ◆ Automated support, less error prone
 - ◆ Often most general case
 - ◆ e.g. Eclipse extract method makes all variables parameters
 - ◆ Limitation of current program analysis techniques



Anti-Patterns

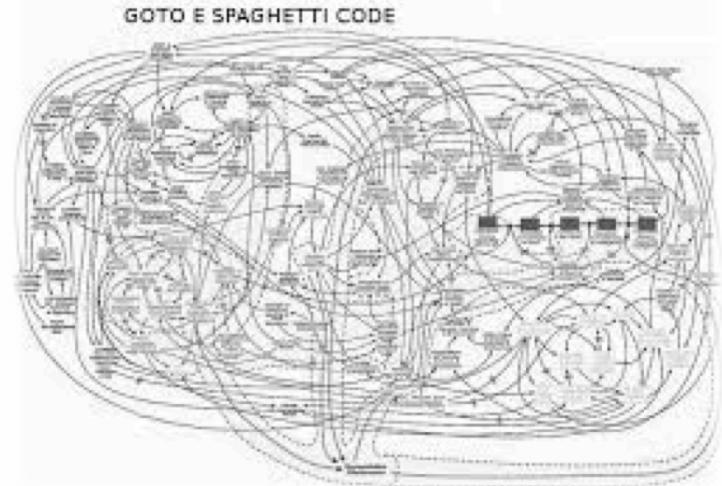


Lessons Learned
from failures
and their remedies

AntiPatterns: Vaccinations
against Object
Misuse” [Akroyd 96]

Example: Spaghetti Code

- ◆ An undocumented piece of source code
- ◆ Cannot be extended or modified
 - ◆ Reason: convoluted structure
 - ◆ Effect: significant cost in modification



Spaghetti Code: Symptoms

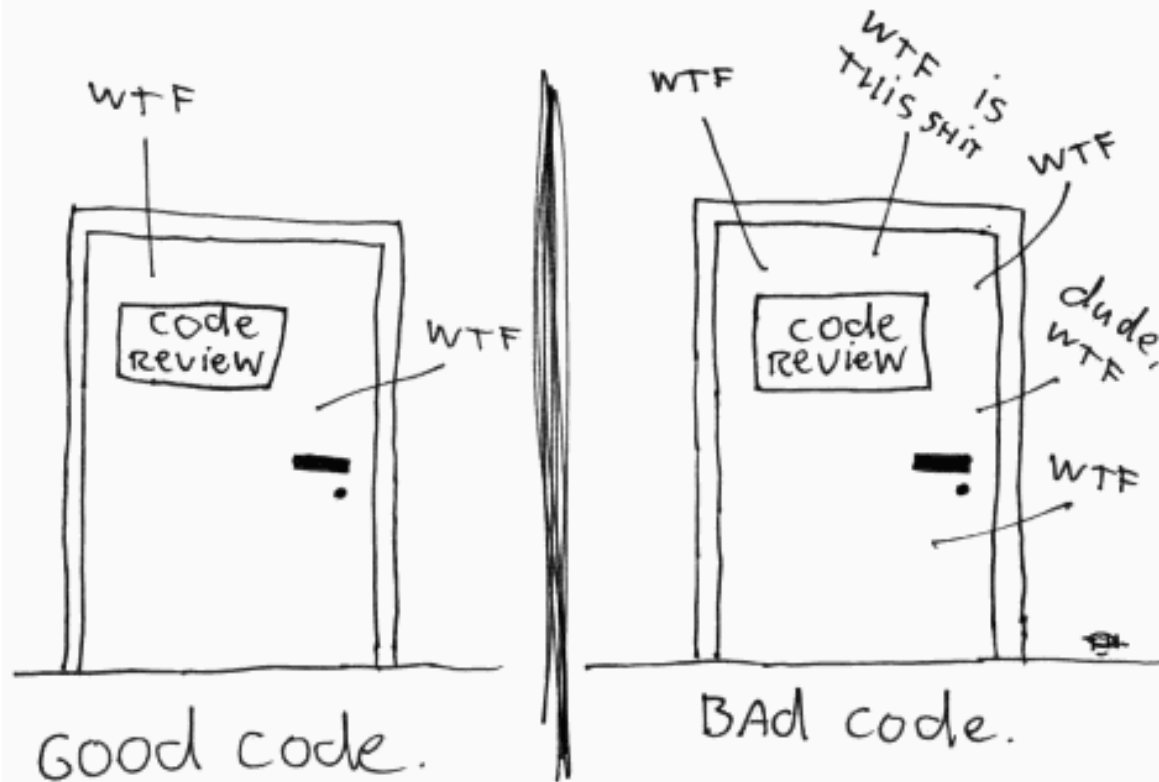
- ◆ Quick demonstration code integrated in the running system
- ◆ Obsolete or scanty documentation
- ◆ 50% time spent learning what the code does
- ◆ “Hesitant programmer syndrome”
 - ◆ Perhaps easier to rewrite this code
 - ◆ More likely to break it than extend it
- ◆ Cannot be reused
 - ◆ Cannot change the used library/components
 - ◆ Cannot optimize performance
- ◆ Duplication
 - ◆ “I don’t know what that piece of code was doing, so I rewrote what I thought should happen, but I cannot remove the redundant code because it breaks the system.”

Symptoms in an OO program

- ◆ Many OO method with no parameters
- ◆ Suspicious class or global variable
- ◆ Strange relationships between classes
- ◆ Process-oriented methods
 - ◆ Objects with process-oriented names
- ◆ OO advantage lost
 - ◆ Inheritance cannot be used to extend
 - ◆ Polymorphism cannot be used

Spaghetti Code: Symptoms

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



Solution

- **Refactor to generalize: *Create an abstract superclass***

1. Make subclass function signatures compatible
2. Add function signatures to the superclass
3. Make function bodies and variables compatible
4. Migrate common code to the superclass

- **Refactor to specialize: *Simplify conditionals***

1. For each condition, create a subclass with matching invariant
2. Copy the code into the subclass
3. Simplify code based upon invariant
4. Specialize the superclass constructor

- **Refactor to combine: *Capture aggregations and components***

- Type A. Move members from an aggregate class to a components class
- Type B. Move members from component classes to aggregate class
- Type C. Convert inheritance into an aggregation

Root Cause of Anti-Patterns: Haste



Root Cause of Anti-Patterns: Apathy



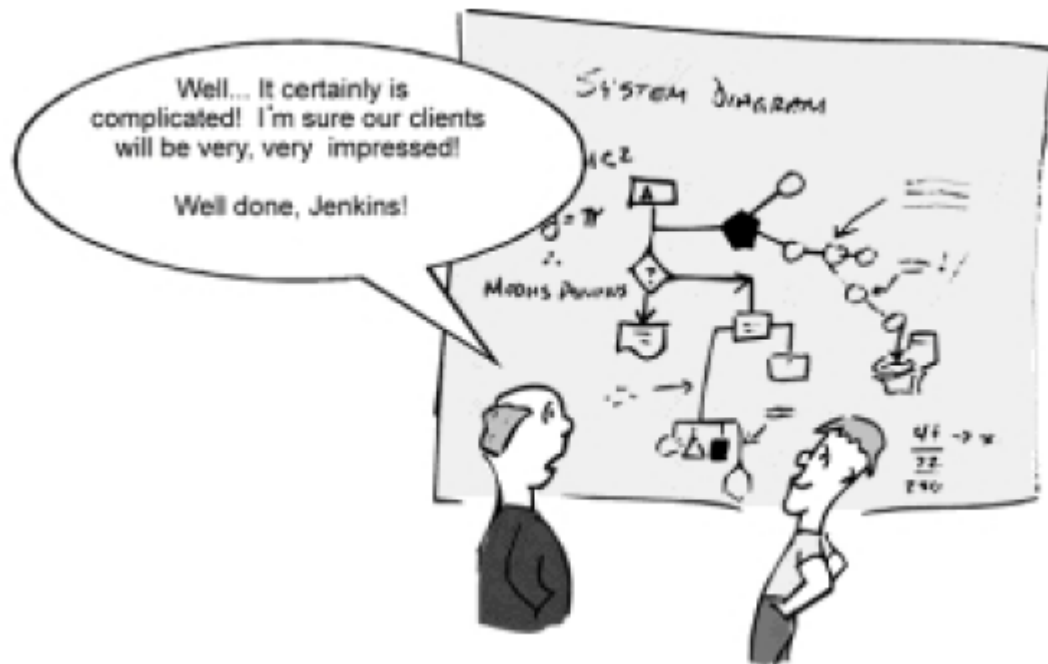
Root Cause of Anti-Patterns: Narrow-Mindedness



Root Cause of Anti-Patterns: Sloth



Root Cause of Anti-Patterns: Avarice



Root Cause of Anti-Patterns: Ignorance



Root Cause of Anti-Patterns: Pride

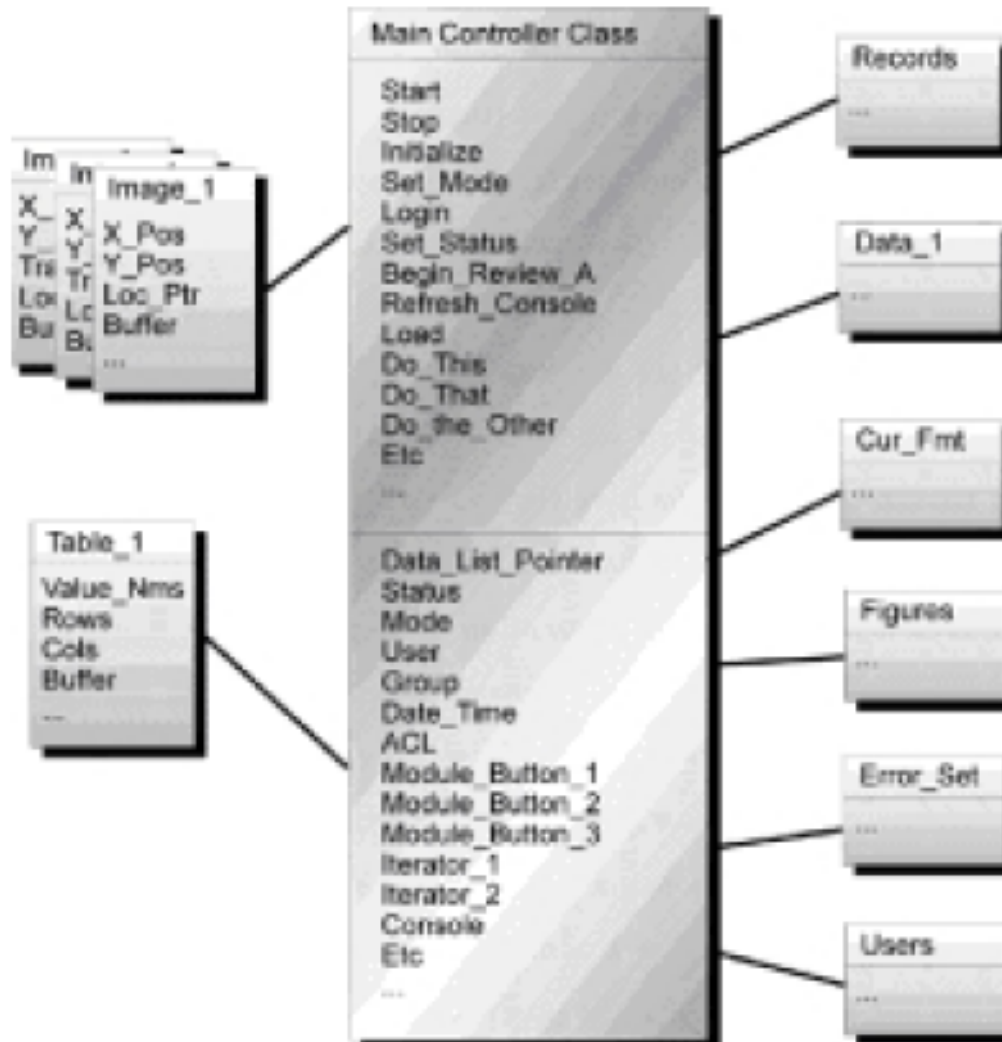


Another Anti Pattern: The BLOB

- ◆ Also known as
 - ◆ Winnebago and the God class
- ◆ Scale: Entire application
- ◆ General Form:
 - ◆ One class monopolizes the processing
 - ◆ Other classes are data classes



The Design of an Example Blob



Symptoms of a Blob

- ◆ Single Class
 - ◆ Large number of attributes
 - ◆ Large number of operations
- ◆ Unrelated attributes and operations
 - ◆ Overall lack of cohesiveness
- ◆ Too complex to reuse and test
- ◆ Expensive to load into memory



Refactored Solution

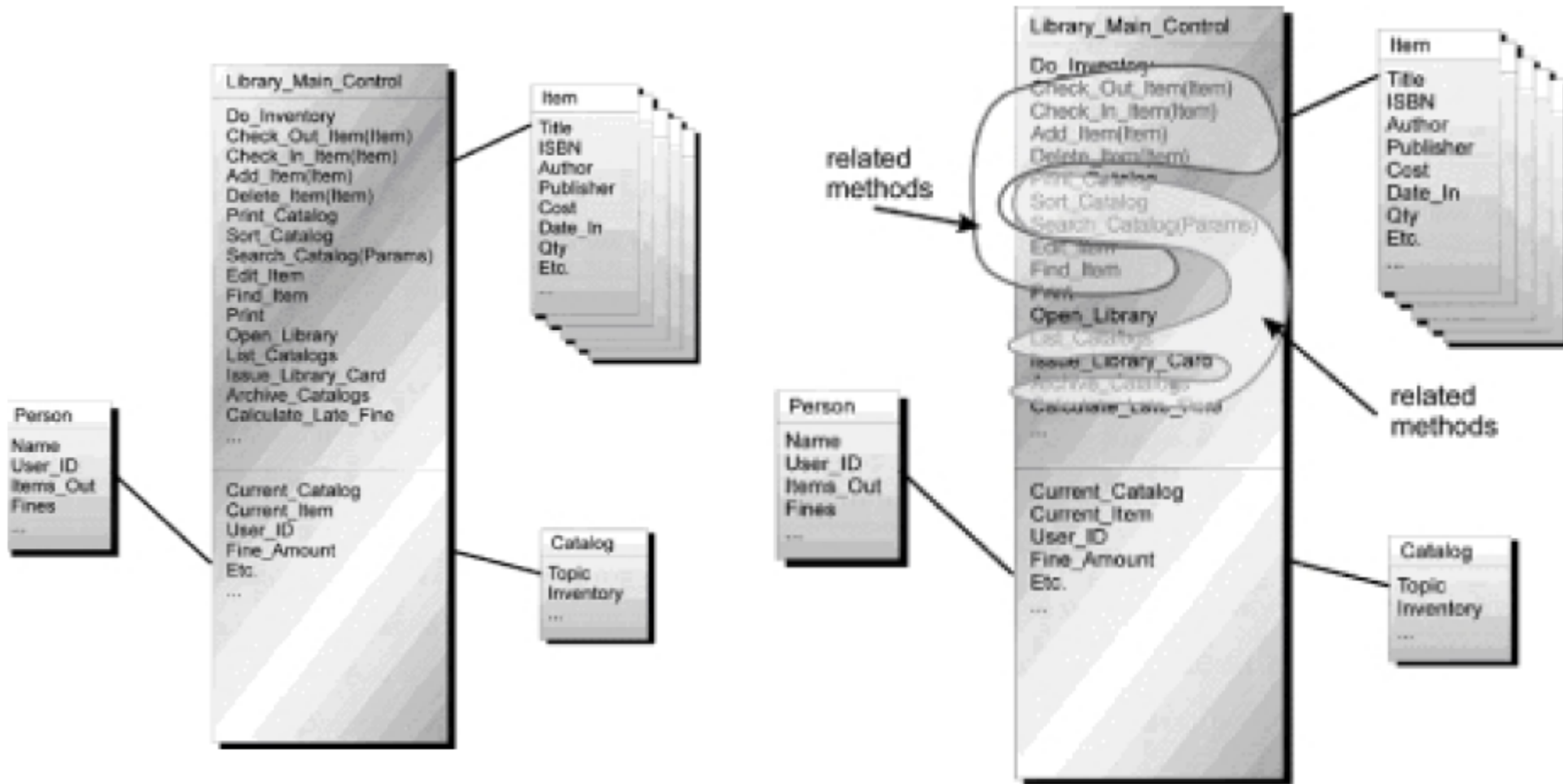
- ◆ Identify or categorize related things
 - ◆ Attributes, Operations
- ◆ Where do these categories naturally belong?
 - ◆ Apply move method, move field refactorings
- ◆ Remove redundant associations

```
function Match-Expression($string, $regex)
{
    if ($string -match $regex)
    { $Matches[1..($Matches.Count-1)] }
    else { @() }
}

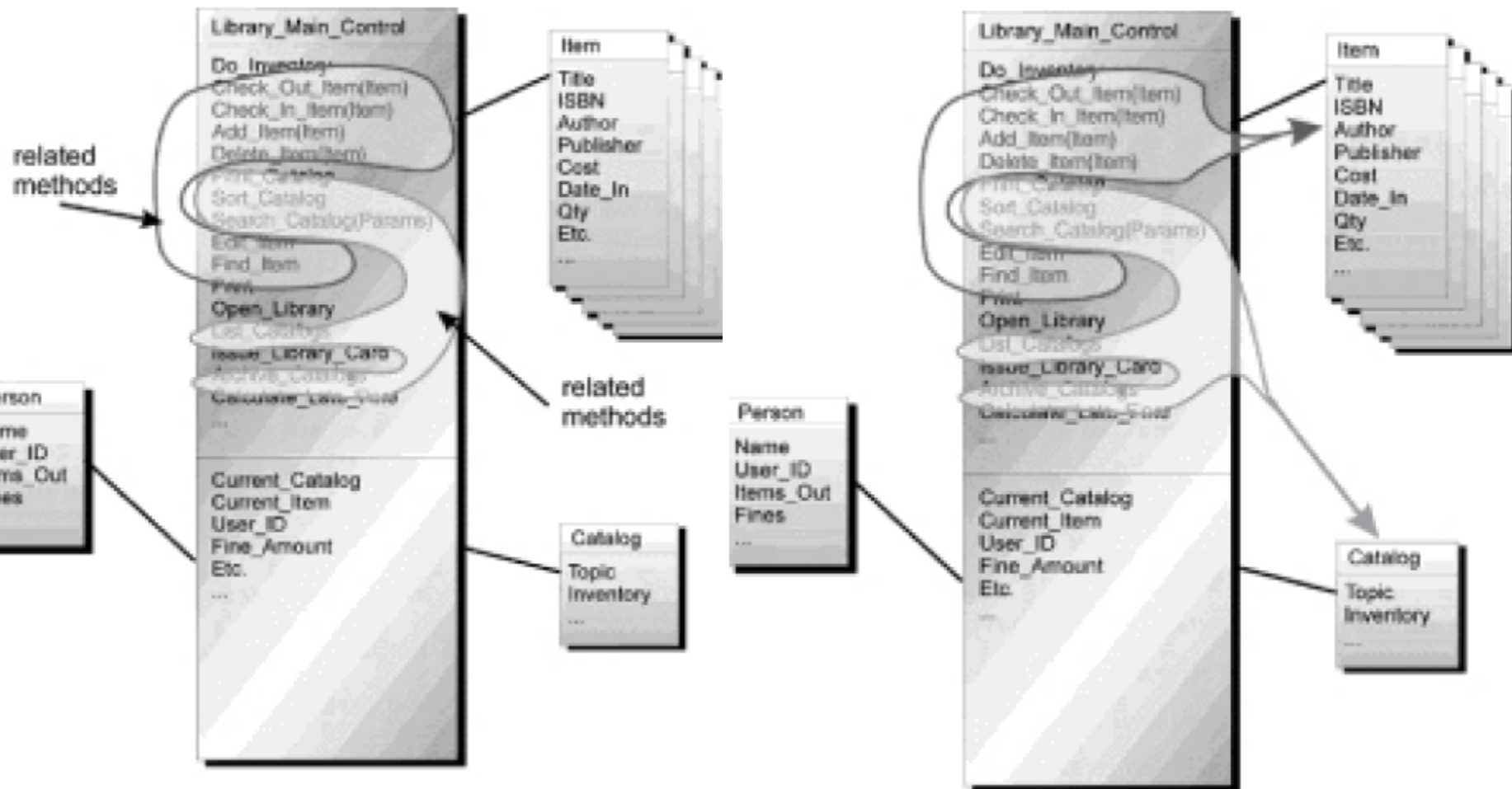
$yearValue = (Get-Date).Year
$froozle = "A.{0}-MarkIV" -f $yearValue
$regex = "([A-Z]+\)\. (\d+)- (\w+)"
$term = null
$year = null
$rendition = null
if ($froozle -match $regex) {
    $term = $Matches[1]
    $year = $Matches[2]
    $rendition = $Matches[3]
}
if ($term -like "A*") {
    Process-Stuff ...
}

$yearValue = (Get-Date).Year
$froozle = "A.{0}-MarkVI" -f $yearValue
$regex = "([A-Z]+\)\. (\d+)- (\w+)"
$term, $year, $rendition =
    Match-Expression $froozle $regex
if ($term -like "A*") {
    Process-Stuff ...
}
```

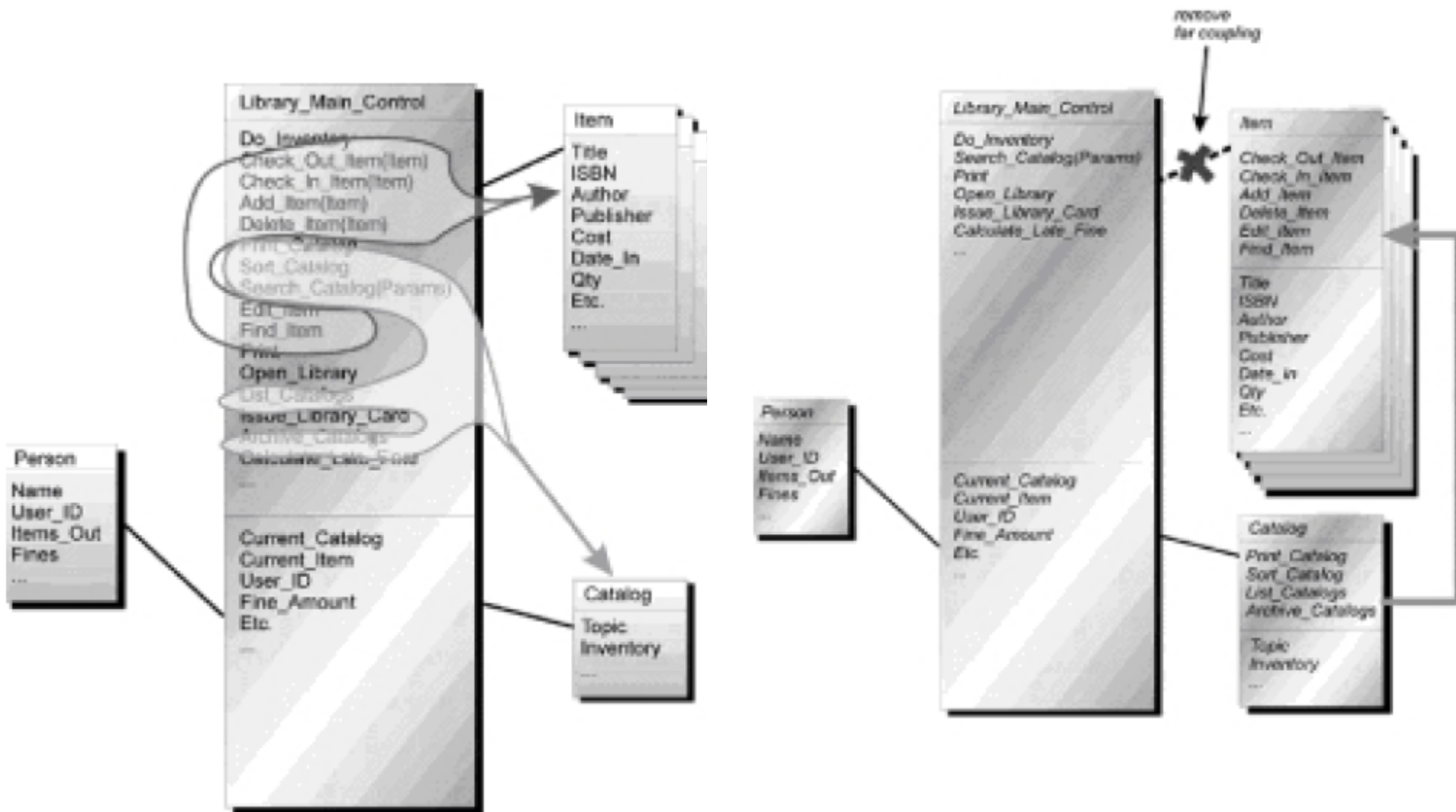
Categories in Example Application



Migration in Example Application



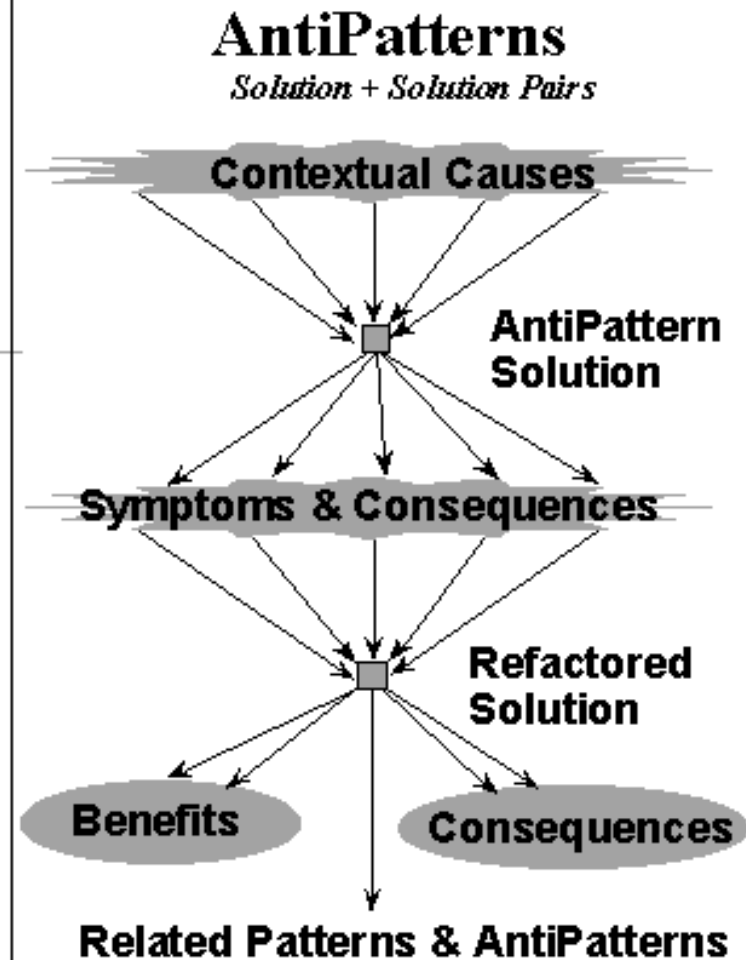
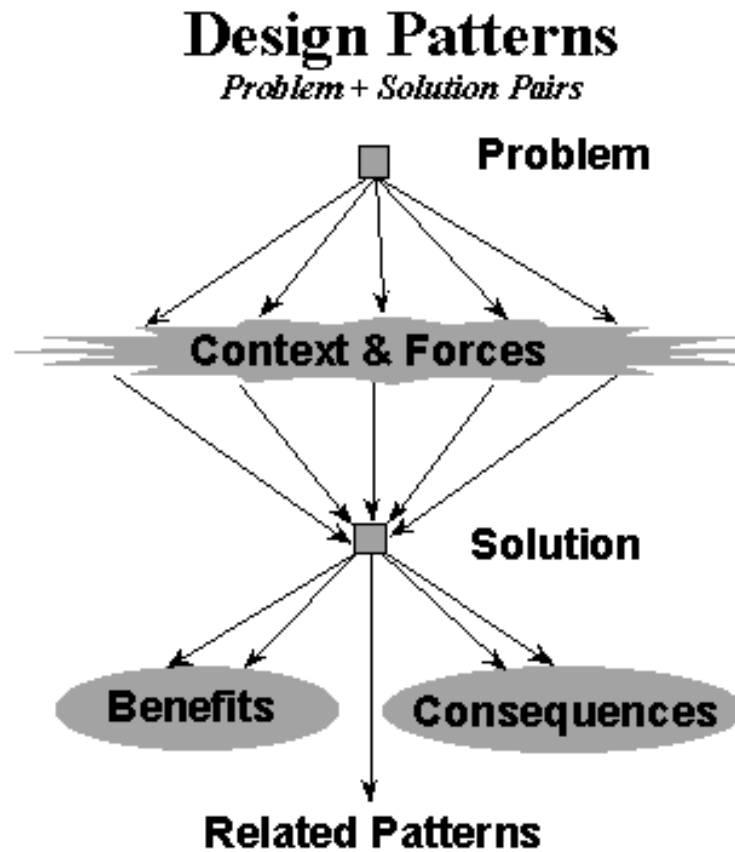
Migration in Example Application



Why Study Anti-Patterns?

- ◆ Provide a method of efficiently mapping a general situation to a specific class of solutions
- ◆ Provide real world experience in recognizing recurring problems in the software industry
- ◆ Provide a common vocabulary for identifying problems and discussing solutions

The Reference Model



Anti-Patterns

- ◆ Describes:
 - ◆ Commonly occurring solution to a problem
 - ◆ Solution often leads to negative consequences
- ◆ Results from ignorance, lack of experience, applying good patterns to wrong context, etc.
- ◆ Purpose of cataloguing:
 - ◆ Recognize
 - ◆ Remedy, often by refactoring

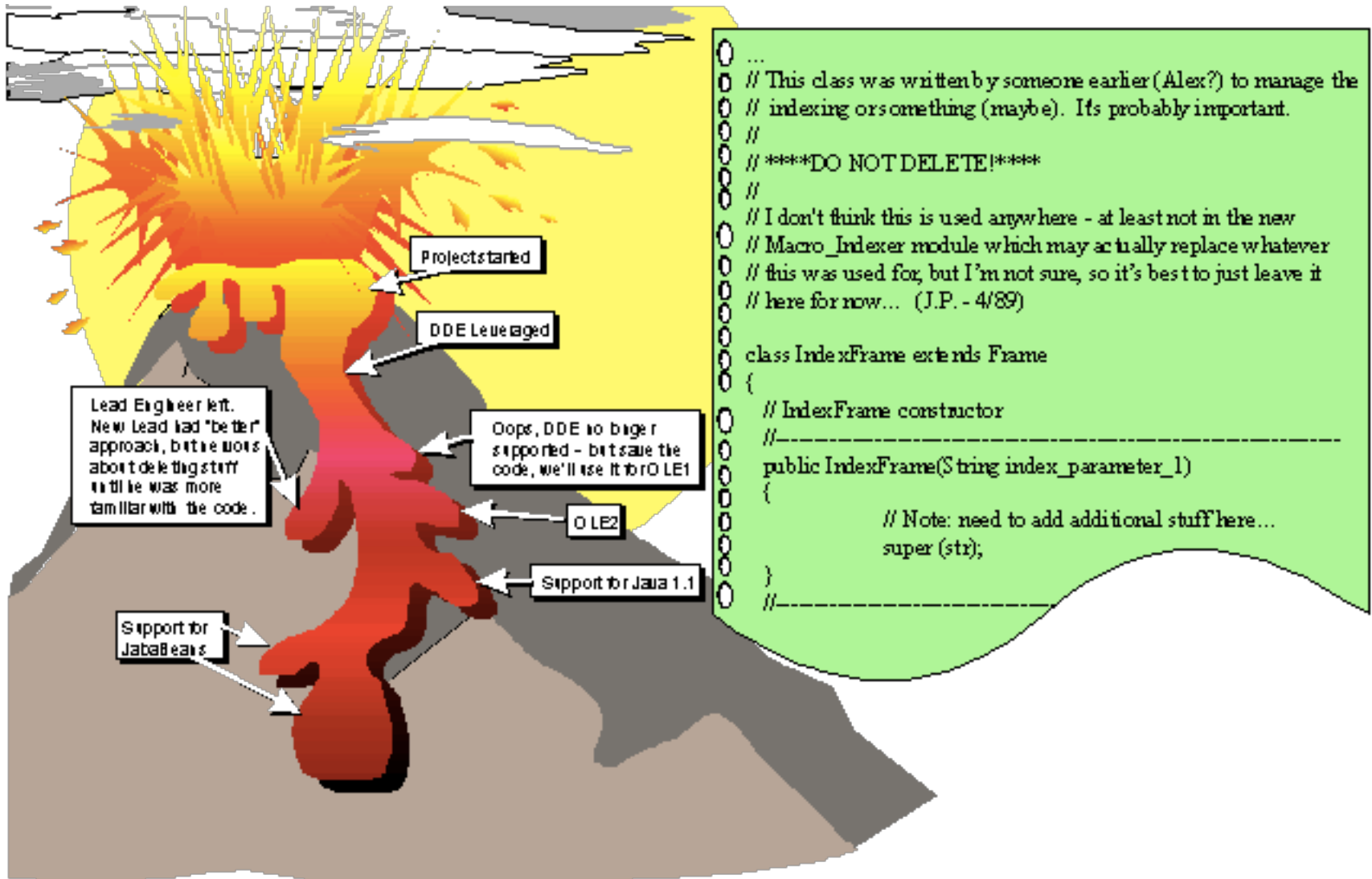
Describing an Anti-Pattern

- ◆ General Form
- ◆ Symptoms to recognize general form
 - ◆ How to identify
 - ◆ Example: One big class, a lot unrelated methods
 - ◆ Example: Many methods with no arguments
- ◆ Causes that lead to the general form
 - ◆ lack of design experience
- ◆ Refactored solution:
 - ◆ How to change into a healthier solution
 - ◆ Split into smaller classes
 - ◆ Identify or categorize attributes and operations

Anti-Pattern: Lava Flow

- ◆ Also Known As: Dead Code
- ◆ Scale: Application
- ◆ Refactored Solution Name: Architectural Configuration Management
- ◆ Refactored Solution Type: Process
- ◆ Root Causes: Avarice, Greed, Sloth
- ◆ Unbalanced Forces: Management of Functionality, Performance, Complexity
- ◆ Anecdotal Evidence: “Oh that! Well Ray and Emil (they’re no longer with the company) wrote that routine back when Jim (who left last month) was trying a workaround for Irene’s input processing code (she’s in another department now, too). I don’t think it’s used anywhere now, but I’m not really sure. Irene didn’t really document it very clearly, so we figured we would just leave well enough alone for now. After all, the bloomin’ thing works doesn’t it?!”

Anti-Pattern: Lava Flow



Symptoms of Lava Flow

- ◆ Frequent unjustifiable variables and code fragments
- ◆ Undocumented complex code segments
 - ◆ important-looking functions, classes,
 - ◆ These segments don't clearly relate to the system architecture.
- ◆ Very loose, “evolving” system architecture
- ◆ Whole blocks of commented-out code with no explanation or documentation
- ◆ Lots of “in flux” or “to be replaced” code areas



Symptoms of Lava Flow

- ◆ Unused (dead) code, just left in.
- ◆ Unused, inexplicable, or obsolete interfaces
- ◆ If existing Lava Flow code is not removed, it can continue to proliferate as code is reused in other areas.
- ◆ If the process that leads to Lava Flow is not checked, there can be exponential growth as succeeding developers, too rushed or intimidated to analyze the original flows, continue to produce new, secondary flows as they try to work around the original ones, this compounds the problem.
- ◆ As the flows compound and harden, it rapidly becomes impossible to document the code or understand its architecture enough to make improvements.

Cause

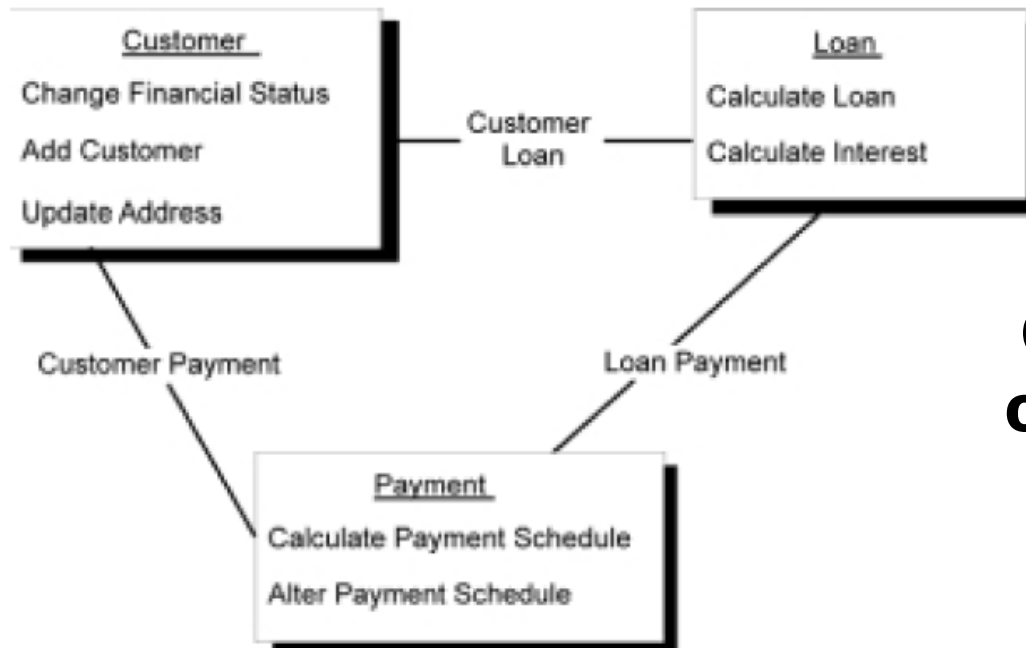
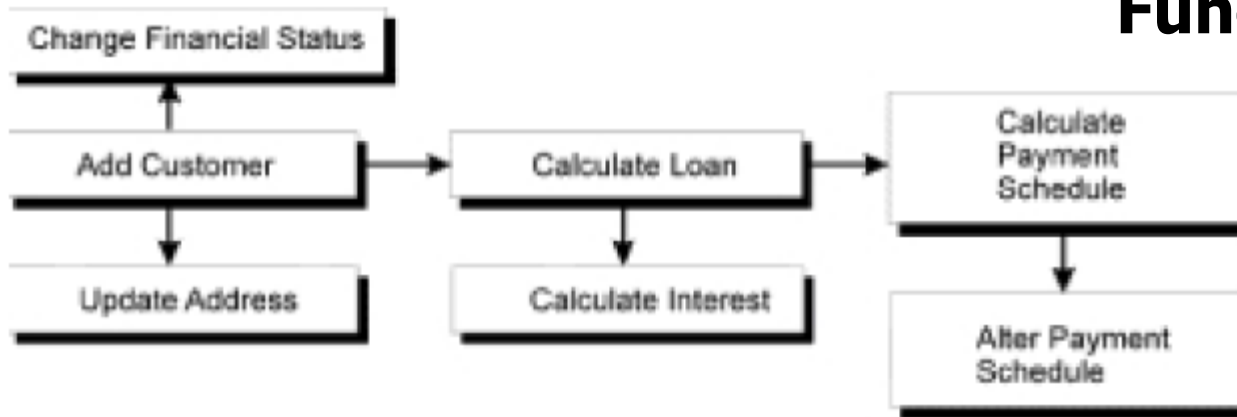
- ◆ R&D code placed into production
- ◆ Uncontrolled distribution of unfinished code. Implementation of several trial approaches toward implementing some functionality. Often single-developer (lone wolf) written code.
- ◆ Lack of architecture
- ◆ Repetitive development process
 - ◆ Goals not clear
 - ◆ Design decisions not hidden
 - ◆ Rework, backtrack, and develop prototypes
 - ◆ Hasty changes, no refactoring
- ◆ Too costly to analyze the existing code base

Functional Decomposition

- ◆ Also Known As: No Object-Oriented AntiPattern
“No OO” [Akroyd 96]
- ◆ Most Frequent Scale: Application
- ◆ Refactored Solution Name: Object-Oriented Reengineering
- ◆ Refactored Solution Type: Process
- ◆ Root Causes: Avarice, Greed, Sloth
- ◆ Unbalanced Forces: Management of Complexity, Change
- ◆ Anecdotal Evidence: “This is our ‘main’ routine, here in the class called LISTENER.”

Example

Functional



Object-oriented

Boat Anchor

- ◆ Piece of software or hardware that serves no useful purpose on the current project
- ◆ Often a costly acquisition, which makes the purchase even more ironic
- ◆ At acquisition pitch to “decision makers”
- ◆ No technical evaluation of the product
- ◆ Significant effort to make it work
- ◆ After efforts found to be useless



Boat Anchor



US \$2.91 Billion Deal



MOTOROLA

DigiFloor.com

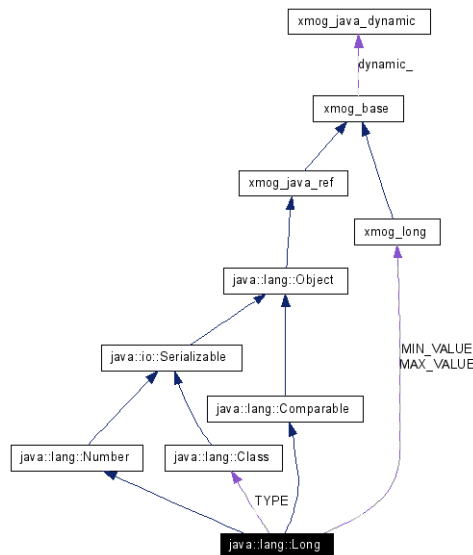
Golden Hammer

- ◆ I have a hammer and everything is a nail



Yo-yo Problem

- ◆ A programmer has to read and understand a program whose inheritance graph is so long and complicated that the programmer has to keep flipping between many different class definitions in order to follow the control flow of the program



For More Anti-Patterns

- ◆ <http://en.wikipedia.org/wiki/Anti-pattern>