

COMP 3111

SOFTWARE ENGINEERING

IMPLEMENTATION

LEARNING OBJECTIVES

1. Know the **importance** and the **major activities** of implementation.
2. Understand how to **write solid code**.
3. Learn how to **protect your code**.
4. Know how to **debug your code**.
5. Know how to **improve your code**.



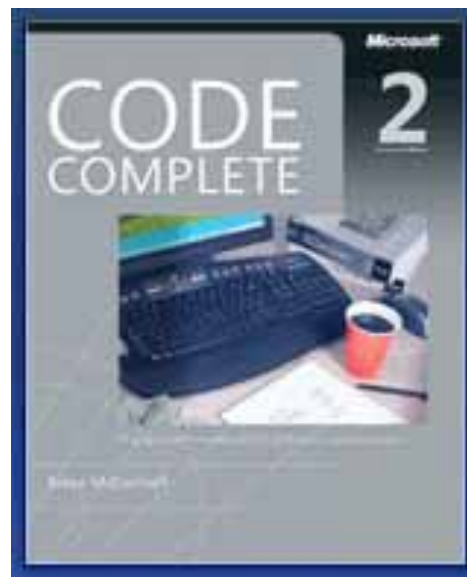
IMPLEMENTATION OUTLINE

Implementation Overview

- Life Cycle Role
- The Purpose of Implementation
- Implementation Activities

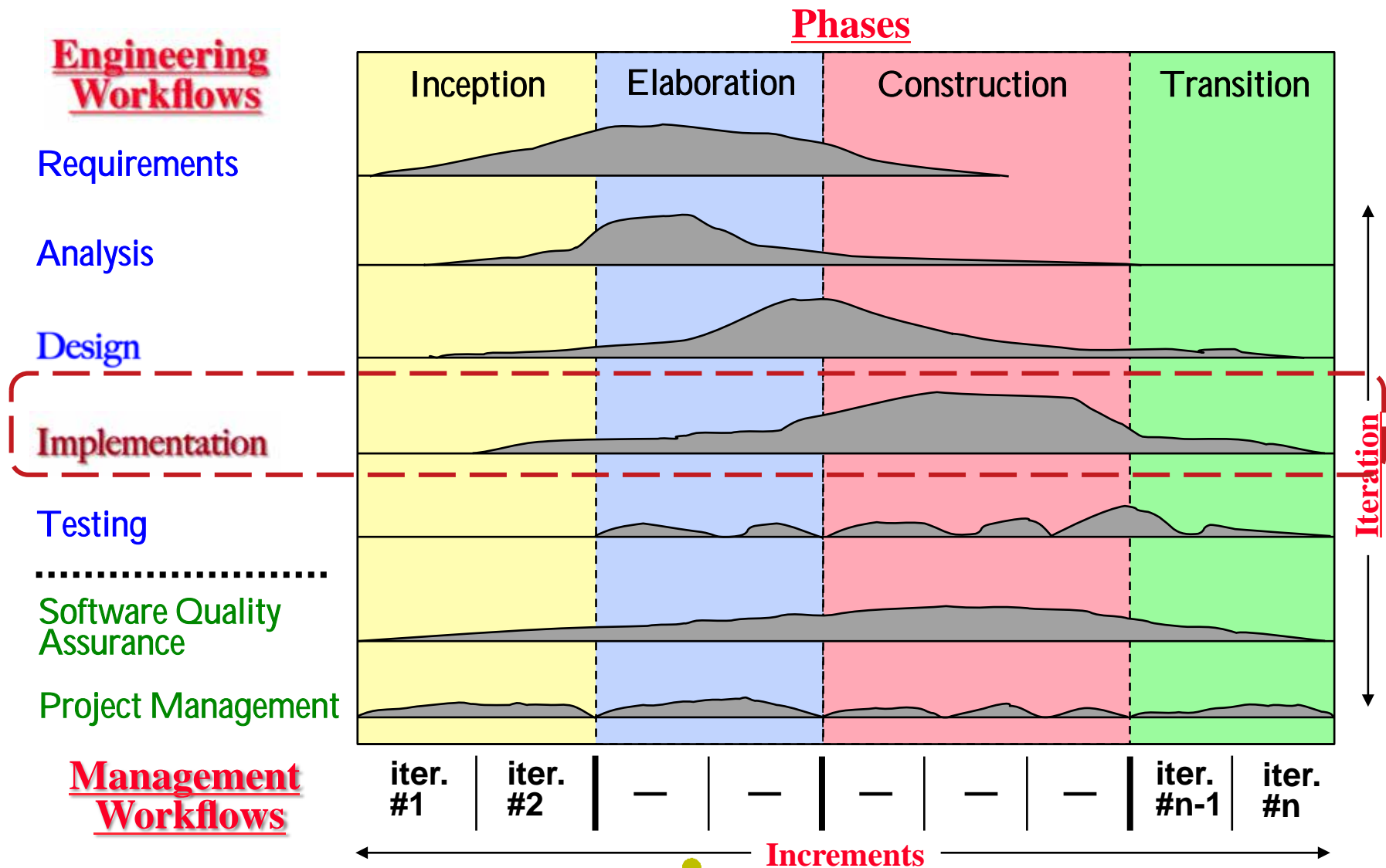
Writing Solid Code

- Reasoning About Code
- Defensive Programming
- Code Review
- Debugging
- Refactoring



Slides adapted from
Mike Ernst
U of Washington

IMPLEMENTATION LIFE CYCLE ROLE



THE PURPOSE OF IMPLEMENTATION

Implementation transforms a design into **executable code**.

- The **implementation workflow** implements the system in terms of **modules** and **subsystems**.
 - A **module** is a **physical, replaceable** part of a system that **packages implementation** and conforms to and **provides a set of interfaces**.
 - A **subsystem** **organizes modules** into more manageable pieces.
- Examples of modules:
 - source code
 - binaries
 - scripts
 - executables



IMPLEMENTATION ACTIVITIES

- **Assign classes** to modules

☞ This is **programming language dependent**.

- **Generate source code** for each class

☞ Need to **choose** and **code** suitable **algorithms** to implement methods.

- **Integrate modules and subsystems** by compiling them and linking them together into *executable modules*.

☞ Need an **integration plan** (will be discussed in the Testing workflow).

- **Distribute the executable modules** onto processing nodes.

REASONING ABOUT CODE

You: Need to write a `max()` method for an `IntList` class.

Boss: Prove that it works.

You: Run tests on some sample input.

What is the problem with this approach?



REASONING ABOUT CODE

- To reason about code we need to determine **what facts are true during execution**.
 - `x > 0`
 - for all nodes `n`: `n.next.previous == n`
 - array `a` is sorted
 - `x + y == z`
 - if `x != null`, then `x.a > x.b`

Applications

- **Ensure** code is correct (via reasoning or testing).
- **Understand** why code is incorrect.

ASSERTIONS

{true}

<- What is true here?

x = 17;

{x = 17}

<- What is true here?

y = 42;

{x = 17 \wedge y = 42}

<- What is true here?

z = x + y;

{x = 17 \wedge y = 42 \wedge z = 59} <- What is true here?

- An **assertion** is a logical formula inserted at some point in a program.
 - **precondition** - an **assertion** inserted *prior to execution*
 - **postcondition** - an **assertion** inserted *after execution*

FORWARD REASONING

- We know what is true before running the code (i.e., we know the precondition).

✎ What is true after running the code (i.e., given a precondition, what is the postcondition)?

Applications

If a class (representation) invariant holds before running the code, does it still hold after running the code?

Example

```
// precondition: x is even
x = x + 3;
y = 2 * x;
x = 5;
// postcondition: ??
```

BACKWARD REASONING

- We know what is true after running the code (i.e., we know the postcondition).

✎ What must be true before running the code in order to ensure that (i.e., give a postcondition, what is the precondition)?

Applications

- (Re-)establish a class invariant at operation exit: what is required?
- **Reproduce a bug:** What must the input have been?

Example

```
// precondition: ??  
x = x + 3;  
y = 2 * x;  
x = 5;  
// postcondition: y > x
```

**How did you
(informally)
compute this?**

FORWARD VERSUS BACKWARD REASONING

- **Forward reasoning is more intuitive for most people**
 - Helps you understand what **will happen** (simulates the code).
 - But, **introduces facts that may be irrelevant to the goal!**
 - The set of current facts may get large!
 - Takes longer to realize that the **task may be hopeless.**
- **Backward Reasoning is usually more helpful.**
 - Helps you understand what **should happen**.
 - Given **a specific goal**, indicates **how to achieve it**.
 - Given **an error**, **gives a test case** that exposes it.

REASONING ABOUT CODE STATEMENTS

Technique: Convert assertions about programs into logic rules.

General plan

- Eliminate code one statement at a time.
 - There is a (forward and backward) logic rule for each statement in the programming language.
 - **Loops have no rule:** you have to *guess a loop invariant*.
- Rely on knowledge of logic and types to reason about the code.
 - **Hoare logic**

Notation: $P \{ \text{code} \} Q$

means “if P is true and you execute **code**, then Q is true afterward”

BACKWARD REASONING

- Technique for backward reasoning:
 - Compute the **weakest precondition** (**wp**).
 - There is a **weakest precondition rule** for each statement in the programming language.
 - The weakest precondition **yields the strongest specification** for the computation (analogous to function specification).

BACKWARD REASONING: ASSIGNMENT

```
// precondition: ??  
x = e;  
// postcondition: Q
```

👉 **Precondition** = Q with all (free) occurrences of **x** replaced by **e**.

Example

```
// precondition: ??  
x = x + 1;  
// postcondition: x > 0
```

Precondition = $(x+1) > 0 \Rightarrow x > -1$

- We write this as wp for “weakest precondition”
 $wp(\text{“x=e;”}, Q) = Q$ with **x** replaced by **e**

BACKWARD REASONING: OPERATION CALLS

```
// precondition: ??  
x = foo();  
// postcondition: Q
```

👉 If the operation has no side effects:
just like ordinary assignment.

👉 If it has side effects:
an assignment to every variable it modifies.

- Use the operation specification to determine the new value.

BACKWARD REASONING: COMPOSITION (STATEMENT SEQUENCES, BLOCKS)

```
// precondition: ??  
S1;           // some statement  
S2;           // another statement  
// postcondition: Q
```

 **Work from back to front.**

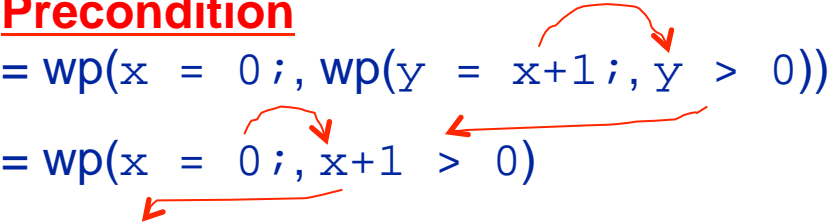
$\text{precondition} = \text{wp}(\text{"S1; S2;"}, Q) = \text{wp}(\text{"S1;"}, \text{wp}(\text{"S2;"}, Q))$

Example

```
// precondition: ??  
x = 0;  
y = x+1;  
// postcondition: y > 0
```

Precondition

$= \text{wp}(x = 0; , \text{wp}(y = x+1; , y > 0))$
 $= \text{wp}(x = 0; , x+1 > 0)$
 $= \text{wp}(1 > 0)$
 $= \text{true}$



BACKWARD REASONING: IF-THEN-ELSE

```
// precondition: ??  
if (B) S1 else S2  
// postcondition: Q
```

✋ Essentially case analysis.

$$\begin{aligned} \text{wp}(\text{"if (B) S1 else S2"}, Q) = \\ B \Rightarrow \text{wp}(\text{"S1"}, Q) \wedge !B \Rightarrow \text{wp}(\text{"S2"}, Q) \end{aligned}$$

BACKWARD REASONING: IF-THEN-ELSE

```
// precondition: ??  
if (x == 0)  
    {x = x + 1;}  
else  
    {x = x/x;}  
// postcondition: x ≥ 0
```

Precondition

$$\begin{aligned} &= \text{wp}(\text{"if } (x==0) \{x=x+1;\} \text{ else } \{x=x/x;\}", x \geq 0) \\ &= (x==0 \wedge (\text{wp}(\text{"x=x+1"}, x \geq 0))) \quad \vee \quad (x \neq 0 \wedge \text{wp}(\text{"x=x/x"}, x \geq 0)) \\ &= (x==0 \wedge x+1 \geq 0) \quad \vee \quad (x \neq 0 \wedge x/x \geq 0) \\ &= (x==0) \quad \vee \quad \text{true} \\ &= \text{true} \end{aligned}$$

BACKWARD REASONING: Loops

- **A loop represents an unknown number of paths**
 - Case analysis is problematic.
 - Recursion presents the same issue.
- **Cannot enumerate all paths**
 - This is what makes testing and reasoning hard!

BACKWARD REASONING: Loops

```
// precondition:  $x \geq 0 \wedge y = 0$   
while (x != y)  
{  
    y = y + 1;  
}  
// postcondition:  $x = y$ 
```

Does this loop
terminate?

1. The precondition guarantees that $x \geq y$.
2. Every time through the loop
 - $x \geq y$ still holds.
 - y is incremented by 1.
 - x is unchanged.
 - Therefore, y is closer to x .
3. Since there are only a finite number of integers between x and y , y will eventually equal x .
4. Execution exits the loop as soon as $x = y$.

REASONING ABOUT LOOPS BY INDUCTION

- **We just made an inductive argument.**
- **What are we inducting over?**
 - The number of iterations.
- **Computation induction**
 - Show that conjecture **holds for zero iterations**.
 - Show that it **holds after $n+1$ iterations**.
 - Assuming that it **holds after n iterations**.
- **Two things to prove.**
 - Some **property is preserved** (known as “**partial correctness**”).
 - The loop invariant is preserved by each iteration.
 - The **loop completes** (known as “**termination**”).
 - The “decrementing function” is reduced by each iteration.

PROPERTIES OF LOOP INVARIANT (LI)

```
// Precondition: P
while (B) S;
// Postcondition: Q
```

Equivalently: $P \{ \text{while (B) S;} \} Q$

- Find a loop invariant, **LI**, such that
 - 1) $P \Rightarrow LI$ → LI holds initially
 - 2) $LI \wedge B \{S\} LI$ → LI holds if the loop executes once
 - 3) $(LI \wedge !B) \Rightarrow Q$ → $(LI \wedge !B)$ establishes the postcondition
- It is sufficient to know that if the loop terminates, **Q** will hold.
- Finding the invariant is **the key** to reasoning about loops.
- Inductive assertions is a complete method of proof.
 - If a loop satisfies pre/post conditions, there exists an invariant sufficient to prove it.

LOOP INVARIANT FOR THE EXAMPLE

```
// precondition:  $x \geq 0 \wedge y = 0$ 
while (x != y)
{
    y = y + 1;
}
// postcondition:  $x = y$ 
```

- What is a suitable loop invariant?
- What has to be true to make the loop work?

$LI = x \geq y$

- | | |
|---------------------------------------------------|------------------------------------------|
| 1) $x \geq 0 \wedge y = 0 \Rightarrow LI$ | LI holds immediately before the loop |
| 2) $LI \wedge x \neq y \{y = y+1;\} LI$ | LI holds at end of the loop body |
| 3) $(LI \wedge \neg(x \neq y)) \Rightarrow x = y$ | $\{LI \wedge \neg B\} \rightarrow \{Q\}$ |

EXAMPLE: REASONING ABOUT CODE

Starting with the postcondition and statements, fill in the intermediate assertions and weakest precondition for each of the following code fragments.

1. Assignment Statements

```
w=2*w;  
z=-w;  
y=v+1;  
x=min(y, z);  
{x<0}
```

2. If-Then-Else Statements

```
if (x!=0)  
    {z=x;}  
else  
    {z=x+1;}  
{z>0}
```

DEFENDING AGAINST INVALID INPUTS

- **Check the values of all data from external sources.**
- **Check the values of all routine input parameters.**
- **Decide how to handle bad inputs.**

 **Assertions are your friend!**

- **Example:** Visual C#

```
Debug.Assert(denominator != 0,  
    "Denominator is unexpectedly equal to 0.");
```

 **Validators are a type of assertion!**

ASSERTION USES

- Assertions can be used to check whether:
 - An **input parameter's value** falls within its expected range.
 - An **output parameter's value** falls within its expected range.
 - A **file or stream is open** (or closed) when a routine begins executing (or when it ends executing).
 - The **value of an input-only variable** is not changed by a routine.
 - A **pointer is non-null**.
 - An **array** or other container passed into a routine can **contain at least X number of data elements**.
 - A **table has been initialized** to contain real values.
 - A **container is empty** (or full) when a routine begins executing (or when it finishes).

ASSERTION GUIDELINES

- **Use assertions for conditions that should never occur.**
 - Use **error-handling code** for conditions you expect to happen.
- **Avoid putting executable code into assertions.**
 - The compiler may eliminate assertions.
- **Use assertions to document and verify preconditions and postconditions.**
- **Error handling techniques**
 - Return a neutral (harmless) value.
 - Log a message to a file.
 - Return an error code.
 - Display an error message.

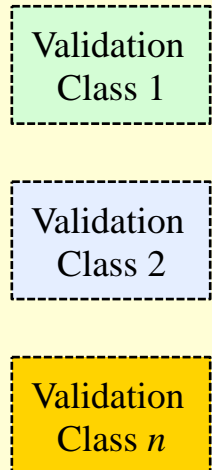
BARRICADE YOUR PROGRAM

Possible inputs

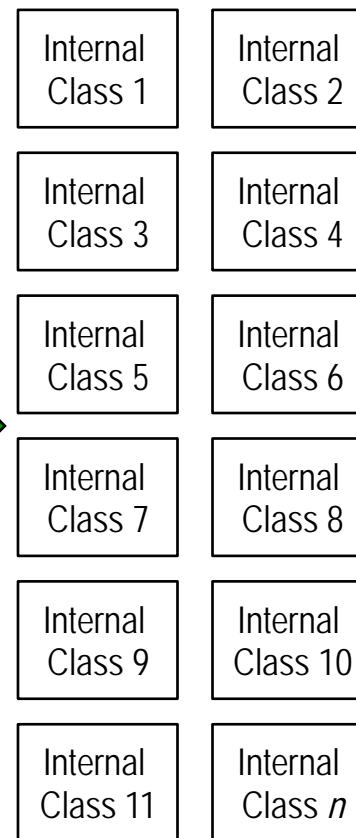


Data here is assumed to be dirty and untrusted.

Barricade



These classes are responsible for cleaning the data. They make up the barricade.



These classes can assume data is clean and trusted.

Convert input data to the proper type at input time.

CODE REVIEW: WHAT IS IT?

- Off-line version of _____.
- Review code written by **other developers**.

 **A common practice in industry.**

- **Not finding faults** in others, but _____.
- **Voluntary review** is important.

 **Otherwise you will be forced to review other developers' broken code.**

CODE REVIEW: MOTIVATION

- Can catch most bugs, design flaws early.
- More than one person has seen every piece of code.
- Forces code authors to articulate their decisions and to participate in the discovery of flaws.
- Allows junior personnel to get early hands-on experience without hurting code quality.
- Accountability: both author and reviewers.
- Explicit non-purpose: assessment of performance.

CODE REVIEW: PROCESS

- **What is reviewed?**
 - A specification or design document
 - A coherent module (sometimes called an “inspection”)
 - A single checkin or code commit (incremental review)
- **Who participates in the review?**
 - One other developer
 - A group of developers
- **Where does the review take place?**
 - In-person meeting
 - Best to prepare beforehand: artifact is distributed in advance.
 - Preparation usually identifies more defects than the actual meeting.
 - Email/chat

CODE REVIEW: GOALS AND TECHNIQUES

- **Specific focus?**

- Sometimes, a specific list of defects or code characteristics
 - Error-prone code
 - Previously discovered problem types
 - Security
 - Checklist (coding standards)
 - Automated tools (type checkers, lint) can be better

- **Review technique**

- Does developer present the artifact to a group?
- Only identify defects or also brainstorm fixes?
- Sometimes, use a specific methodology
 - “Walkthrough” = playing computer, trace values of sample data

WAYS TO GET YOUR CODE RIGHT

- **Testing**
 - For **uncovering problems** and **increasing confidence**.
- **Defensive programming**
 - Programming with **testing** and **debugging** in mind.
- **Debugging**
 - Finding out **why** a program is **not functioning** as intended.
- **Testing ≠ Debugging**
 - Testing**: Reveals the existence of a problem.
 - Debugging**: Pinpoints the location **plus** the cause of a problem.

DEBUGGING: DEFENDING AGAINST BUGS

1. **Make errors impossible by design**
2. **Do not introduce defects**
3. **Make errors immediately visible**
4. **Last resort is debugging**



DEBUGGING: DEFENSE 1— MAKE ERRORS IMPOSSIBLE BY DESIGN

- **In the language**
 - E.g., Java makes memory overwrite bugs impossible.
- **In the protocols/libraries/modules**
 - TCP/IP will guarantee that data is not reordered.
 - Java BigInteger will guarantee that there will be no overflow.
- **In self-imposed conventions**
 - Hierarchical locking makes deadlock bugs impossible.
 - Banning the use of recursion will make infinite recursion/insufficient stack bugs go away.
 - Immutable data structures will guarantee behavioural equality.

 **Caution:** You must maintain the discipline!

DEBUGGING: DEFENSE 2— DO NOT INTRODUCE DEFECTS

- **Get things right the first time**
 - **Do not code before you think! Think before you code!**
 - If you are making lots of easy-to-find bugs, you are also making hard-to-find bugs. → *Do not use the compiler as a crutch.*
- **Especially true when debugging is going to be hard**
 - Concurrency
 - Difficult test and instrument environments
 - Program must meet timing deadlines
- **Simplicity is the key**
 - **Modularity**
 - Divide program into chunks that are easy to understand
 - Use abstract data types with well-defined interfaces
 - Use defensive programming
 - **Specification**
 - Write specification for all modules so that an explicit, well-defined interface exists for each module that clients can rely on.

DEBUGGING: DEFENSE 3— MAKE ERRORS IMMEDIATELY VISIBILITY

- If you cannot prevent bugs, you can [try to localize them](#) to a small part of the program.

Assertions: Catch bugs early, before failure has a chance to contaminate (and be obscured by) further computation.

Unit testing: When you test a module in isolation, you can be confident that any bug you find is in that module (unless it's in the test driver).

Regression testing: Run tests as often as possible when changing code. If there is a failure, chances are there is a mistake in the code you just changed.

- When localized to a single operation or small module, bugs can usually be found simply by studying the program text.

DEBUGGING: DEFENSE 3— BENEFITS OF IMMEDIATE VISIBILITY

- **The key difficulty of debugging is in finding the code fragment responsible for an observed problem.**
 - An operation may return an erroneous result, but be itself error free if there is prior corruption of data.
- **The earlier a problem is observed, the easier it is to fix.**
 - Frequently check assertions.
- **The general approach: fail-fast**
 - Check invariants, do not just assume them.
 - Do not try to recover from bugs — this just obscures them.

DEBUGGING: DO NOT HIDE BUGS

```
// k is guaranteed to be present in a
int i = 0;
while (true)
{
    if (a[i] == k)
        break;
    i++;
}
```

This code fragment searches an array *a* for a value *k*.

The value is guaranteed to be in the array.

If that guarantee is broken (by a bug), the code throws an exception and dies.

Temptation: Make the code more “robust” by not failing.

DEBUGGING: DO NOT HIDE BUGS

```
// k is guaranteed to be present in a
int i = 0;
while (i < a.length)
{
    if (a[i] == k)
        break;
    i++;
}
```

Now at least the loop will always terminate, **BUT ...**

DEBUGGING: DO NOT HIDE BUGS

```
// k is guaranteed to be present in a
int i = 0;
while (i < a.length)
{
    if (a[i] == k)
        break;
    i++;
}
assert (i < a.length): "Key not found.";
```

Assertions let us document and check invariants.

Abort the program as soon as a problem is detected.



DEBUGGING: INSERTING CHECKS

- **Insert lots of checks with an intelligent checking strategy.**
 - Precondition checks
 - Consistency checks
 - Bug-specific checks
- **Goal: Stop the program as close to a bug as possible.**
 - Use debugger to see where you are, explore the program a bit.

DEBUGGING: CHECKS IN PRODUCTION CODE

- **Should assertions and checks be included in production code?**

Yes Stop program if check fails—don't want to take a chance that the program will do something wrong.

No May need the program to keep going, maybe bug does not have such bad consequences.

 **The correct answer depends on the context!**

Ariane 5

- The program halted because of overflow in an unused value.
- An exception was thrown, but not handled until the top level.
- The rocket crashed ...

DEBUGGING: REGRESSION TESTING

- **Whenever you find and fix a bug**
 - Add a test for it
 - Re-run all your tests
- **Why is this a good idea?**
- **Run regression tests as frequently as you can afford to.**
 - Automate the process.
 - Make concise test sets with few unnecessary tests.

DEBUGGING: LOGGING EVENTS

- **Often you would like to have some indication of the past when a check fails.**
- **Design a logging infrastructure**
 - Dump events to a file (strings).
 - Events have consistent format to enable efficient searches.
 - Sometimes (usually for timing reasons) we must keep a lot in memory, not on disk.
 - Use circular logs to avoid resource exhaustion.
- **Important in debugging in customer environments**
 - May not have access to the customer use
 - Only the log is available
 - Use information in the log to help reproduce the bug.

DEBUGGING: THE LAST RESORT

- **Bugs happen.**

Industry average: 10 bugs per 1000 lines of code.

- **Bugs that are not immediately localized happen.**
 - Found during integration testing.
 - Reported by user.

Step 1: Clarify the symptom.

Step 2: Find and understand cause, create a test.

Step 3: Fix.

Step 4: Rerun all tests.

DEBUGGING: KINDS OF BUGS

Quick, easy bugs (a few minutes)

Medium bugs (a few hours)

Hard bugs (a small number of days)

Really bad bugs (many days to never)

- Look for bugs in this order!
- There are different debugging strategies for each type.



DEBUGGING STRAGTEGY: EASY BUGS

- **Hope for an easy bug; take a first quick look**
 - Look at backtrace in the debugger.
 - Look at code where you think there might be a problem, maybe use a debugger or a few print statements.
 - Try to get lucky.

 **Make the first shot quick! Don't get sucked in!**

DEBUGGING STRATEGY: MEDIUM BUGS

- **Look for medium bug with next shot**
 - Use print statements.
 - Design an organized print strategy.
 - Use legible, easy to read error messages.

 **Make the medium shot medium! Don't get sucked in!**



DEBUGGING STRAGTEGY: HARD BUGS

- **Rebuild the system from scratch.**
- **Explain the bug to a friend.**
- **Make sure it is a bug!**
 - ☞ The program may be working correctly and you don't realize it!
- **Minimize the input required to exercise the bug.**
- **Add checks to the program.**
 - Minimize the distance between the error and detection.
 - Use **binary search** to narrow down the possible locations.
- **Use event logs to record events in the execution history.**

DEBUGGING STRATEGY: REDUCE INPUT SIZE

```
boolean substr(String s, String b)
```

- Returns **false** for

s = “The wworld is ggreat! Liffe is wwonderful! I am so vvery happy all of the ttime!”

b = “very happy”

even though “very happy” is a substring of s

DEBUGGING GENERAL STRATEGY: SIMPLIFY

- **In general, find simple input that will produce the bug.**
 - Usually not the input that revealed the existence of the bug.
- **Start with data that revealed the bug.**
 - Keep paring it down (binary search can help).
 - Often leads directly to an understanding of the cause.
- **When not dealing with simple operation calls**
 - Think of “test input” as the set of steps needed to reliably trigger the bug.

DEBUGGING: LOCALIZING A BUG

- **Take advantage of modularity**
 - Start with everything and take away pieces until bug disappears.
 - Start with nothing and add pieces back in until bug appears.
- **Take advantage of modular reasoning**
 - Trace through program, viewing intermediate results.
- **Use binary search to speed things up**
 - Bug happens somewhere between first and last statement.
 - Do a binary search on that ordered set of statements.

DEBUGGING: BINARY SEARCH ON BUGGY CODE

```
public class MotionDetector {  
    private boolean first = true;  
    private Matrix previous = new matrix();  
  
    public Point apply(Matrix current) {  
        if (first) {  
            previous = current;  
        }  
        Matrix motion = new Matrix();  
        getDifference(previous, current, motion);  
        applyThreshold(motion, motion, 10);  
        labelImage(motion, motion);  
        Hist hist = getHistogram(motion);  
        int top = hist.getMostFrequent();  
        applyThreshold(motion, motion, top, top);  
        Point result = getCentroid(motion);  
        previous.copy(current);  
        return result;  
    }  
}
```


**Quickly home in on bug in $O(\log n)$
time by repeated subdivision.**

no problem yet

**Check
intermediate
result at half-
way point.**

problem exists

DEBUGGING: HEISENBUGS

- **In a sequential, deterministic program**
 **A bug should be repeatable.**
- **But the real world is not that nice ...**
 - Continuous input/environment changes
 - Timing dependencies
 - Concurrency and parallelism
- **Bugs sometimes occur randomly.**
- **Bugs are sometimes hard to reproduce.**
 - When using debugger or assertions → bug goes away.
 - Only happens when under heavy load.
 - Only happens once in a while.

DEBUGGING: HARSH ENVIRONMENTS

- **Harsh environments**
 - Bug is nondeterministic, difficult to reproduce.
 - Can't print or use debugger.
 - Can't change timing of program (or bug has to do with timing).
- **Build an event log (circular buffer).**
- **Log events during execution of program as it runs at speed.**
- **When you detect an error, stop the program and examine the log.**

DEBUGGING: WHERE IS THE BUG?

- **The bug is not where you think it is.**
 - Ask yourself where it cannot be; explain why.
- **Look for stupid mistakes first, such as:**
 - Reversed order of arguments: `Collection.copy(src, dest)`
 - (Mis)Spelling of identifiers
 - Same object versus equals: `a == b` versus `a.equals(b)`
 - Failure to reinitialize a variable
 - Deep versus shallow copy
- **Make sure that you have correct source code**
 - Recompile everything

DEBUGGING: WHEN THE GOING GETS TOUGH

- **Reconsider assumptions**
 - E.g., has the OS changed? Is there room on the hard drive?
 - Debug the code, not the comments.
- **Start documenting your system**
 - Gives a fresh angle and highlights areas of confusion.
- **Get help**
 - We all develop blind spots.
 - Explaining the problem to others often helps.
- **Walk away**
 - Trade latency for efficiency—**sleep!**
 - One good reason to start early.

DEBUGGING: FINDING BUGS AUTOMATICALLY

- **FindBugs**, Bill Pugh, University of Maryland
 - Recognizes 284 different **bug patterns**.
 - (<http://findbugs.sourceforge.net/bugDescriptions.html>)
- **ReCrash**, Sunghum Kim, MIT/HKUST
 - Makes **software failures reproducible** by preserving object state.
 - <http://groups.csail.mit.edu/pag/reCrash/>



REFACTORING

- **Refactoring** is **improving** a piece of software's **internal structure** without altering its **external behaviour**.
 - Incurs a **short term time/work cost** to reap **long-term benefits**.
- **Why needed?**
 - After several months and new versions, many code bases reach one of the following states:
 - **rewritten**: Nothing remains from the original code.
 - **abandoned**: The original code is thrown out and rewritten from scratch.
 - due to a system evolving to meet new needs and add new features.
 - If the code's structure does not also evolve, it will “**rot**”.
 - This can happen even if the code was initially reviewed and well-designed at the time of checkin, and even if checkins are reviewed.

WHY REFACTOR?

Why fix a part of your system that isn't broken?

- Each part of your system's code has **three purposes**:
 1. to **execute** its functionality.
 2. to **allow change**.
 3. to **communicate well** to developers who read it.
- If the code does not do all three of these, **it is broken**.



REFACTORING: LOW-LEVEL

Names

- Renaming (operations, variables).
- Naming (extracting) “magic” constants.

Procedures

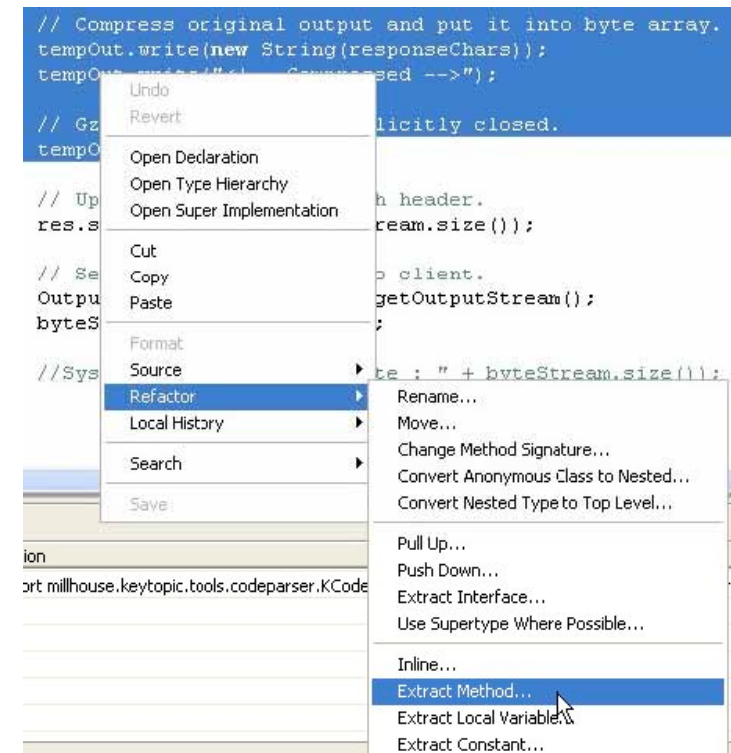
- Extracting code into a method.
- Extracting common functionality (including duplicate code) into a module/operation/etc.
- Inlining an operation/procedure.
- Changing operation signatures.

Reordering

- Splitting one operation into several to improve cohesion and readability (by reducing its size).
- Putting statements that semantically belong together near each other.

IDE SUPPORT FOR LOW-LEVEL REFACTORING

- Eclipse/Visual Studio support
 - variable / method / class renaming
 - method or constant extraction
 - extraction of redundant code snippets
 - method signature change
 - extraction of an interface from a type
 - method inlining
 - providing warnings about method invocations with inconsistent parameters
 - help with self-documenting code through auto-completion



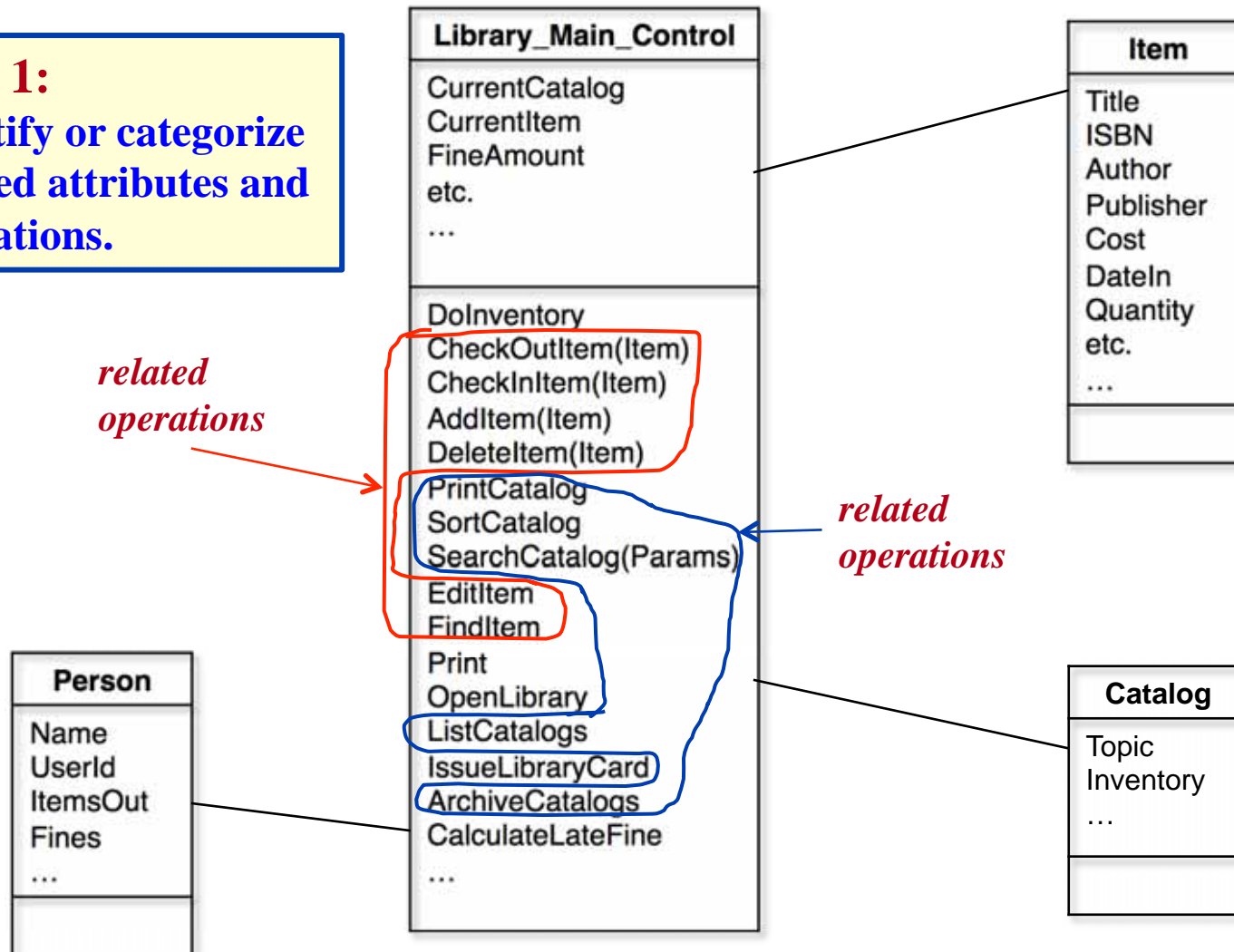
HIGHER-LEVEL REFACTORING

- Refactoring to design patterns (will discuss later).
- Exchanging obscure language idioms with safer alternatives.
- Performance optimization.
- Clarifying a statement that has evolved over time or is unclear.
- Compared to low-level refactoring, high-level refactoring is:
 - Not as well supported by tools.
 - **Much more important!**



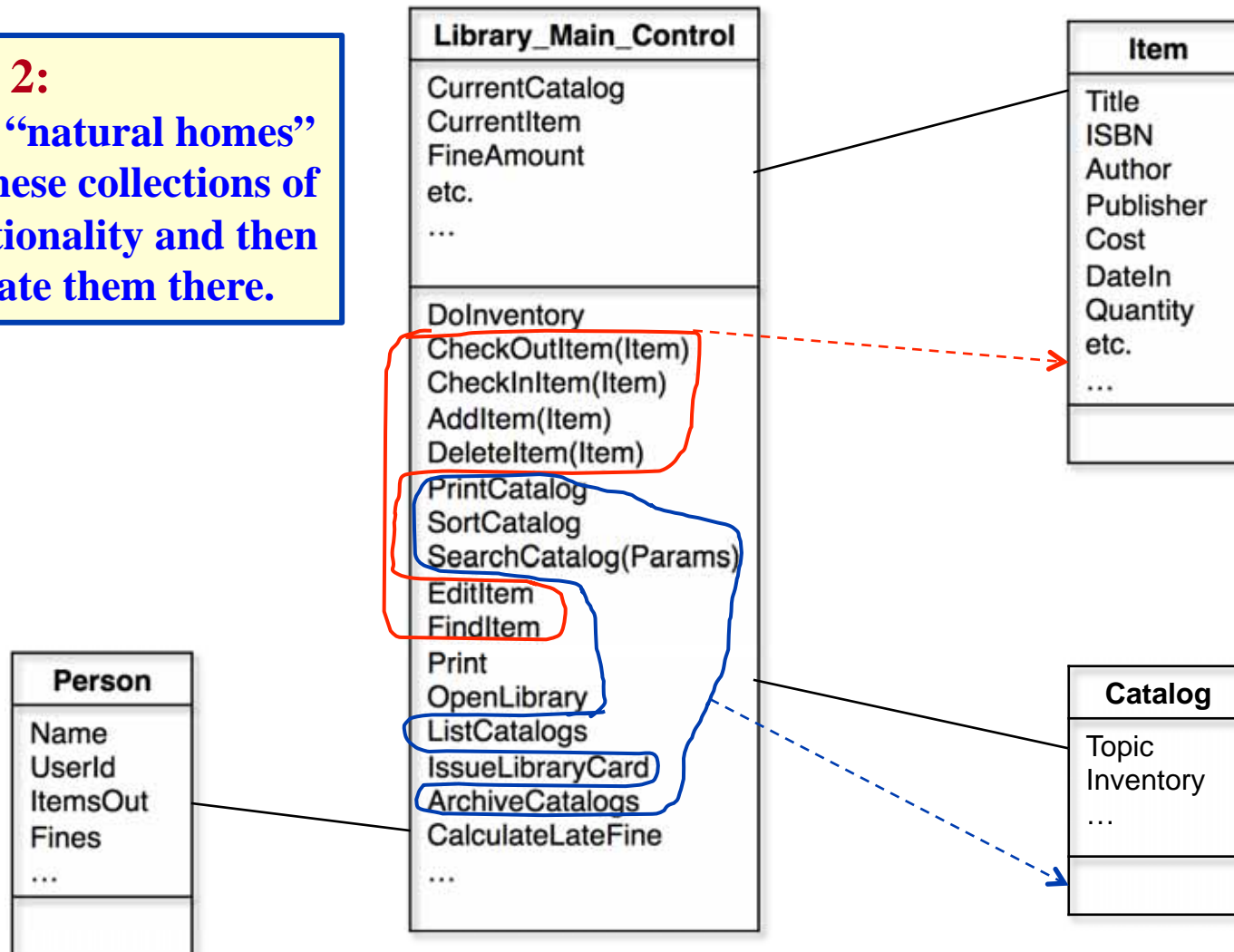
REFACTORING EXAMPLE: THE LIBRARY BLOB

Step 1:
Identify or categorize
related attributes and
operations.



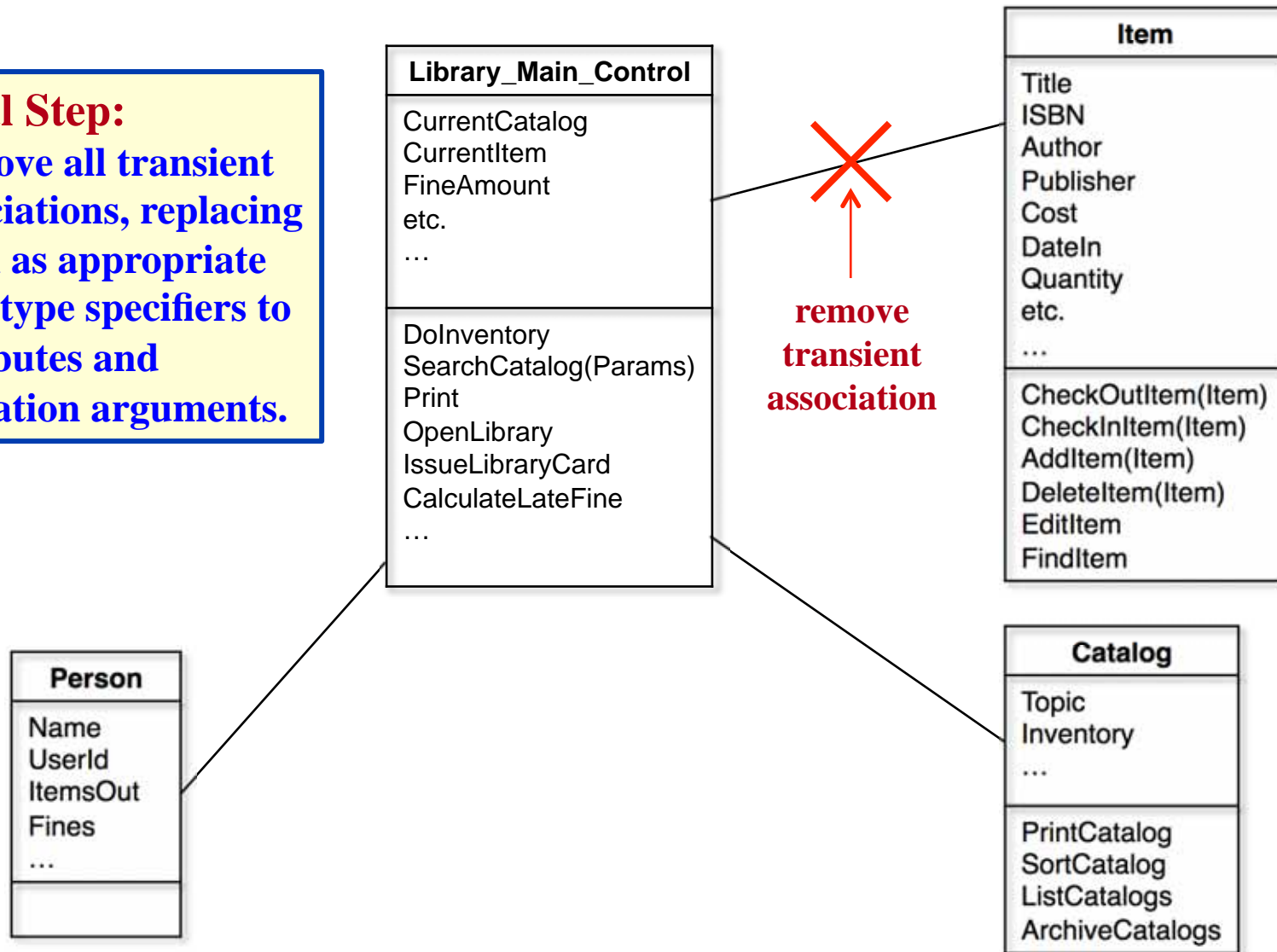
REFACTORING EXAMPLE: THE LIBRARY BLOB

Step 2:
Find “natural homes”
for these collections of
functionality and then
migrate them there.

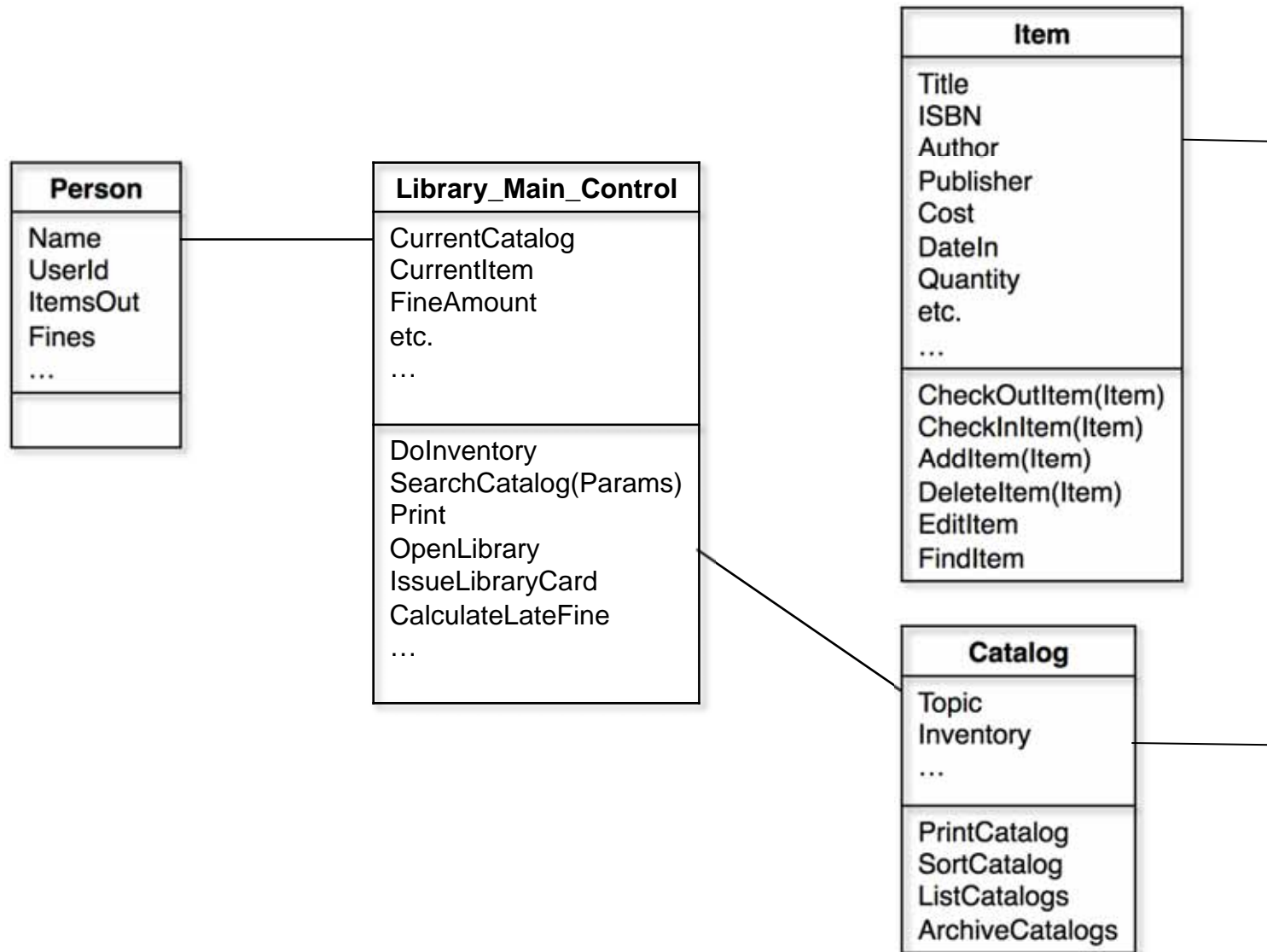


REFACTORING EXAMPLE: THE LIBRARY BLOB

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



REFACTORING EXAMPLE: THE LIBRARY BLOB



REFACTORING PLAN

Suppose you want to add new features to code that is not particularly well designed or thoroughly tested, but works so far.

- **What should you do?**
 - Assume that you have adequate time to “do things right.”
 - (Not always a valid assumption in software development!)
 - Write unit tests that verify the code’s external correctness.
 - They should pass on the current, badly designed code.
 - First, refactor the code.
 - Some unit tests may now break. Fix the bugs.
 - Add the new features.

"I DON'T HAVE TIME TO REFACTOR!"

- **Refactoring incurs an up-front cost.**
 - Many developers do not want to do it.
 - Most management do not like it because they lose time and gain “nothing” (no new features).
- **However ...**
 - Well-written code is much more conducive to rapid development (some estimates put ROI at 500% or more for well-done code).
 - Finishing refactoring increases programmer morale.
 - Developers prefer working in a “clean house.”
- **When to refactor?**
 - Best done continually (like testing) as part of the development process.
 - It is hard to do well late in a project (like testing). **WHY?**

IMPLEMENTATION: SUMMARY

- **The Implementation workflow:**
 - Implements classes (primarily operation methods) in modules.
 - Organizes modules into subsystems.
 - Integrates all modules and subsystems into the final system.
 - Assigns executable modules to processing nodes.
- **Writing solid code requires skills in:**
 - Reasoning about code
 - Defensive programming
 - Code review
 - Debugging
 - Refactoring