COMP 3111
SOFTWARE ENGINEERING

LECTURE 11
IMPLEMENTATION

LEARNING OBJECTIVES

1. Know the purpose and the major activities of implementation.

2. Know how to protect your code.

3. Know how to improve your code.

4. Know how to debug your code.

5. Know how to backup your code.

9, 10

IMPLEMENTATION OUTLINE

Implementation Overview

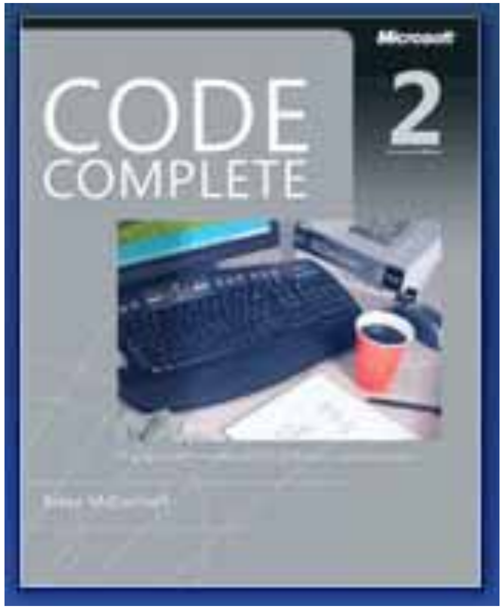
- Life Cycle Role
- The Purpose of Implementation
- Implementation Activities

Producing Solid Code

- Defensive Programming
- Code Review
- Refactoring

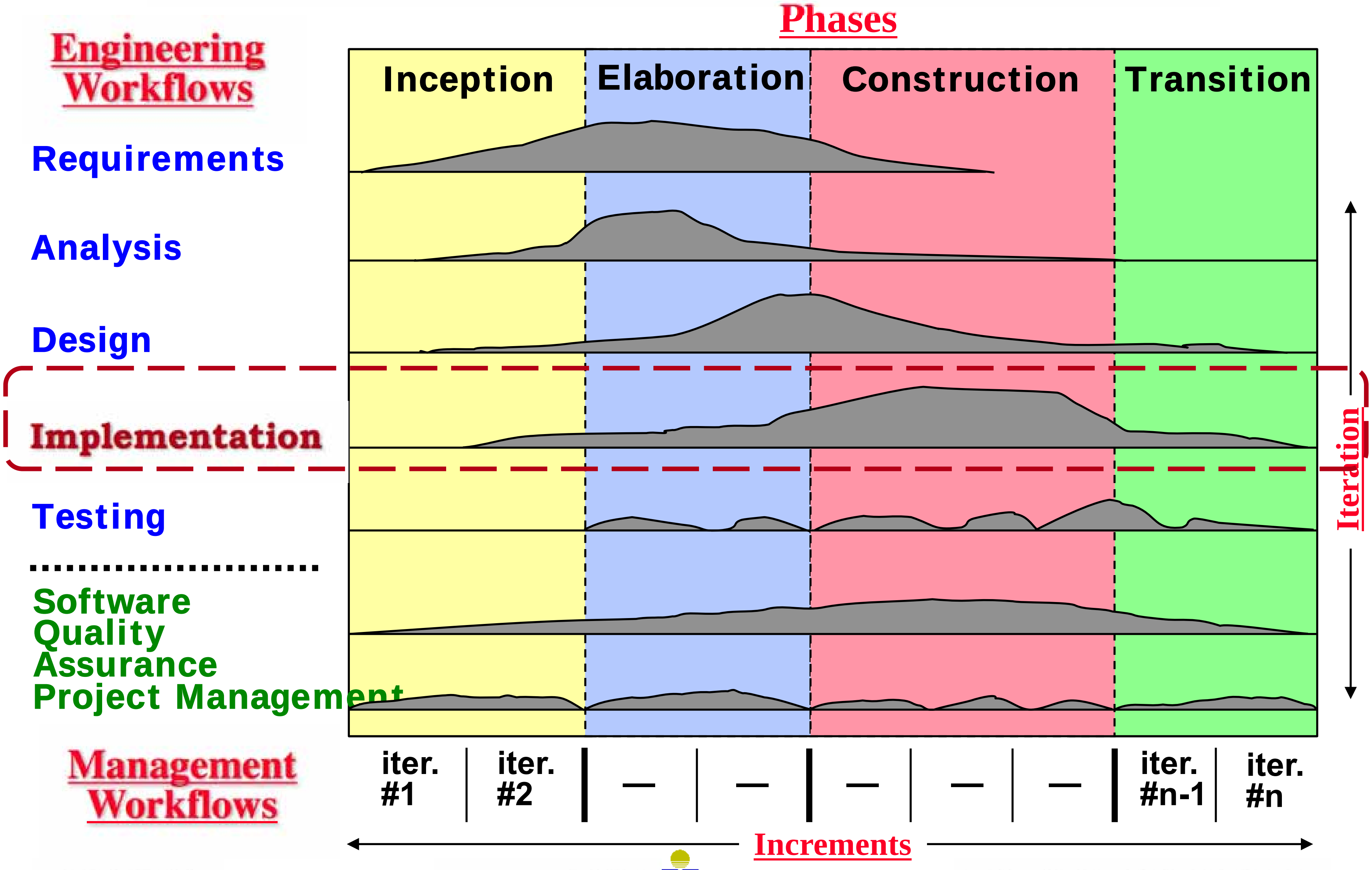
Debugging

Configuration Management



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
  - scripts
  - binaries
  - executables

IMPLEMENTATION ACTIVITIES

- **Generate source code** for each class
  - ☞ Need to choose and code suitable algorithms to implement methods.
- **Assign classes** to modules
  - ☞ This is programming language dependent.
- **Integrate modules and subsystems** by compiling them and linking them together into *executable modules*.
  - ☞ Need version control (will be discussed shortly).
  - ☞ Need an integration plan (will be discussed in the Testing workflow).
- **Distribute the executable modules** onto processing nodes.

DEFENSIVE PROGRAMMING

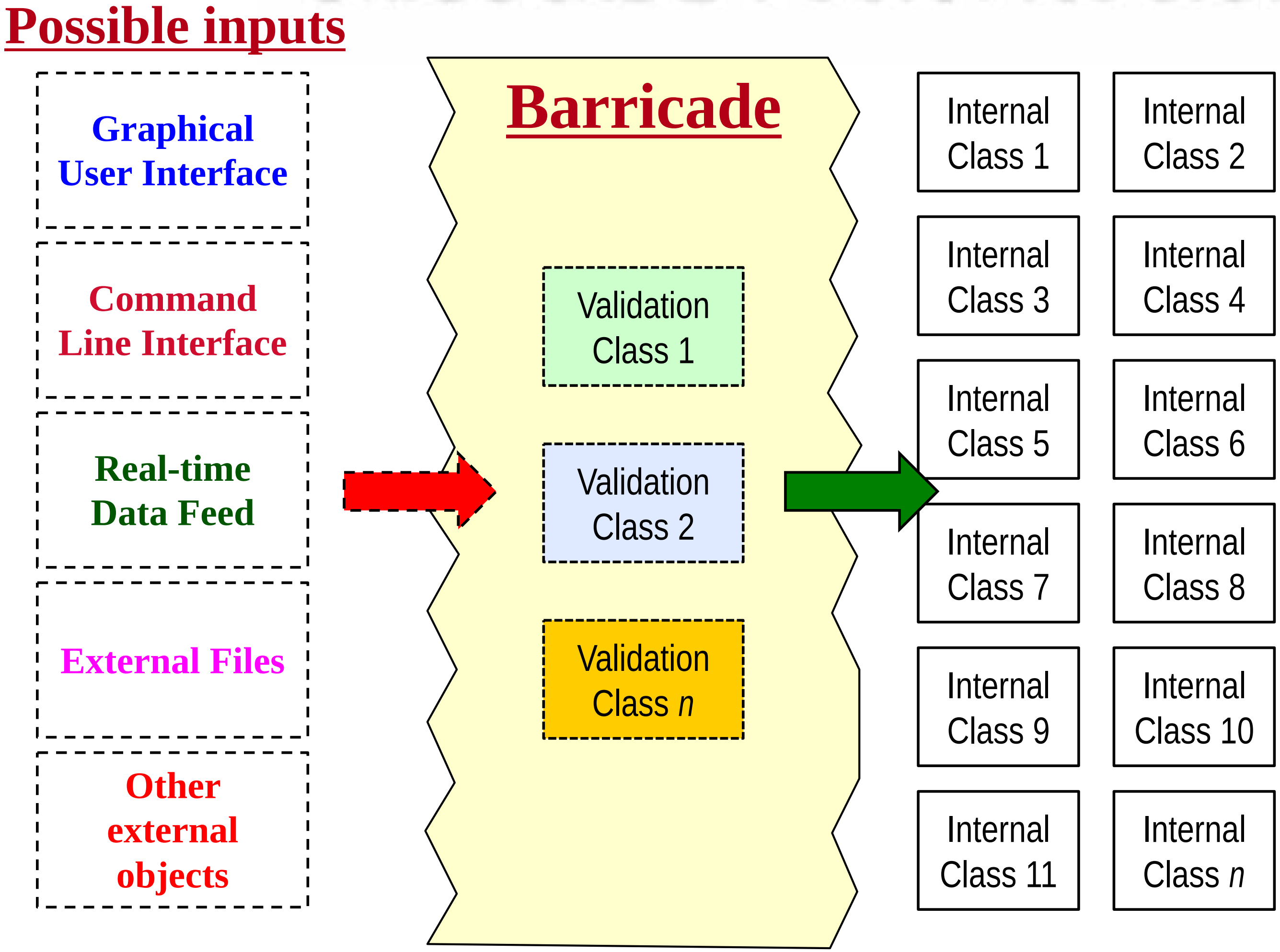
THE RULE

PROTECT YOURSELF AT ALL TIMES!

NEVER TRUST ANYONE!

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

DEFENSIVE PROGRAMMING:
BARRICADE YOUR PROGRAM



Convert input data to the proper type at input time.

Assertions are your friend!

Data here is assumed to be dirty and untrusted.

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

These classes can assume data is clean and trusted.

ASSERTIONS

<- What is true here?

```
x = 17;
```

<- What is true here?

```
y = 42;
```

<- What is true here?

```
z = x + y;
```

<- 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

ASSERTIONS: FORWARD REASONING

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

☞ **What must be true after running the code?**
(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: ??
```

ASSERTIONS: 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 to ensure that?**
  (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 of determining the relevant facts 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.

USING ASSERTIONS

- 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).

USING ASSERTIONS: GUIDELINES

- **Use assertions to document and verify preconditions and postconditions.**

- **Use assertions for conditions that should never occur.**
  - Use error-handling code for conditions you expect to happen.

- **Avoid putting executable code into assertions.**
  - Do not use for validation; the compiler may eliminate assertions.

Example: Visual C#

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

☞ **Validators are a type of assertion!**

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 piece of completed code (incremental review)

- **Who participates in the review?**
  - Either one other developer or a group of developers

- **Where does the review take place?**
  - Formal, in-person meeting or informal email/chat
    - Best to prepare beforehand: artifact is distributed in advance.
    - Preparation usually identifies more defects than the actual meeting.

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 the developer present the artifact to a group?
  - Is the purpose to only identify defects or also brainstorm fixes?
  - Is a specific methodology used?
    - E.g., "Walkthrough" = playing computer, trace values of sample data

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 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**.

☞ **Code "rots" over time if it's structure does not evolve.**

LOW-LEVEL REFACTORING

IDEs support this type of refactoring.

Names

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

Procedures

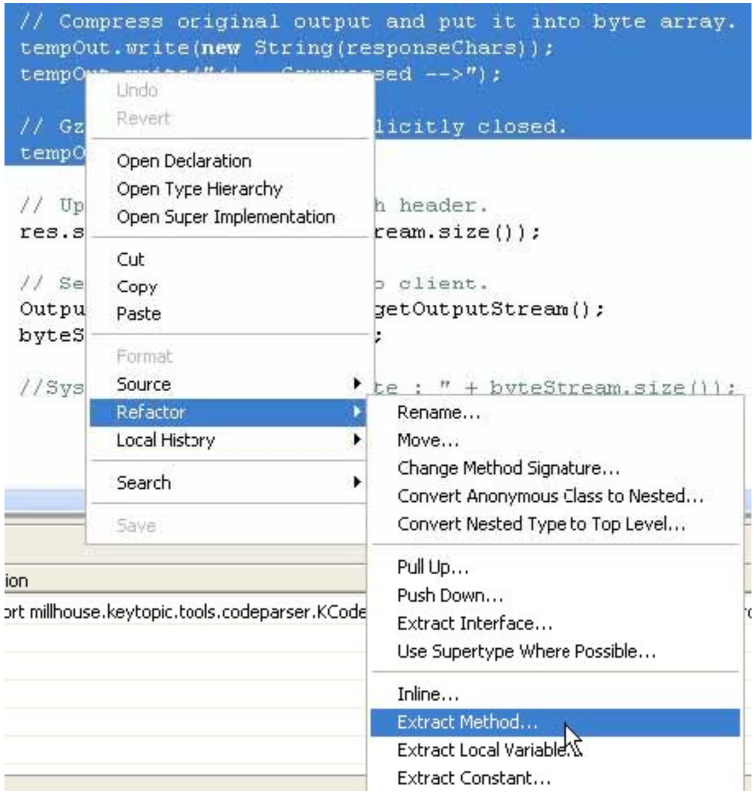
- Extracting code into a method.
- Extracting common functionality (including duplicate code) into a module/operation/etc.
- Inlining an operation/procedure.
- Changing operation signatures.

Reordering

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

LOW-LEVEL REFACTORING: IDE SUPPORT

- 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



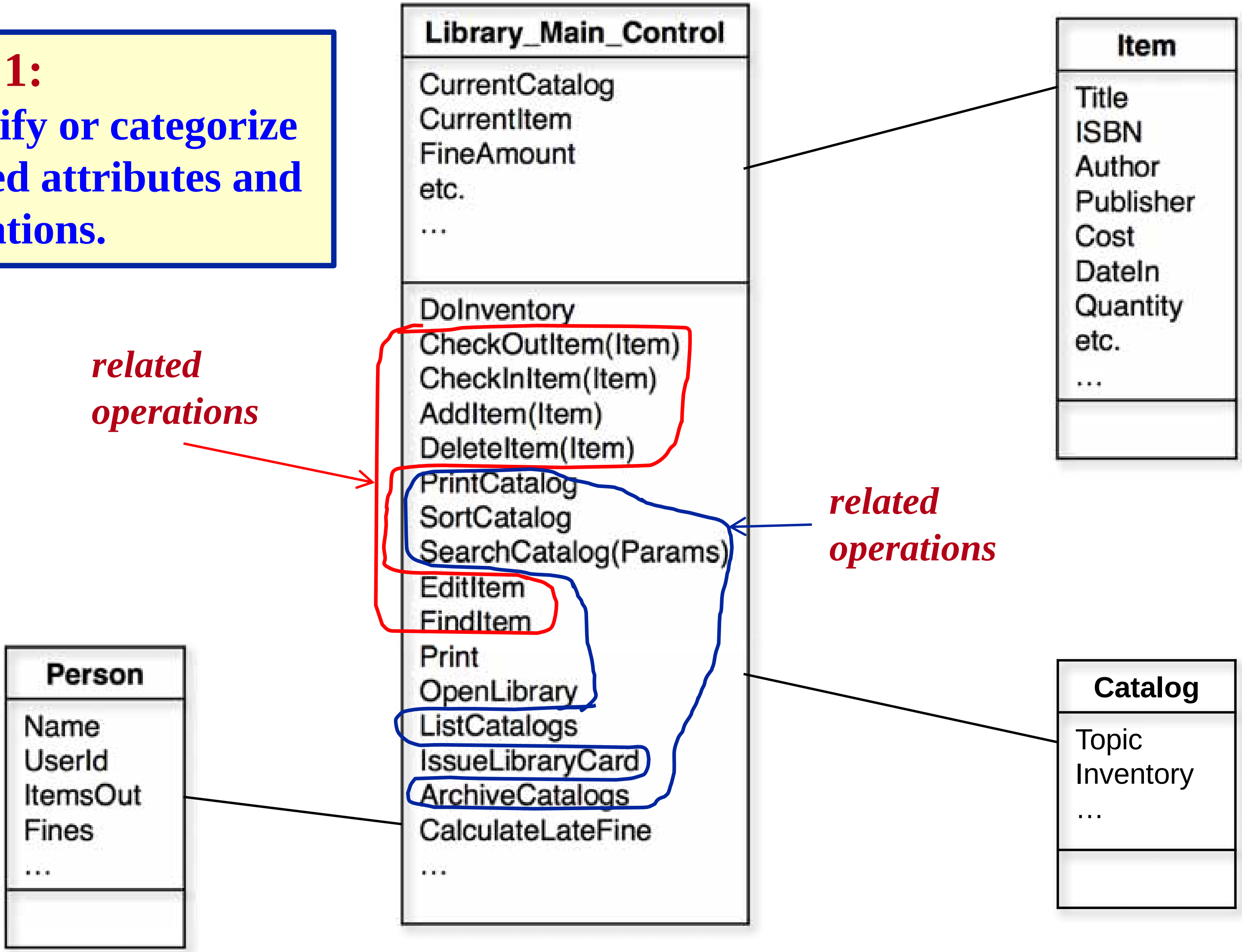
HIGHER-LEVEL REFACTORING

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

☞ **These types of refactoring increase code maintainability.**

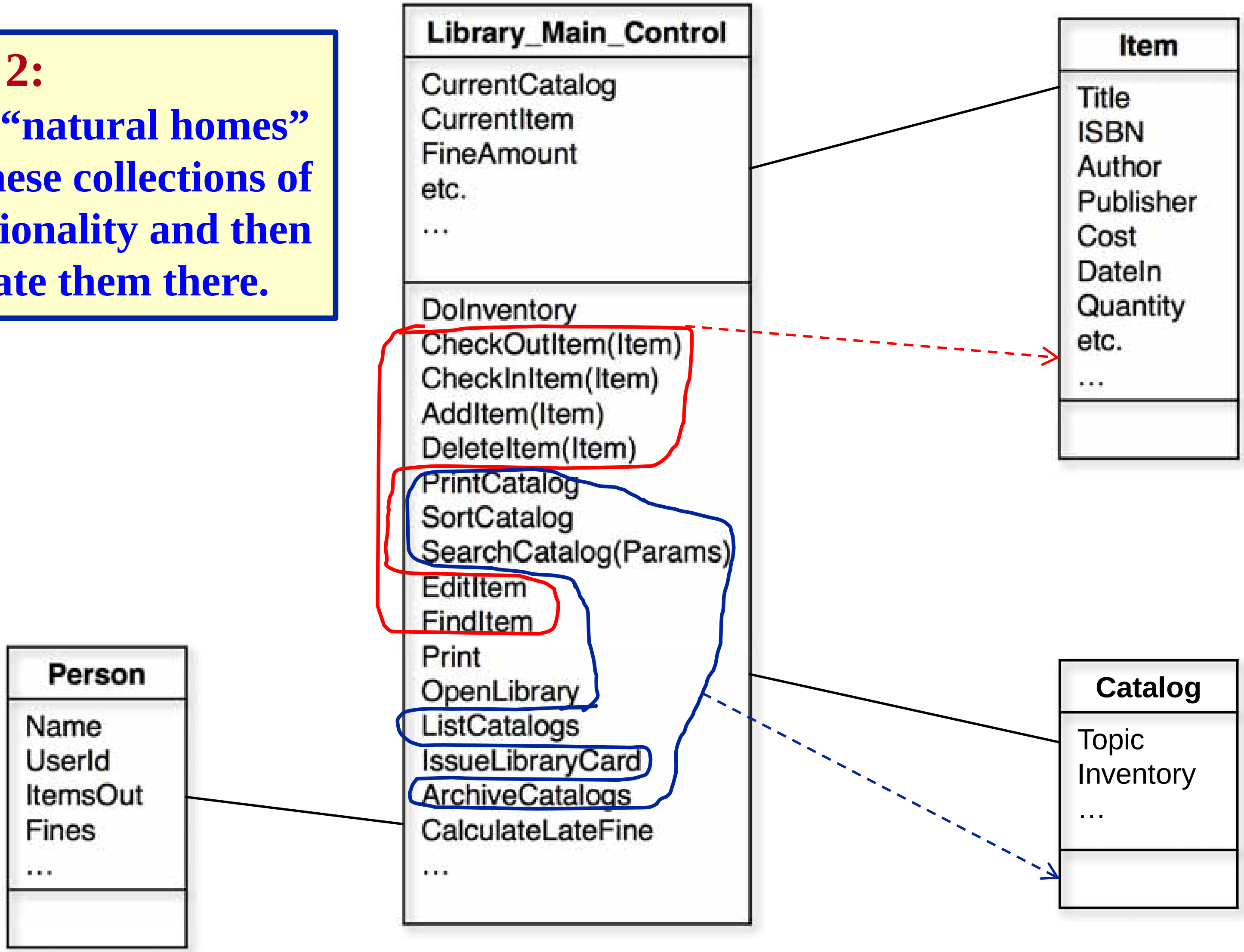
REFACTORING EXAMPLE: THE LIBRARY BLOB

Step 1:
Identify or categorize related attributes and operations.



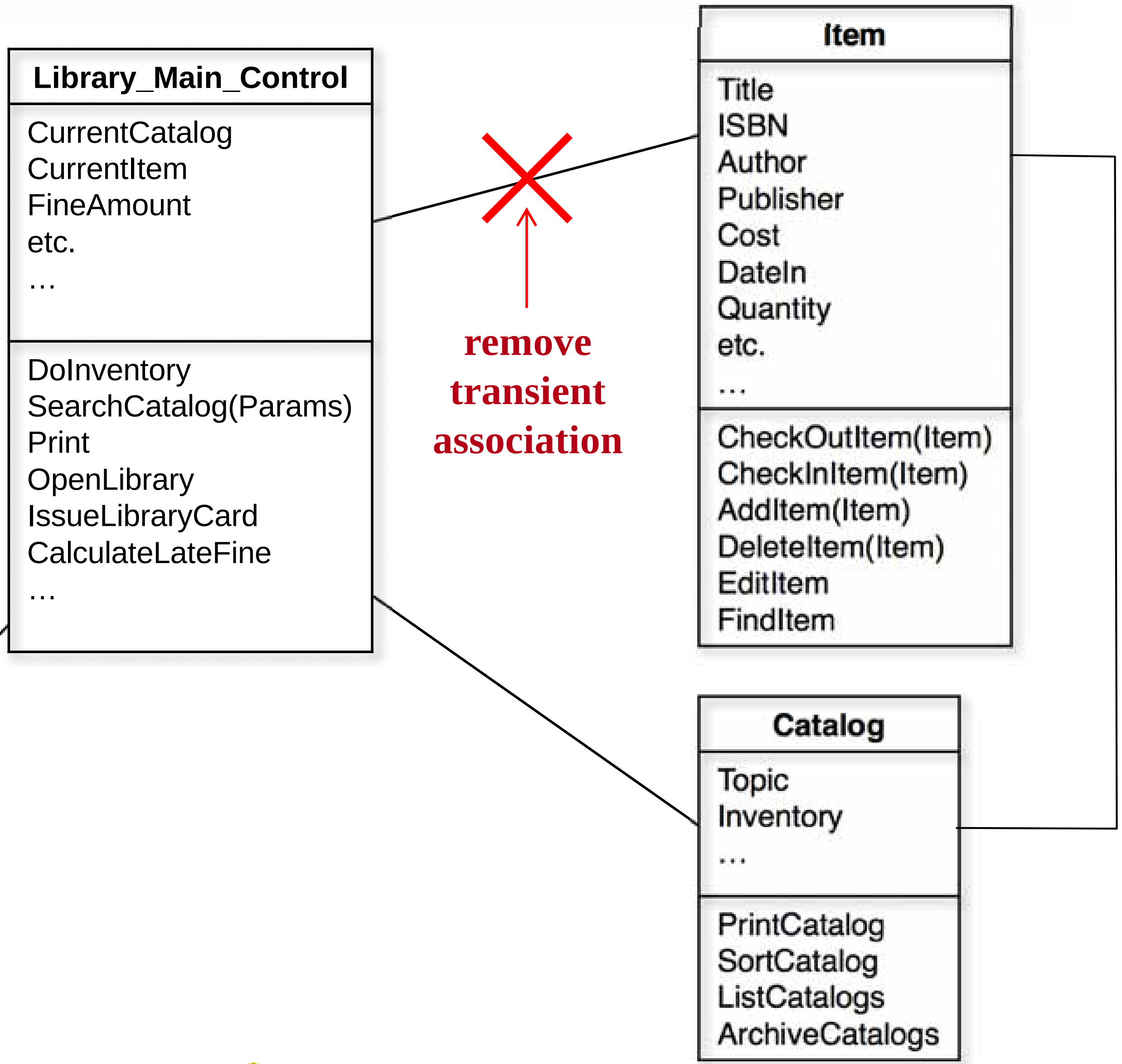
REFACTORING EXAMPLE: THE LIBRARY BLOB

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

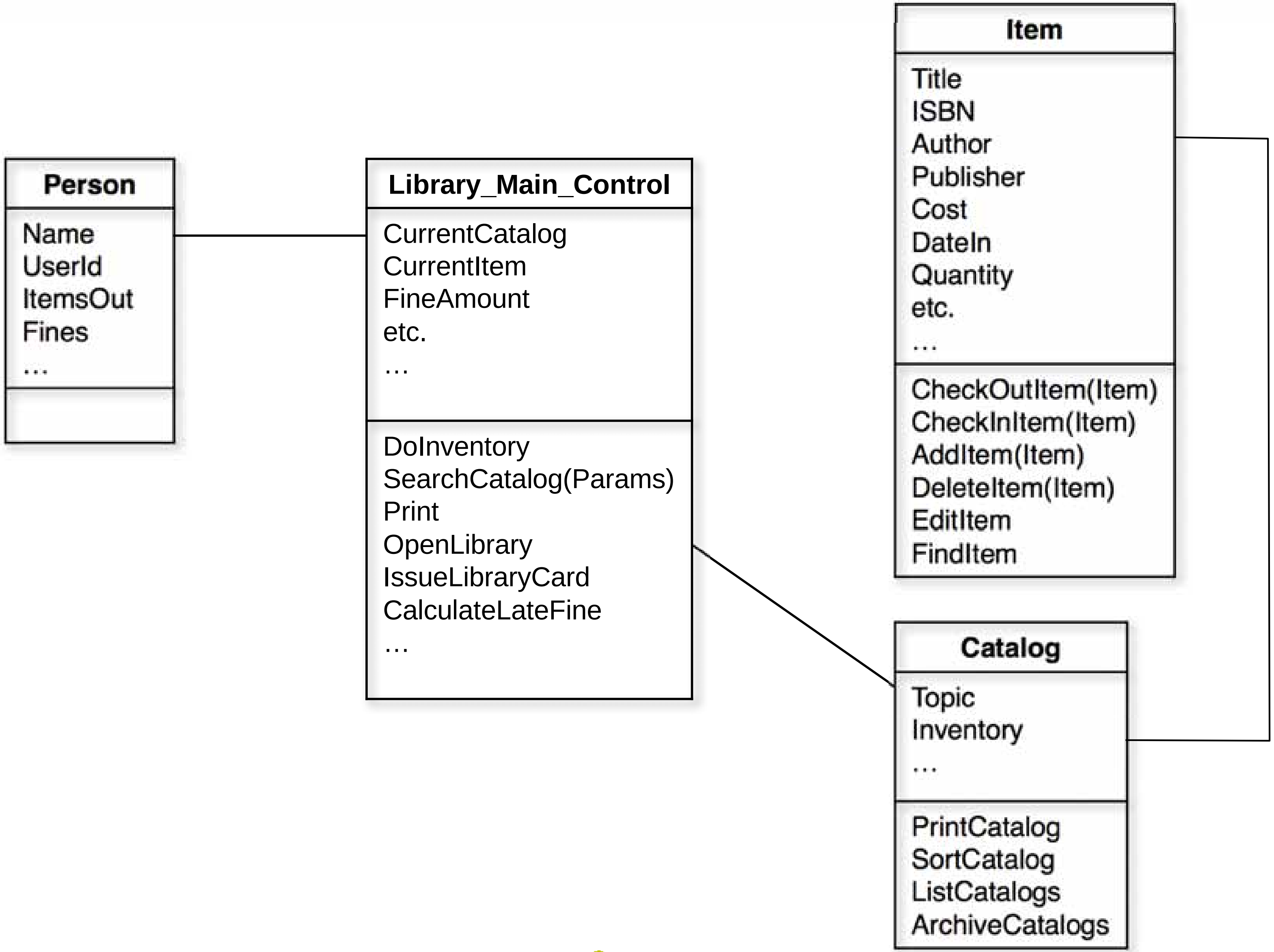


REFACTORING EXAMPLE: THE LIBRARY BLOB

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 does 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?**