

Principles of Software Construction: Objects, Design, and Concurrency

Part 2: Object-oriented analysis and design

Incremental improvements

Charlie Garrod

Bogdan Vasilescu

Administrivia

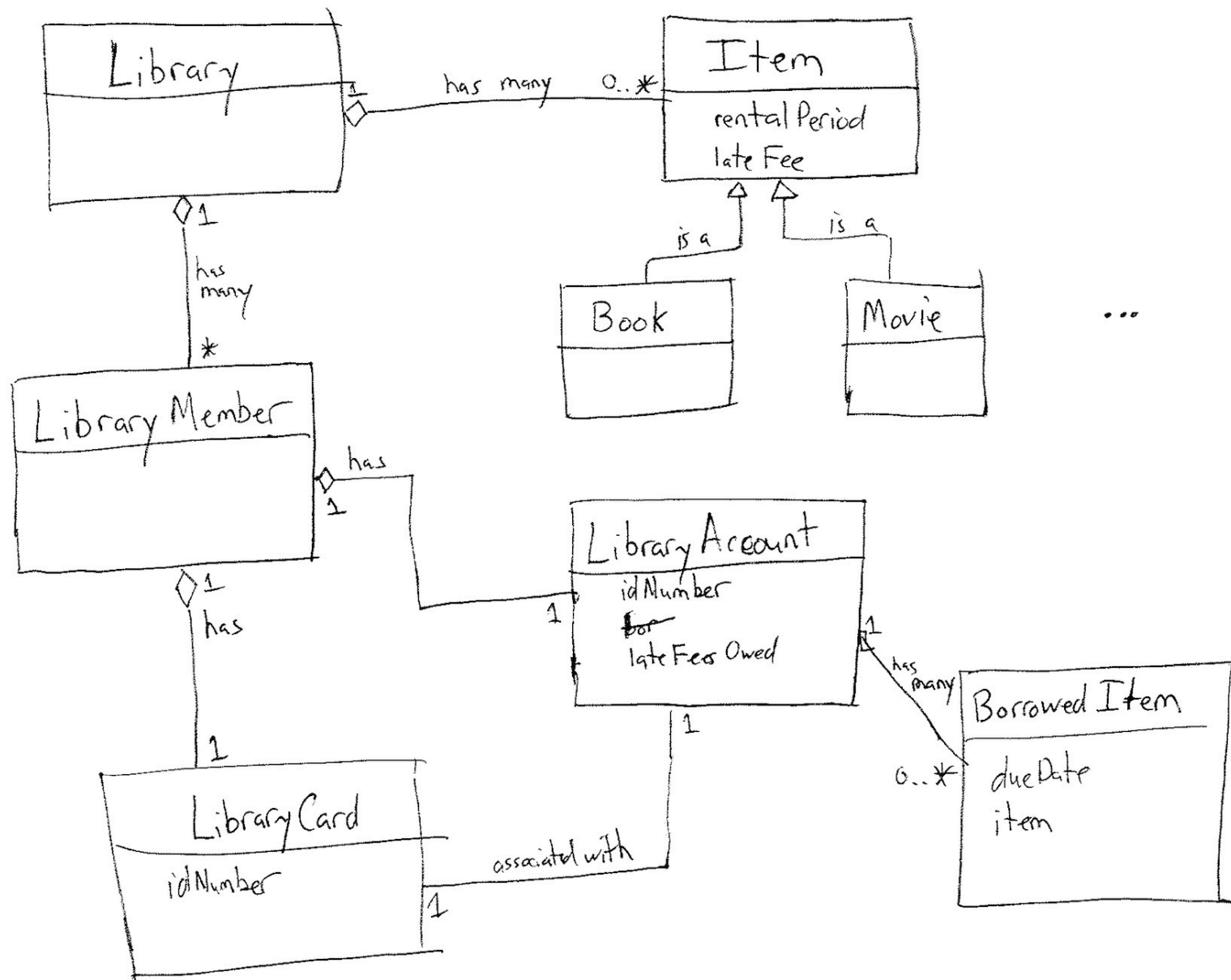
- Homework 4a due Thursday
 - Mandatory design review meeting before the homework deadline
- Midterm back today

Key concepts from last Tuesday

Artifacts of this design process

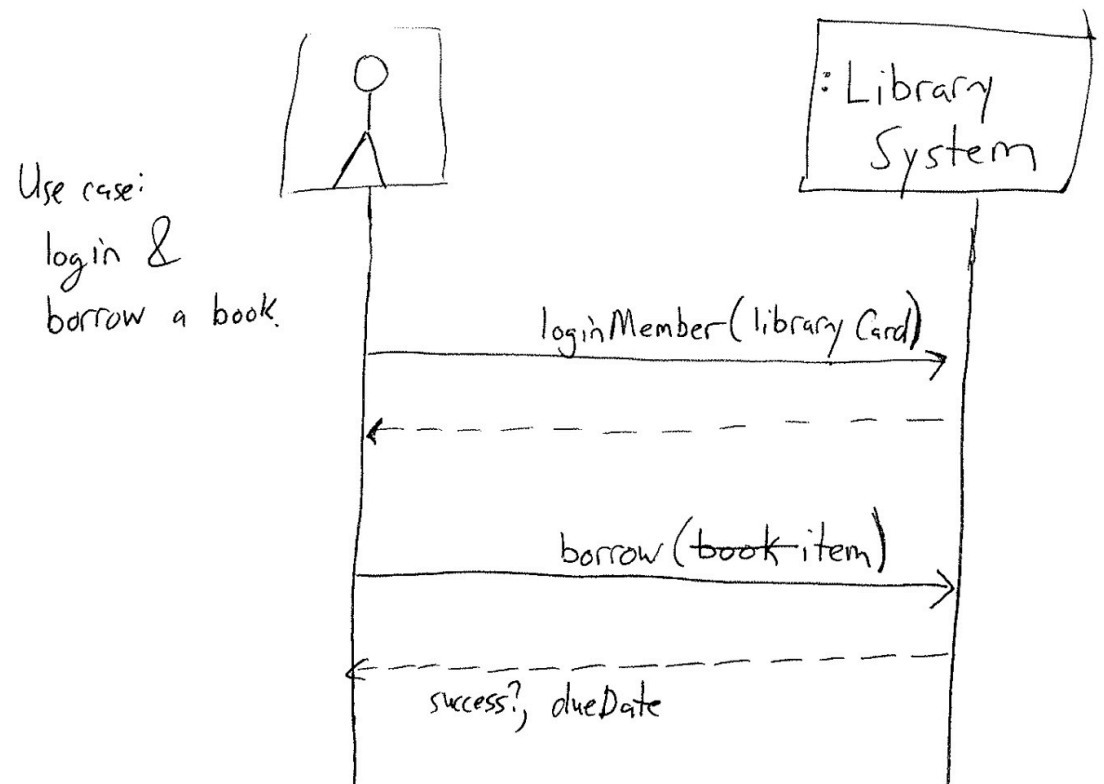
- Model / diagram the problem, define objects
 - Domain model (a.k.a. conceptual model)
 - Define system behaviors
 - System sequence diagram
 - System behavioral contracts
 - Assign object responsibilities, define interactions
 - Object interaction diagrams
 - Model / diagram a potential solution
 - Object model
-
- Understanding the problem
- Defining a solution

An example domain model for a library system



One sequence diagram for the library system

Use case scenario: A library member should be able to use her library card to log in at a library system kiosk and borrow a book. After confirming that the member has no unpaid late fees, the library system should determine the book's due date by adding its loan period to the current day, and record the book and its due date as a borrowed item in the member's library account.



A system behavioral contract for the library system

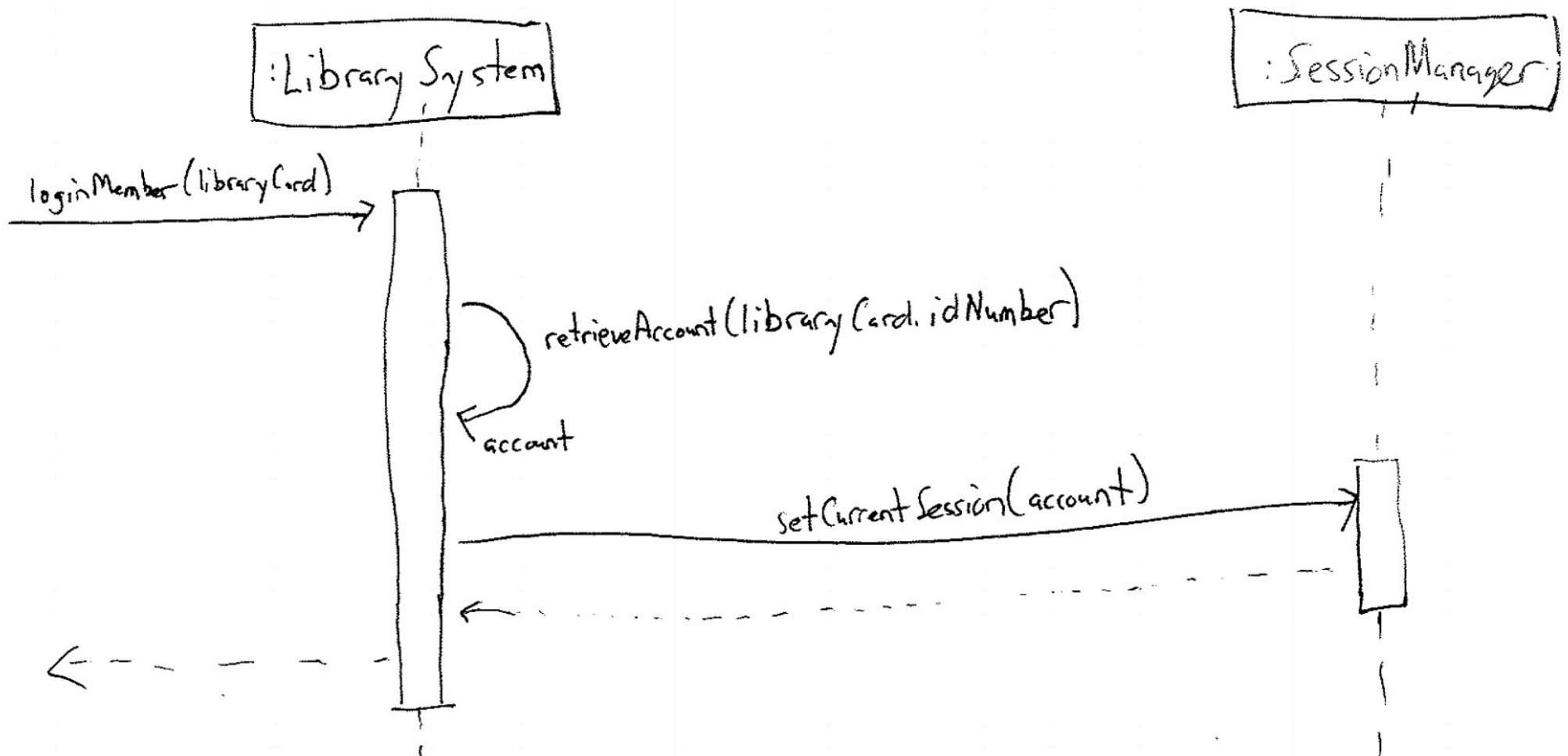
Operation: borrow(item)

Pre-conditions: Library member has already logged in to the system.
Item is not currently borrowed by another member.

Post-conditions: Logged-in member's account records the newly-borrowed item, or the member is warned she has an outstanding late fee.
The newly-borrowed item contains a future due date, computed as the item's rental period plus the current date.

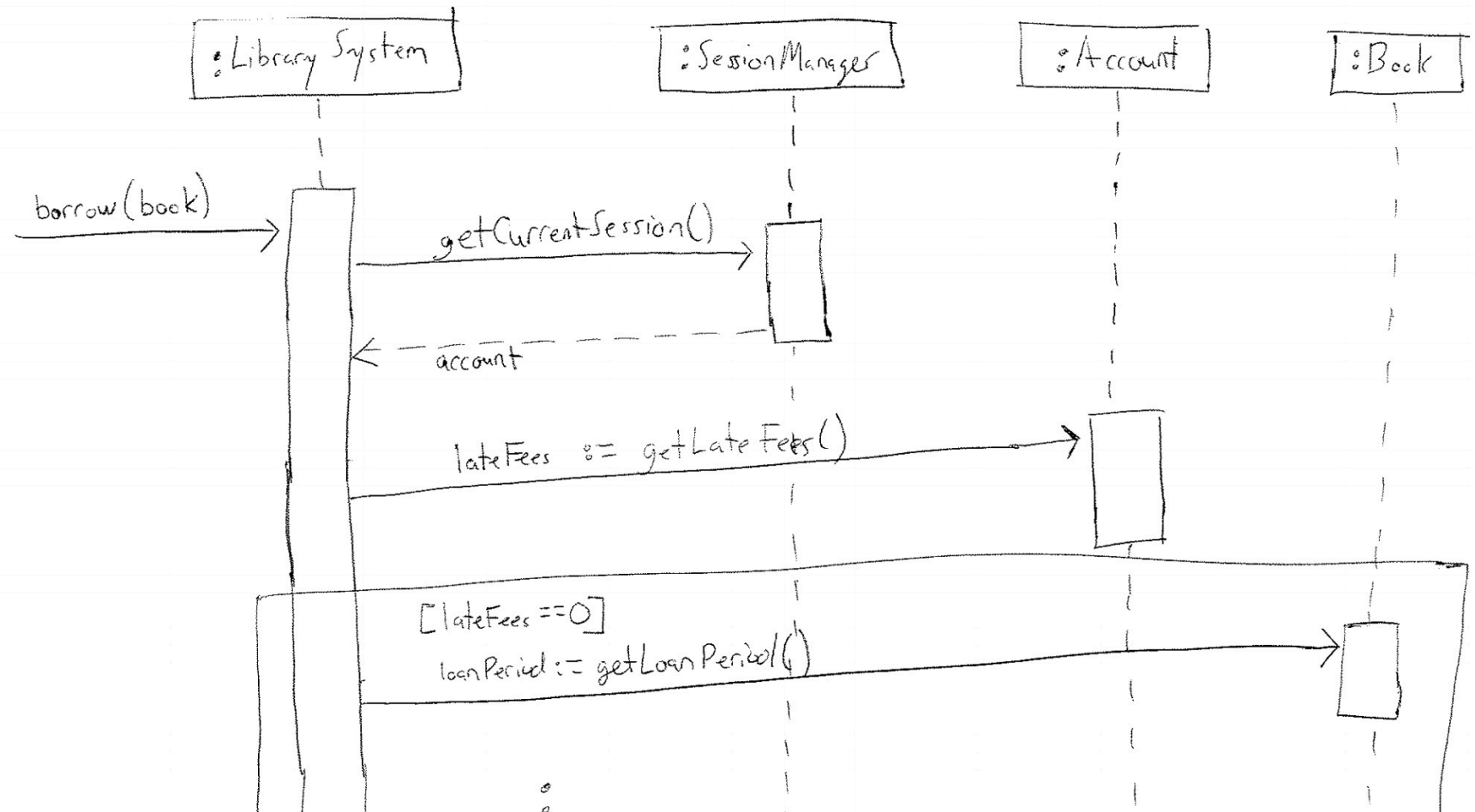
Example interaction diagram #1

Use case scenario: A library member should be able to use her library card to log in at a library system kiosk and ...

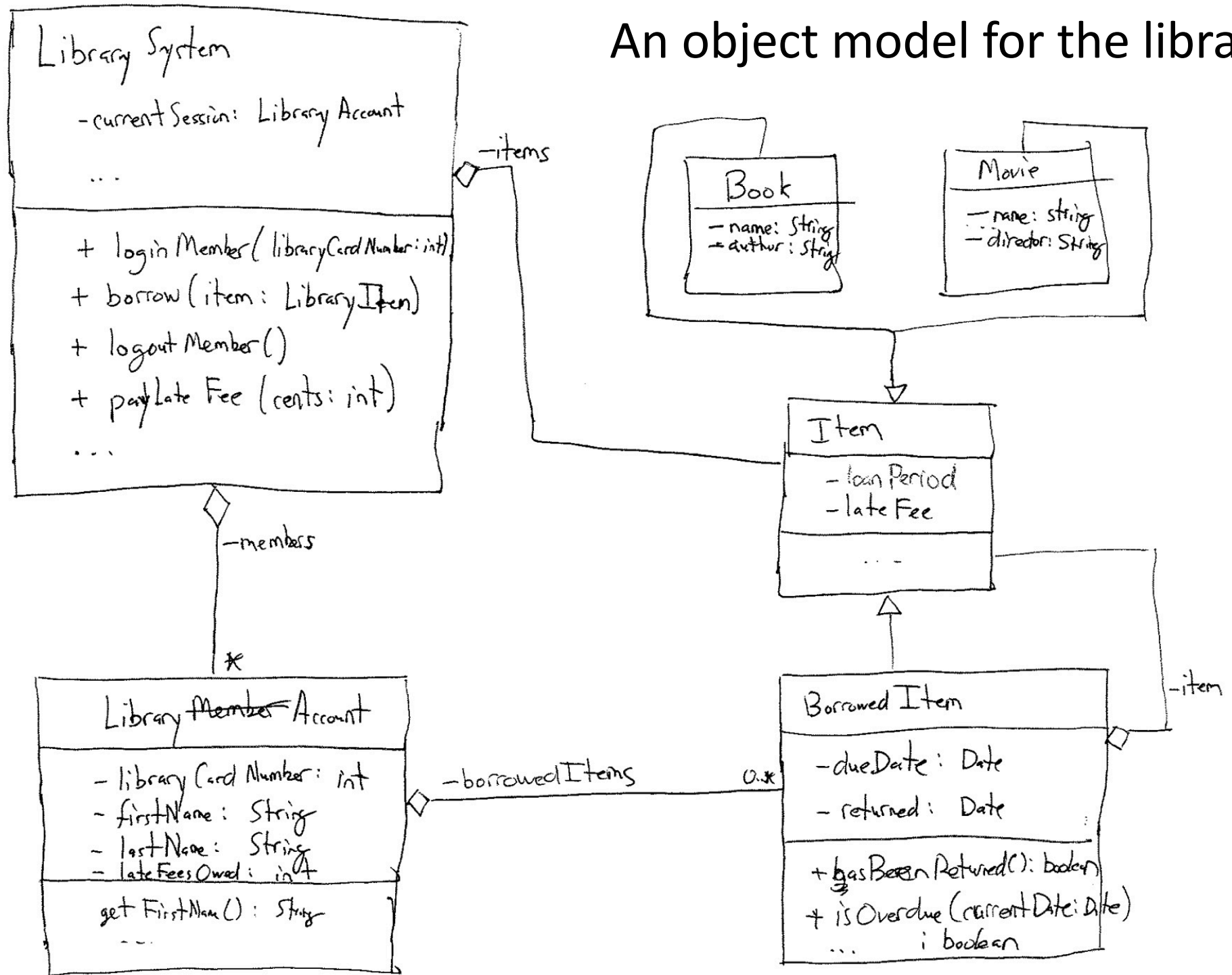


Example interaction diagram #2

Use case scenario: ...and borrow a book. After confirming that the member has no unpaid late fees, the library system should determine the book's due date by adding its loan period to the current day, and record the book and its due date as a borrowed item in the member's library account.

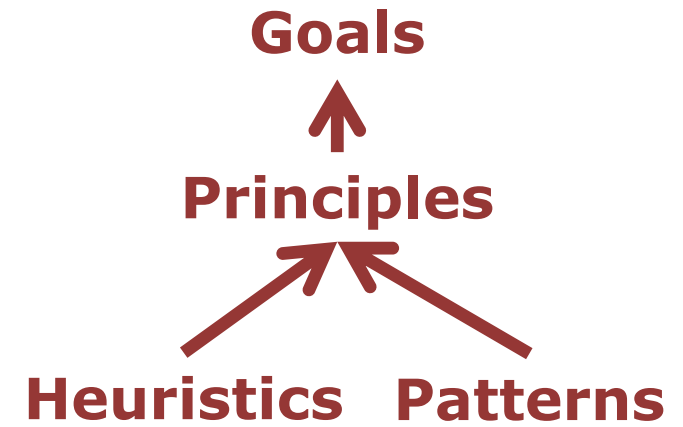


An object model for the library

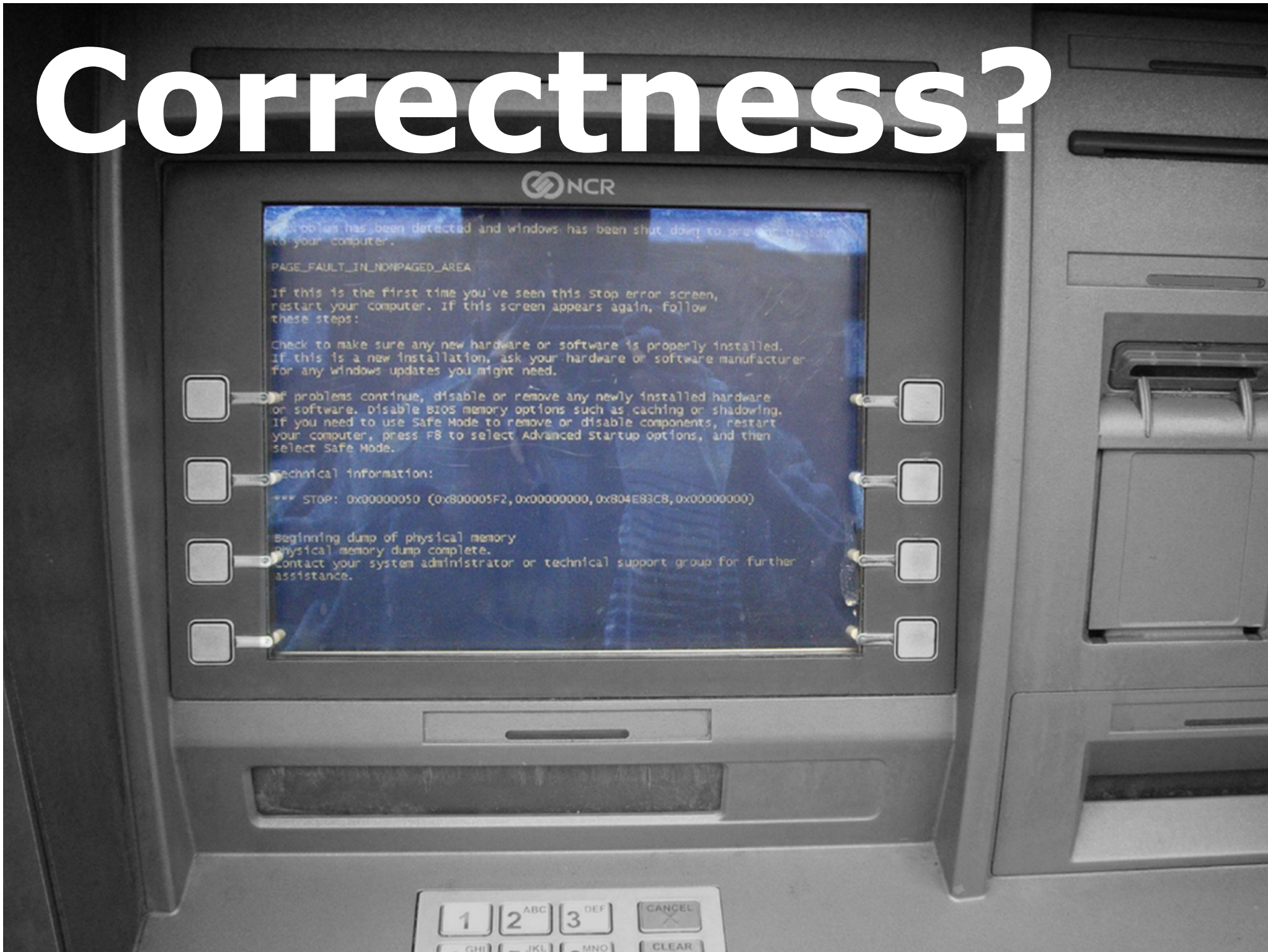


Heuristics for responsibility assignment

- Controller heuristic
- Information expert heuristic
- Creator heuristic



Correctness?



Software Errors

- Functional errors
- Performance errors
- Deadlock
- Race conditions
- Boundary errors
- Buffer overflow
- Integration errors
- Usability errors
- Robustness errors
- Load errors
- Design defects
- Versioning and configuration errors
- Hardware errors
- State management errors
- Metadata errors
- Error-handling errors
- User interface errors
- API usage errors
- ...

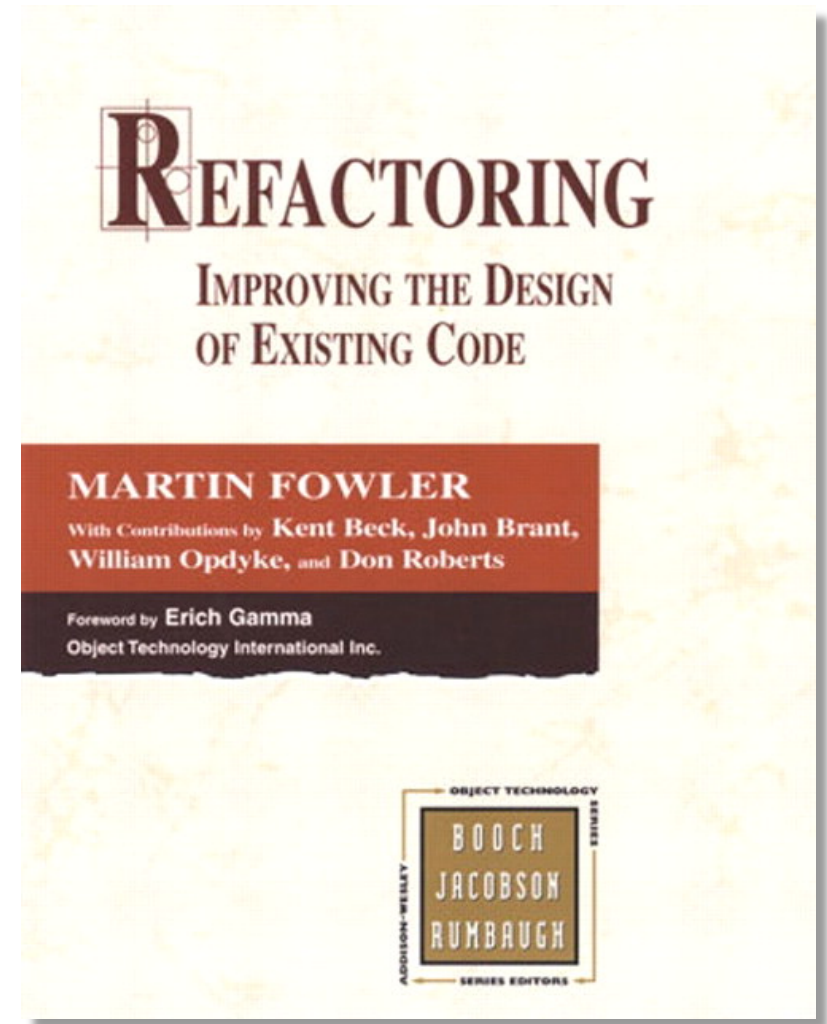
Software Errors

- **Functional errors**
- Performance errors
- Deadlock
- Race conditions
- Boundary errors
- Buffer overflow
- Integration errors
- Usability errors
- Robustness errors
- Load errors
- **Design defects**
- Versioning and configuration errors
- Hardware errors
- State management errors
- Metadata errors
- Error-handling errors
- User interface errors
- API usage errors
- ...

CODE SMELLS

Bad Smells -> Design Defects

- Bad Smells indicate that your code is ripe for refactoring
- Refactoring is about **how** to change code by applying refactorings
- Bad smells are about **when** to modify it

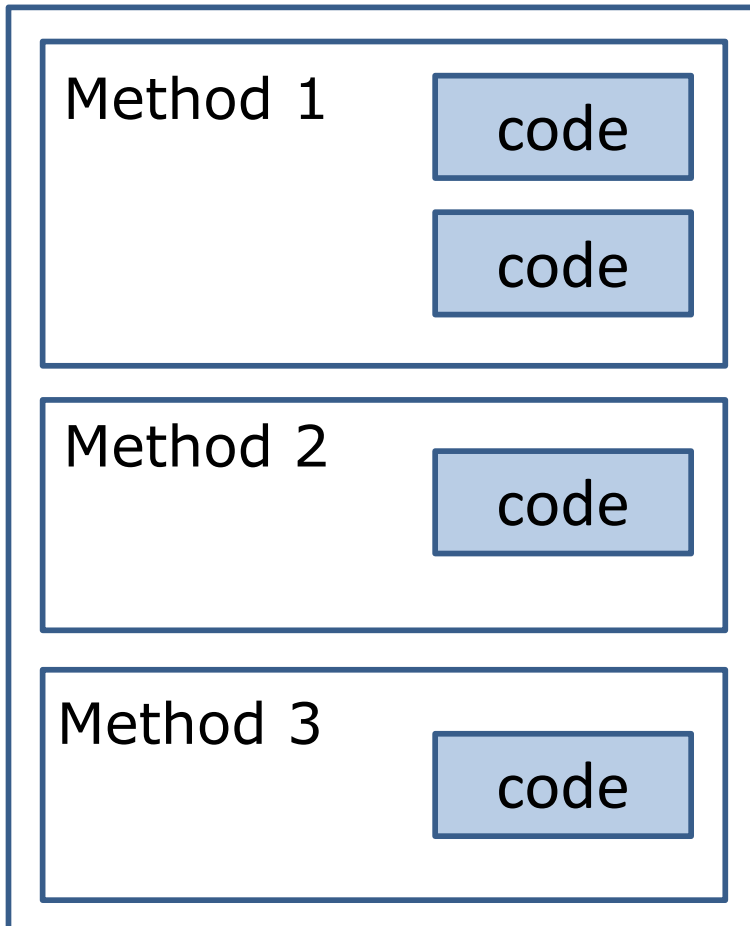


Bad Smells: Classification

- Top crime: **code duplication**
- Class / method organization
 - Large class, **Long Method**, Long Parameter List, Lazy Class, Data Class, ...
- Lack of loose coupling or cohesion
 - Inappropriate Intimacy, **Feature Envy**, Data Clumps, ...
- Too much or too little delegation
 - Message Chains, **Middle Man**, ...
- Non Object-Oriented control or data structures
 - Switch Statements, Primitive Obsession, ...
- Other: **Comments**

Code duplication (1)

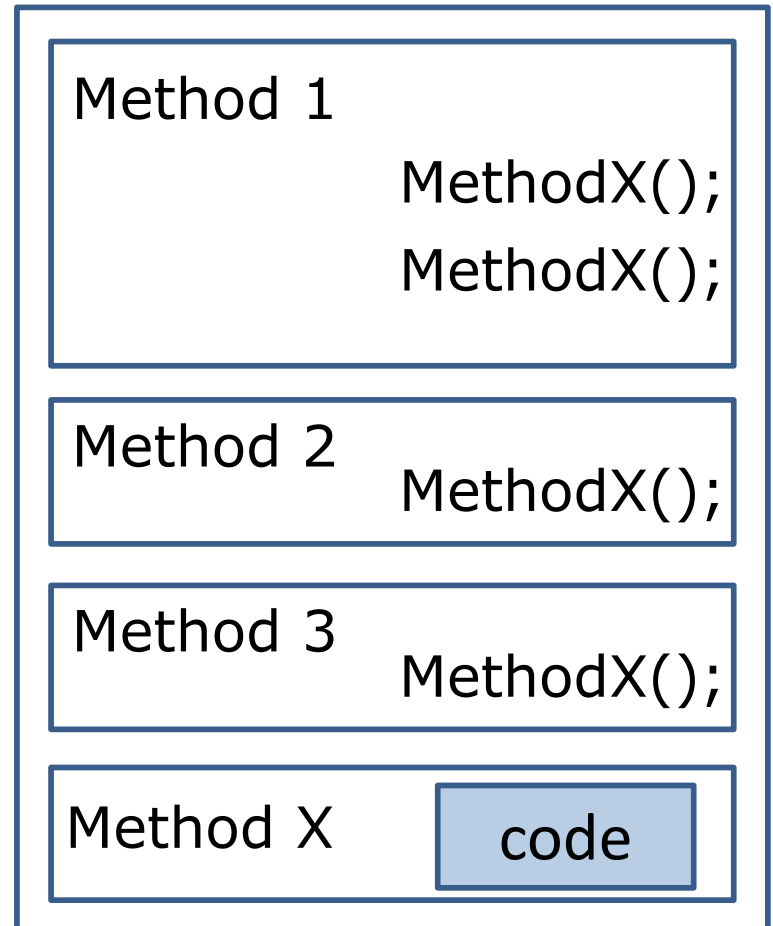
Class



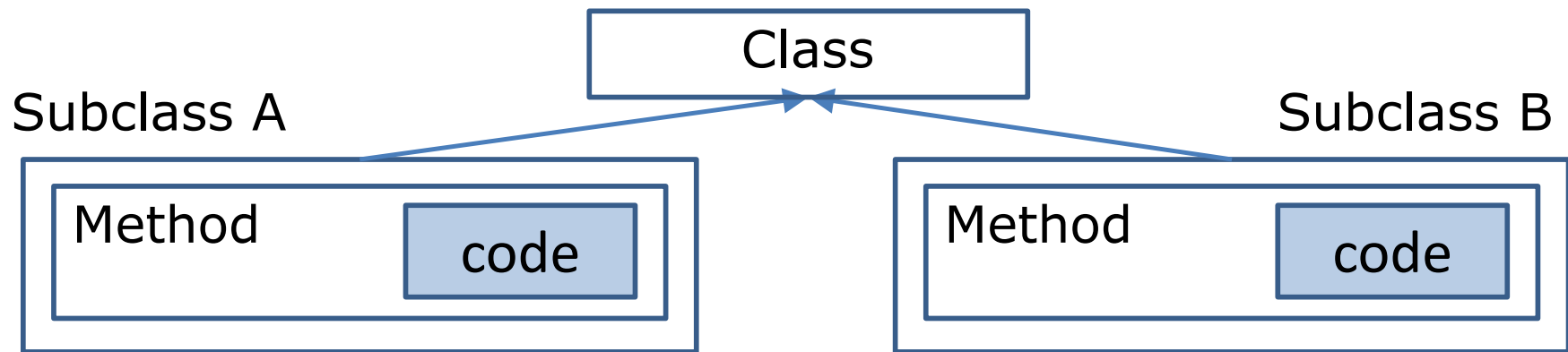
- Extract method
- Rename method



Class



Code duplication (2)

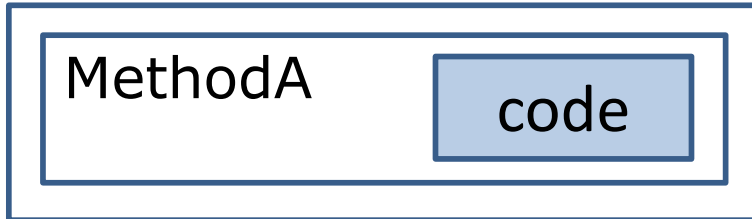


Same expression in two sibling classes:

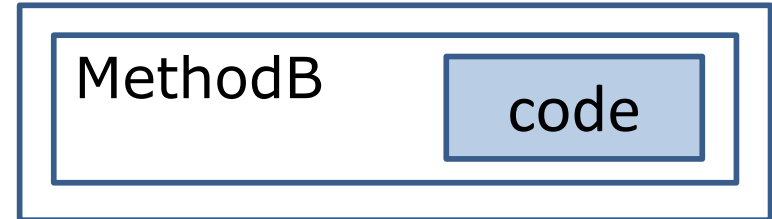
- Same code: Extract method + Pull up field
- Similar code: Extract method + Form Template Method
- Different algorithm: Substitute algorithm

Code duplication (3)

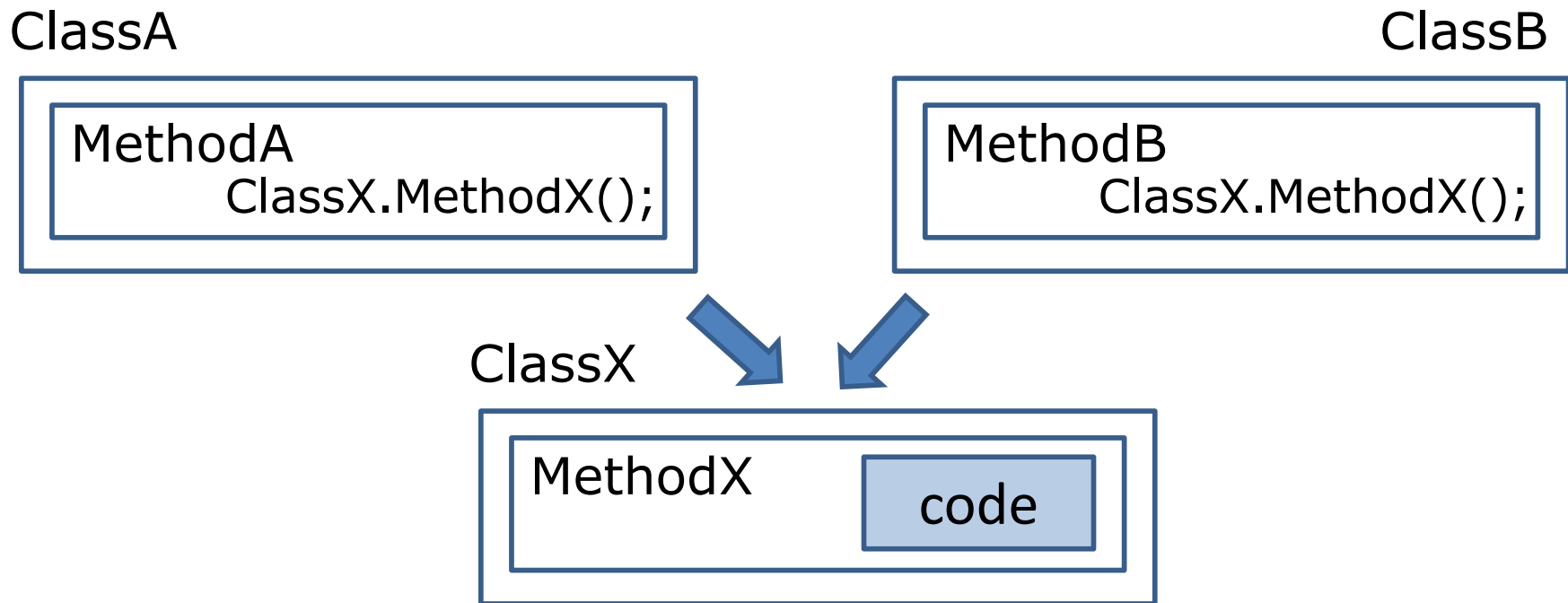
ClassA



ClassB



Code duplication (3)



Same expression in two unrelated classes:

- Extract class
- If the method really belongs in one of the two classes, keep it there and invoke it from the other class

Long method

```
//700LOC
public boolean foo() {
    try {
        synchronized () {
            if () {
            } else {
            }
            for () {
                if () {
                    if () {
                        if () {
                            if ()?
                            {
                                if () {
                                    for () {
                                    }
                                }
                            }
                        } else {
                            if () {
                                for () {
                                    if () {
                                    } else {
                                    }
                                    if () {
                                    } else {
                                        if () {
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    } else {
        if () {
            for () {
                if () {
                } else {
                }
                if () {
                } else {
                    if () {
                    }
                }
            }
        }
    }
}
```

- Remember this?

Source:
<http://thedailywtf.com/Articles/Coding-Like-the-Tour-de-France.aspx>

Solution: Refactoring

- Refactoring is a change to a program that doesn't change the behavior, but improves a non-functional attribute of the code (not reworking).
- Examples:
 - Improve readability
 - Reduce complexity
- Benefits include increased maintainability, and easier extensibility
- Fearlessly refactor when you have good unit tests

Refactoring a long method

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    // Print banner
    System.out.println("*****");
    System.out.println("***** Customer *****");
    System.out.println("*****");
    // Calculate outstanding
    While (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    // Print details
    System.out.println("name: " + _name);
    System.out.println("amount" + outstanding);
}
```

Refactoring a long method

```
void printOwing() {  
    Enumeration e = _orders.elements();  
    double outstanding = 0.0;  
    // Print banner  
    System.out.println("*****");  
    System.out.println("***** Customer *****");  
    System.out.println("*****");  
    // Calculate outstanding  
    While (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        outstanding += each.getAmount();  
    }  
    // Print details  
    System.out.println("name: " + _name);  
    System.out.println("amount" + outstanding);  
}
```

Refactoring a long method

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // Calculate outstanding
    While (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    // Print details
    System.out.println("name: " + _name);
    System.out.println("amount" + outstanding);
}
```

```
void printBanner(){
    System.out.println("*****");
    System.out.println("***** Customer *****");
    System.out.println("*****");
}
```

Extract method

Compile and test to see whether I've broken anything

Refactoring a long method

```
void printOwing() {  
    Enumeration e = _orders.elements();  
    double outstanding = 0.0;  
  
    printBanner();  
  
    // Calculate outstanding  
    While (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        outstanding += each.getAmount();  
    }  
    // Print details  
    System.out.println("name: " + _name);  
    System.out.println("amount" + outstanding);  
}  
void printBanner(){...}
```


Refactoring a long method

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // Calculate outstanding
    While (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    printDetails(outstanding);
}
void printBanner(){...}
void printDetails(outstanding){
    System.out.println("name: " + _name);
    System.out.println("amount" + outstanding);
}
```

Extract method
using local variables



Compile and test to see whether I've broken anything

Refactoring a long method

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // Calculate outstanding
    While (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    printDetails(outstanding);
}

void printBanner(){...}
void printDetails(outstanding){
    System.out.println("name: " + _name);
    System.out.println("amount" + outstanding);
}
```

Refactoring a long method

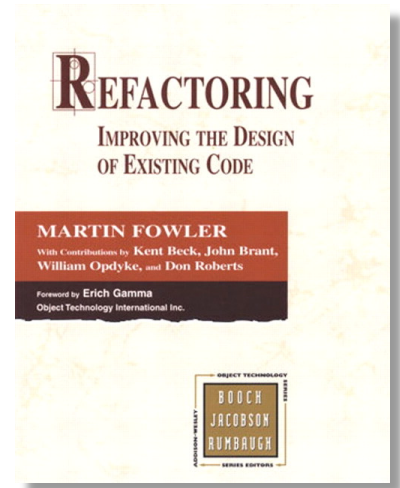
```
void printOwing() {  
    Enumeration e = _orders.elements();  
    double outstanding = getOutstanding();  
    printBanner();  
    printDetails(outstanding);  
}  
void printBanner(){...}  
void printDetails(outstanding){...}
```

```
double getOutstanding() {  
    Enumeration e = _orders.elements();  
    double result = 0.0;  
    While (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        result += each.getAmount();  
    }  
    return result;  
}
```

Extract method
reassigning a local
variable

Compile and test to see whether I've broken anything

Many More Bad Smells and Suggested Refactorings



- Top crime: **code duplication**
- Class / method organization
 - Large class, **Long Method**, Long Parameter List, Lazy Class, Data Class, ...
- Lack of loose coupling or cohesion
 - Inappropriate Intimacy, **Feature Envy**, Data Clumps, ...
- Too much or too little delegation
 - Message Chains, **Middle Man**, ...
- Non Object-Oriented control or data structures
 - Switch Statements, Primitive Obsession, ...
- Other: **Comments**

BACK TO FUNCTIONAL CORRECTNESS

Reminder: Functional Correctness

- The compiler ensures that the types are correct (type checking)
 - Prevents “Method Not Found” and “Cannot add Boolean to Int” errors at runtime
- Static analysis tools (e.g., FindBugs) recognize certain common problems
 - Warns on possible NullPointerExceptions or forgetting to close files
- How to ensure functional correctness of contracts beyond?

Reminder: Formal Verification

- Proving the correctness of an implementation with respect to a formal specification, using formal methods of mathematics.
- Formally prove that all possible executions of an implementation fulfill the specification
- Manual effort; partial automation; not automatically decidable

Reminder: Testing

- Executing the program with selected inputs in a controlled environment (dynamic analysis)
- Goals:
 - Reveal bugs (main goal)
 - Assess quality (hard to quantify)
 - Clarify the specification, documentation
 - Verify contracts

**"Testing shows the presence,
not the absence of bugs**

Edsger W. Dijkstra 1969

Reminder: Testing Decisions

- Who tests?
 - Developers
 - Other Developers
 - Separate Quality Assurance Team
 - Customers
- **When to test?**
 - Before development
 - During development
 - After milestones
 - Before shipping
- **When to stop testing?**

(More in 17-313)

Reminder: Code coverage metrics (useful but dangerous)

- Method coverage – coarse
- Branch coverage – fine
- Path coverage – too fine
 - Cost is high, value is low
 - (Related to *cyclomatic complexity*)

Blackbox: Covering Specifications

- Looking at specifications, not code:
- Test representative case
- Test boundary condition
- Test exception conditions
- (Test invalid case)

Structural Analysis of System under Test

- Organized according to program decision structure

```
public static int binsrch (int[] a, int key) {  
  
    int low  = 0;  
    int high = a.length - 1;  
  
    while (true) {  
  
        if ( low > high ) return -(low+1);  
  
        int mid = (low+high) / 2;  
  
        if      ( a[mid] < key ) low  = mid + 1;  
        else if ( a[mid] > key ) high = mid - 1;  
        else      return mid;  
    }  
}
```

Structural Analysis of System under Test

- Organized according to program decision structure

```
public static int bsrch (int[] a, int key) {  
  
    int low  = 0;  
    int high = a.length - 1;  
  
    while (true) {  
        if ( low > high ) return -(low+1);  
  
        int mid = (low+high) / 2;  
  
        if      ( a[mid] < key ) low  = mid + 1;  
        else if ( a[mid] > key ) high = mid - 1;  
        else      return mid;  
    }  
}
```

Will this statement get executed in a test?
Does it return the correct result?

Structural Analysis of System under Test

- Organized according to program decision structure

```
public static int binsrch (int[] a, int key) {
```

```
    int low  = 0;  
    int high = a.length - 1;
```

```
    while (true) {
```

```
        if ( low > high ) return -(low+1);
```

```
        int mid = (low+high) / 2;
```

```
        if ( a[mid] < key ) low  = mid + 1;
```

```
        else if ( a[mid] > key ) high = mid - 1;
```

```
        else return mid;
```

```
    }
```

```
}
```

Will this statement get executed in a test?
Does it return the correct result?

Could this array index be out of bounds?

Structural Analysis of System under Test

- Organized according to program decision structure

```
public static int binsrch (int[] a, int key) {
```

```
    int low  = 0;  
    int high = a.length - 1;
```

Will this statement get executed in a test?
Does it return the correct result?

```
    while (true) {
```

```
        if ( low > high ) return -(low+1);
```

```
        int mid = (low+high) / 2;
```

```
        if ( a[mid] < key ) low  = mid + 1;
```

```
        else if ( a[mid] > key ) high = mid - 1;
```

```
        else return mid;
```

Could this array index be out of bounds?

Does this return statement ever get reached?

Test suites – ideal vs. real

- Ideal test suites
 - Uncover all errors in code
 - Test “non-functional” attributes such as performance and security
 - Minimum size and complexity
- Real test Suites
 - Uncover some portion of errors in code
 - Have errors of their own
 - Are nonetheless priceless

STATIC ANALYSIS

Stupid Bugs

```
public class CartesianPoint {  
    private int x, y;  
    int getX() { return this.x; }  
    int getY() { return this.y; }  
    public boolean equals(CartesianPoint that) {  
        return (this.getX()==that.getX()) &&  
            (this.getY() == that.getY());  
    }  
}
```

Stupid Subtle Bugs

```
public class Object {  
    public boolean equals(Object other) { ... }  
  
    // other methods...  
}
```

classes with no
explicit superclass
implicitly extend
Object

```
public class CartesianPoint extends Object {  
    private int x, y;  
    int getX() { return this.x; }  
    int getY() { return this.y; }  
    public boolean equals(CartesianPoint that) {  
        return (this.getX()==that.getX()) &&  
            (this.getY()== that.getY());  
    }  
}
```

can't change
argument type
when overriding

This defines a
different equals
method, rather
than overriding
Object.equals()

Fixing the Bug

```
public class CartesianPoint {  
    private int x, y;  
    int getX() { return this.x; }  
    int getY() { return this.y; }
```

Declare our intent
to override;
Compiler checks
that we did it

Use the same
argument type as
the method we
are overriding

@Override

```
public boolean equals(Object o) {  
    if (!(o instanceof CartesianPoint)  
        return false;
```

Check if the
argument is a
CartesianPoint.
Correctly returns
false if o is null

```
    CartesianPoint that = (CartesianPoint) o;
```

```
return (this.getX() == that.getX()) &&  
        (this.getY() == that.getY());
```

Create a variable
of the right type,
initializing it with
a cast

```
}
```

```
}
```

FindBugs

The screenshot shows an IDE with the following components:

- Editor:** Displays `CartesianPoint.java` with the following code:

```
public boolean equals(CartesianPoint p) {  
    return (p.x==this.x) && (p.y==this.y);  
}
```
- Toolbar:** Includes icons for Pro, Java, Dec, Sea, Co, Pro, Cov, His, Bug, Call, and Ana.
- FindBugs Summary:** Shows "0 errors, 2 warnings, 0 others".

Description	Resou
FindBugs Problem (Of concern) (1 item)	
CartesianPoint defines equals and uses Object.hashCode()	Cartes
FindBugs Problem (Scary) (1 item)	
CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)	Cartes
- Bug Info:** Details the selected bug.
 - Location:** CartesianPoint.java: 12
 - Navigation:** CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)
 - Bug Description:** **Bug:** CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)
 - Explanation:** This class defines a covariant version of the `equals()` method, but inherits the normal `equals(Object)` method defined in the base `java.lang.Object` class. The class should probably define a `boolean equals(Object)` method.
 - Confidence:** Normal, **Rank:** Scary (8)
 - Pattern:** EQ_SELF_USE_OBJECT
 - Type:** Eq, **Category:** CORRECTNESS (Correctness)

FindBugs



```
514 Scanning archives (4 / 4)
515 2 analysis passes to perform
516 Pass 1: Analyzing classes (38 / 38) - 100% complete
517 Pass 2: Analyzing classes (4 / 4) - 100% complete
518 Done with analysis
519 FindBugs rule violations were found. See the report at: file:///home/travis/build/CMU-15-
214/[REDACTED]/homework/3/build/reports/findbugs/test.html
520 :homework/3:test
```

CheckStyle

The screenshot shows an IDE window titled 'CartesianPoint.java'. The code defines a `CartesianPoint` class with private fields `X` and `Y`, a constructor, and getter methods. The IDE's right sidebar contains panels for 'Task List', 'Connect Mylyn', and 'Outline'. The 'Outline' panel shows the class structure with `X: int` and `Y: int` fields. The bottom panel displays the CheckStyle error list, indicating 9 warnings. The warnings include issues with whitespace, tab characters, and naming conventions for the `GetY`, `X`, and `Y` identifiers.

```
public final class CartesianPoint {  
    private int X,Y;  
  
    CartesianPoint(int x, int y) {  
        this.X=x;  
        this.Y = y;  
    }  
  
    public int GetY() {  
        return Y;  
    }  
  
    public int getX() {  
        return X;  
    }  
}
```

0 errors, 9 warnings, 0 others

Description	Resou
▼ ⚠ Checkstyle Problem (9 items)	
⚠ ',' is not followed by whitespace.	Carte
⚠ '=' is not followed by whitespace.	Carte
⚠ '=' is not preceded with whitespace.	Carte
⚠ File contains tab characters (this is the first instance).	Carte
⚠ Name 'GetY' must match pattern '^([a-z][a-zA-Z0-9])*\$'.	Carte
⚠ Name 'X' must match pattern '^([a-z][a-zA-Z0-9])*\$'.	Carte
⚠ Name 'Y' must match pattern '^([a-z][a-zA-Z0-9])*\$'.	Carte

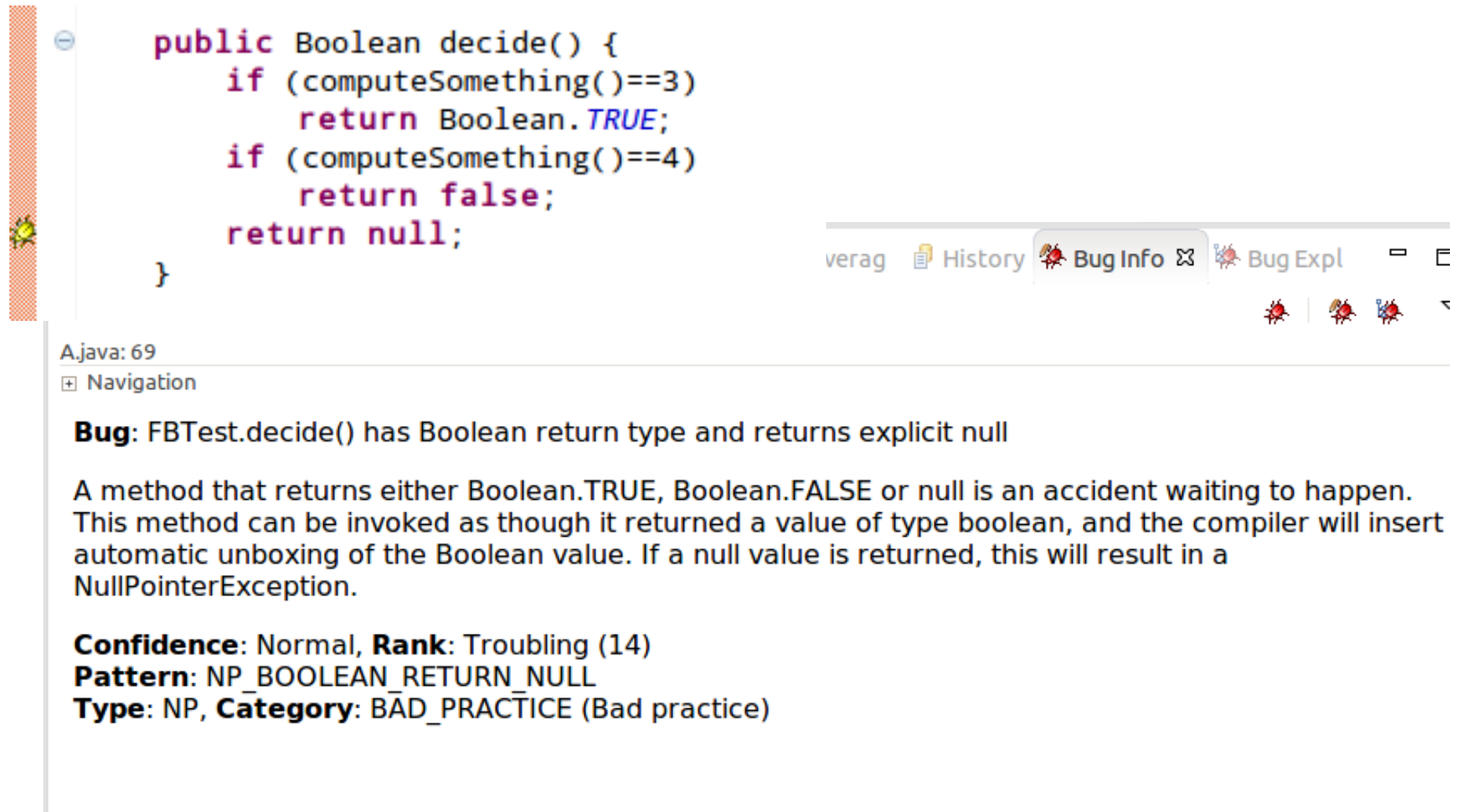
Static Analysis

- Analyzing code without executing it (automated inspection)
- Looks for bug patterns
- Attempts to formally verify specific aspects
- Point out typical bugs or style violations
 - NullPointerExceptions
 - Incorrect API use
 - Forgetting to close a file/connection
 - Concurrency issues
 - And many, many more (over 250 in FindBugs)
- Integrated into IDE or build process
- FindBugs and CheckStyle open source, many commercial products exist

Example FindBugs Bug Patterns

- Correct equals()
- Use of ==
- Closing streams
- Illegal casts
- Null pointer dereference
- Infinite loops
- Encapsulation problems
- Inconsistent synchronization
- Inefficient String use
- Dead store to variable

Bug finding



```
public Boolean decide() {  
    if (computeSomething()==3)  
        return Boolean.TRUE;  
    if (computeSomething()==4)  
        return false;  
    return null;  
}
```

verag History Bug Info Bug Expl

A.java: 69

Navigation

Bug: FBTest.decide() has Boolean return type and returns explicit null

A method that returns either Boolean.TRUE, Boolean.FALSE or null is an accident waiting to happen. This method can be invoked as though it returned a value of type boolean, and the compiler will insert automatic unboxing of the Boolean value. If a null value is returned, this will result in a NullPointerException.

Confidence: Normal, **Rank:** Troubling (14)
Pattern: NP_BOOLEAN_RETURN_NULL
Type: NP, **Category:** BAD_PRACTICE (Bad practice)

Can you find the bug?

```
if (listeners == null)
    listeners.remove(listener) ;
```

JDK1.6.0, b105, sun.awt.x11.XMSelection

Wrong boolean operator

```
if (listeners != null)  
    listeners.remove(listener) ;
```

JDK1.6.0, b105, sun.awt.x11.XMSelection

Can you find the bug?

```
public String sendMessage (User user, String body, Date time) {  
    return sendMessage(user, body, null);  
}  
  
public String sendMessage (User user, String body, Date time,  
    List attachments) {  
    String xml = buildXML (body, attachments);  
    String response = sendMessage(user, xml);  
    return response;  
}
```

Infinite recursive loop

```
public String sendMessage (User user, String body, Date time) {  
    return sendMessage(user, body, null);  
}  
  
public String sendMessage (User user, String body, Date time,  
    List attachments) {  
    String xml = buildXML (body, attachments);  
    String response = sendMessage(user, xml);  
    return response;  
}
```

Can you find the bug?

```
String b = "bob";  
b.replace('b', 'p');  
if(b.equals("pop")){...}
```

Method ignores return value

```
String b = "bob";  
b = b.replace('b', 'p');  
if(b.equals("pop")){...}
```

What does this print?

```
Integer one = 1;
Long addressTypeCode = 1L;

if (addressTypeCode.equals(one)) {
    System.out.println("equals");
} else {
    System.out.println("not equals");
}
```


What does this print?

```
Integer one = 1;
Long addressTypeCode = 1L;

if (addressTypeCode.equals(one)) {
    System.out.println("equals");
} else {
    System.out.println("not equals");
}
```

```
Detector foo = null;  
foo.execute();
```

ASIDE: FINDBUGS NULL POINTER ANALYSIS

FindBugs

- Works on “.class” files containing **bytecode**
 - Recall: Java source code compiled to bytecode; JVM executes bytecode
- Processing using different **detectors**:
 - Independent of each other
 - May share some resources (e.g., control flow graph, dataflow analysis)
 - GOAL: **Low false positives**
 - Each detector is driven by a set of **heuristics**
- Output: bug pattern code, source line number, descriptive message (severity)



Null pointer dereferencing

- Finding some null pointer dereferences require sophisticated analysis:
 - Analyzing across method calls, modeling contents of heap objects
- In practice many examples of **obvious** null pointer dereferences:
 - Values which are always null
 - Values which are null on some control path
- How to design an analysis to find obvious null pointer dereferences?
 - Idea: Look for places where values are used in a suspicious way

Simple Analysis

```
Detector foo = null;  
foo.execute();
```



Dereferencing
Null

```
Detector foo = new Detector(...);  
foo.execute();
```

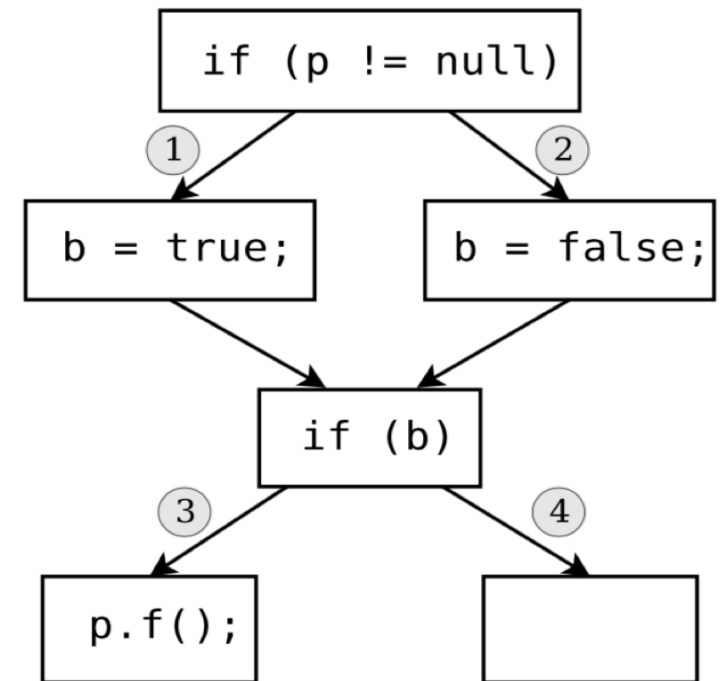


Dereferencing
NonNull

If only it were that simple...

- Infeasible paths (false positives)

```
boolean b;  
if (p != null)  
    b = true;  
else  
    b = false;  
if (b)  
    p.f();
```



- Is a method's parameter null?

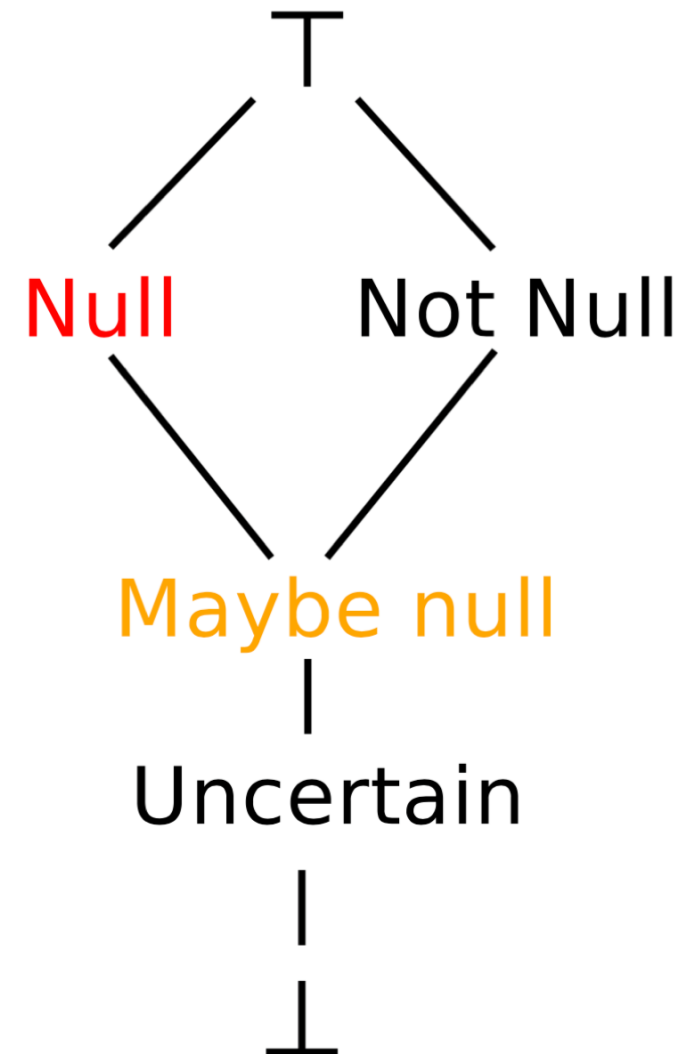
```
void foo(Object obj) {  
    int x = obj.hashCode();  
    ...  
}
```

Dataflow analysis

- At each point in a method, keep track of dataflow facts
 - E.g., which local variables and stack locations might contain null
- Symbolically execute the method:
 - Model instructions
 - Model control flow
 - Until a fixed point solution is reached

Dataflow values

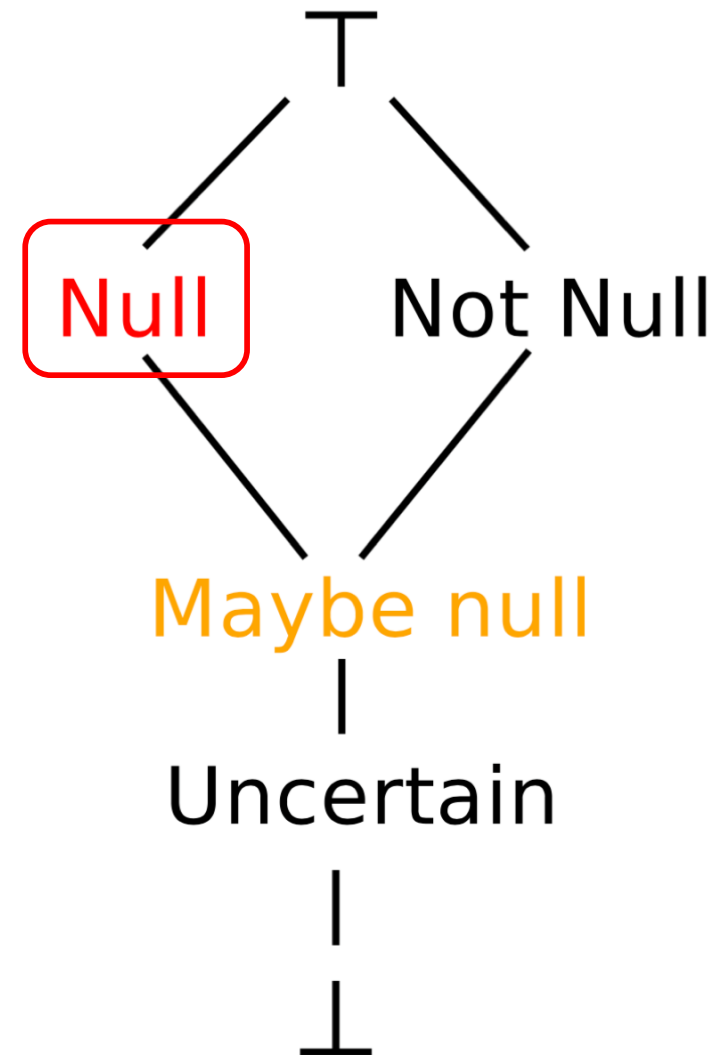
- Model values of local variables and stack operands using lattice of symbolic values
- When two control paths merge, use *meet* operator to combine values:



Dataflow values

- Model values of local variables and stack operands using lattice of symbolic values
- When two control paths merge, use *meet* operator to combine values:

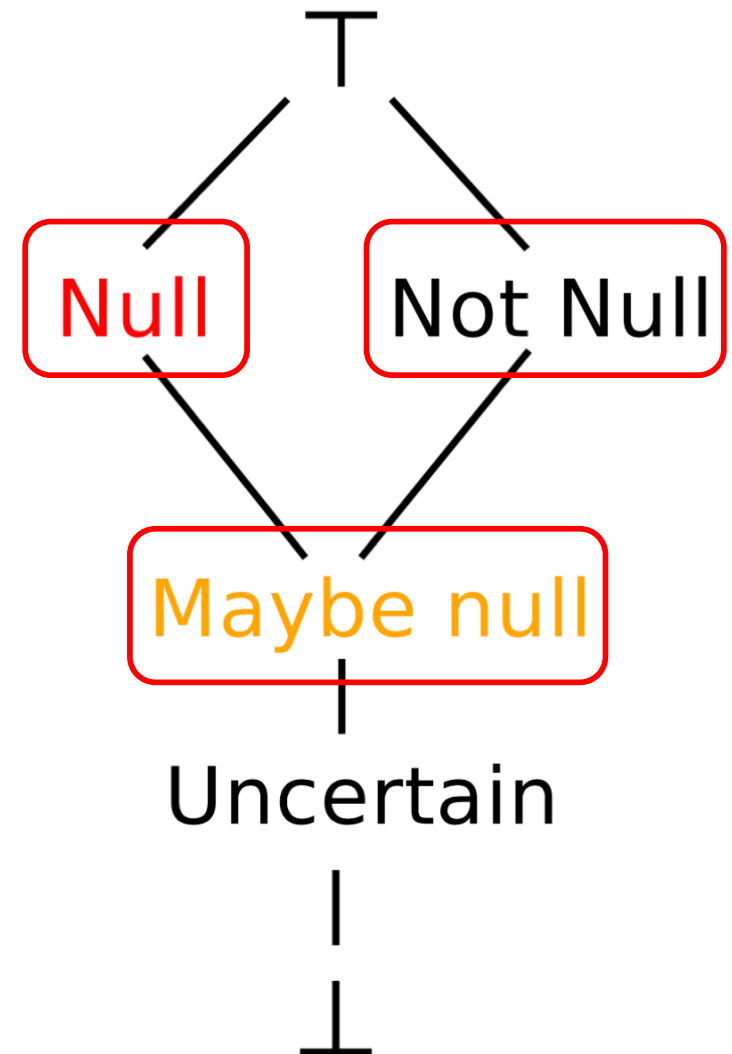
$$\text{Null} \diamond \text{Null} = \text{Null}$$



Dataflow values

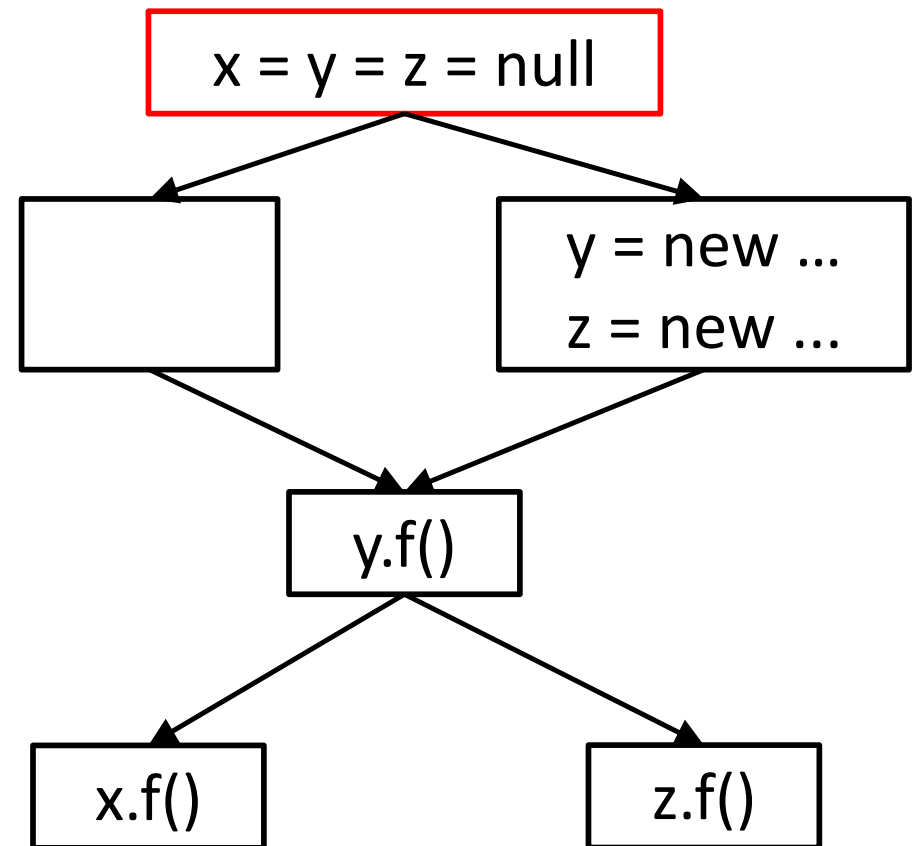
- Model values of local variables and stack operands using lattice of symbolic values
- When two control paths merge, use *meet* operator to combine values:

Null \diamond Not Null = Maybe Null



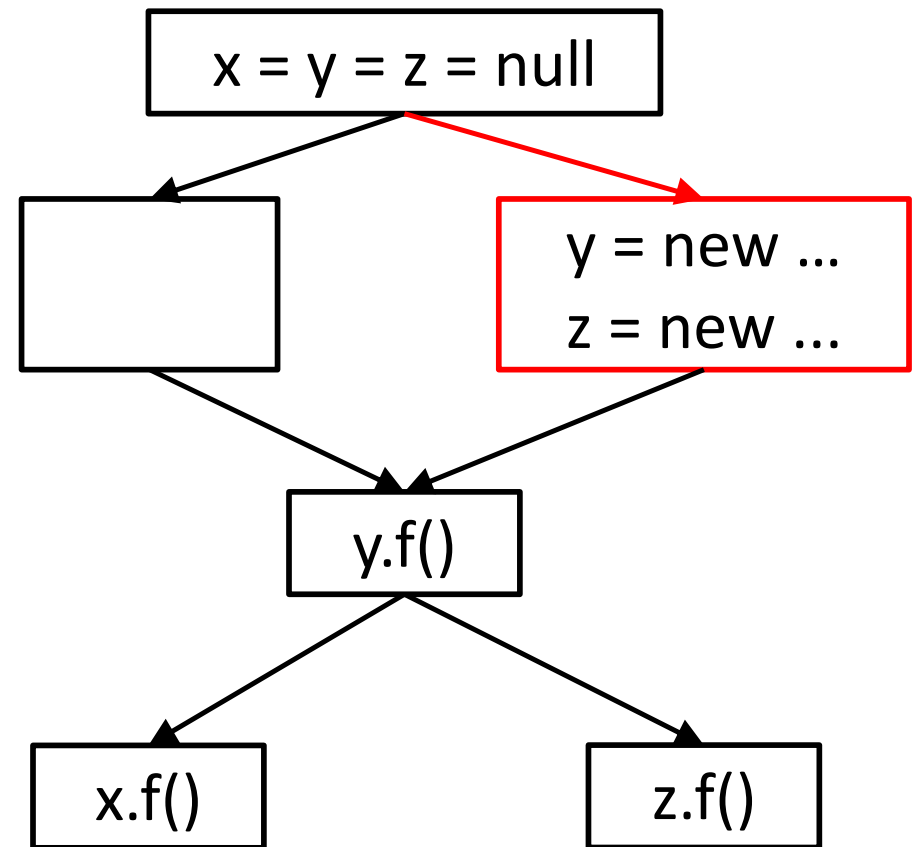
Null-pointer dataflow example

```
x = y = z = null;  
if (cond) {  
    y = new ...;  
    z = new ...;  
}  
y.f();  
if (cond2)  
    x.f();  
else  
    z.f();
```



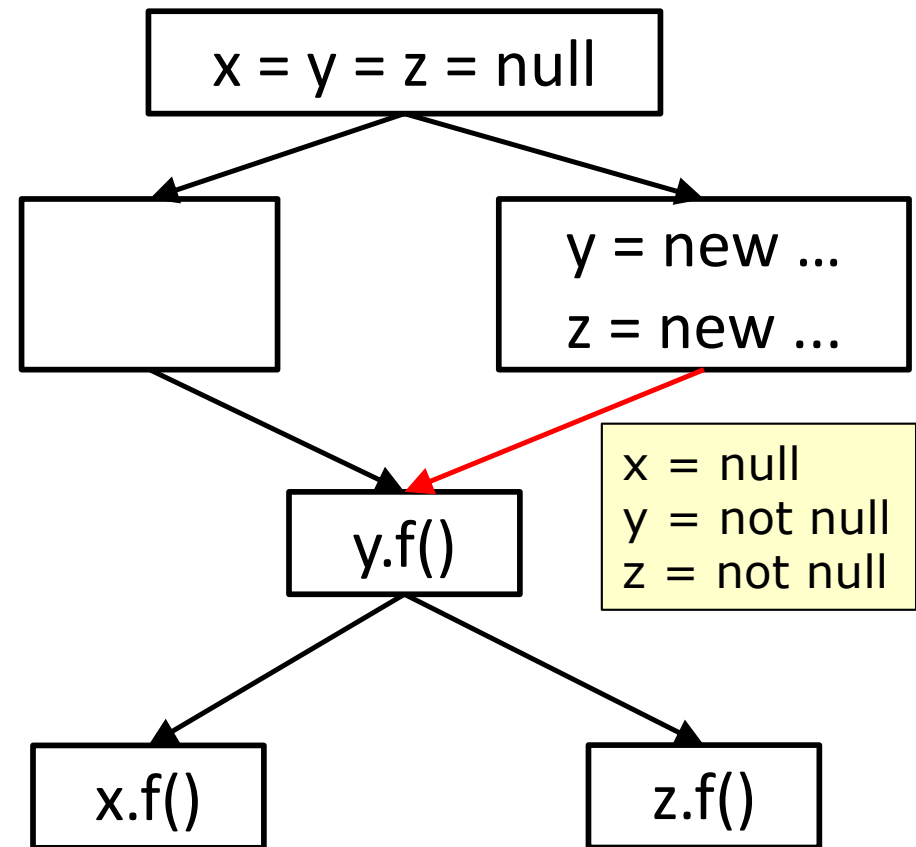
Null-pointer dataflow example

```
x = y = z = null;  
if (cond) {  
    y = new ...;  
    z = new ...;  
}  
y.f();  
if (cond2)  
    x.f();  
else  
    z.f();
```



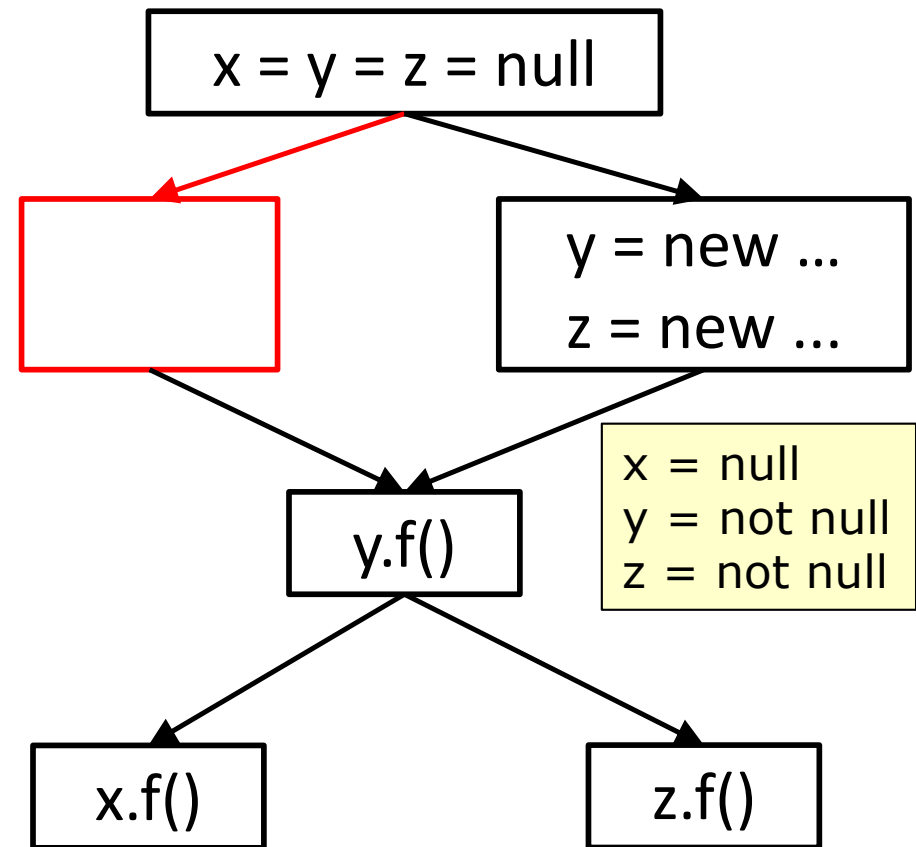
Null-pointer dataflow example

```
x = y = z = null;  
if (cond) {  
    y = new ...;  
    z = new ...;  
}  
y.f();  
if (cond2)  
    x.f();  
else  
    z.f();
```



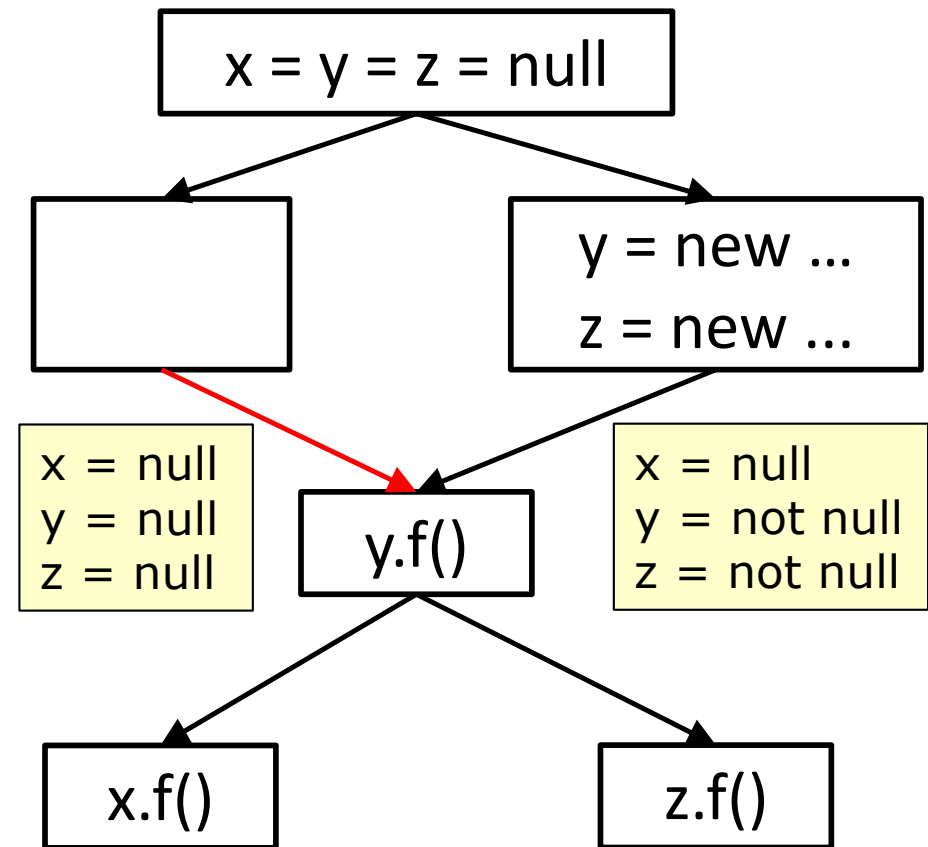
Null-pointer dataflow example

```
x = y = z = null;  
if (cond) {  
    y = new ...;  
    z = new ...;  
}  
y.f();  
if (cond2)  
    x.f();  
else  
    z.f();
```



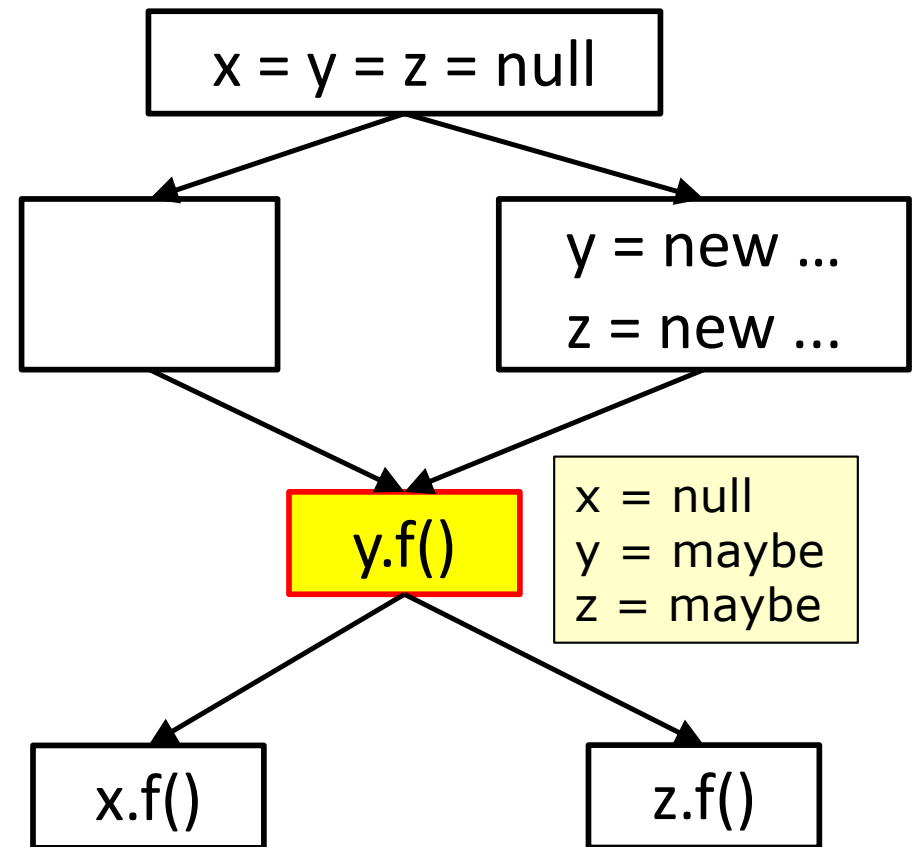
Null-pointer dataflow example

```
x = y = z = null;
if (cond) {
    y = new ...;
    z = new ...;
}
y.f();
if (cond2)
    x.f();
else
    z.f();
```



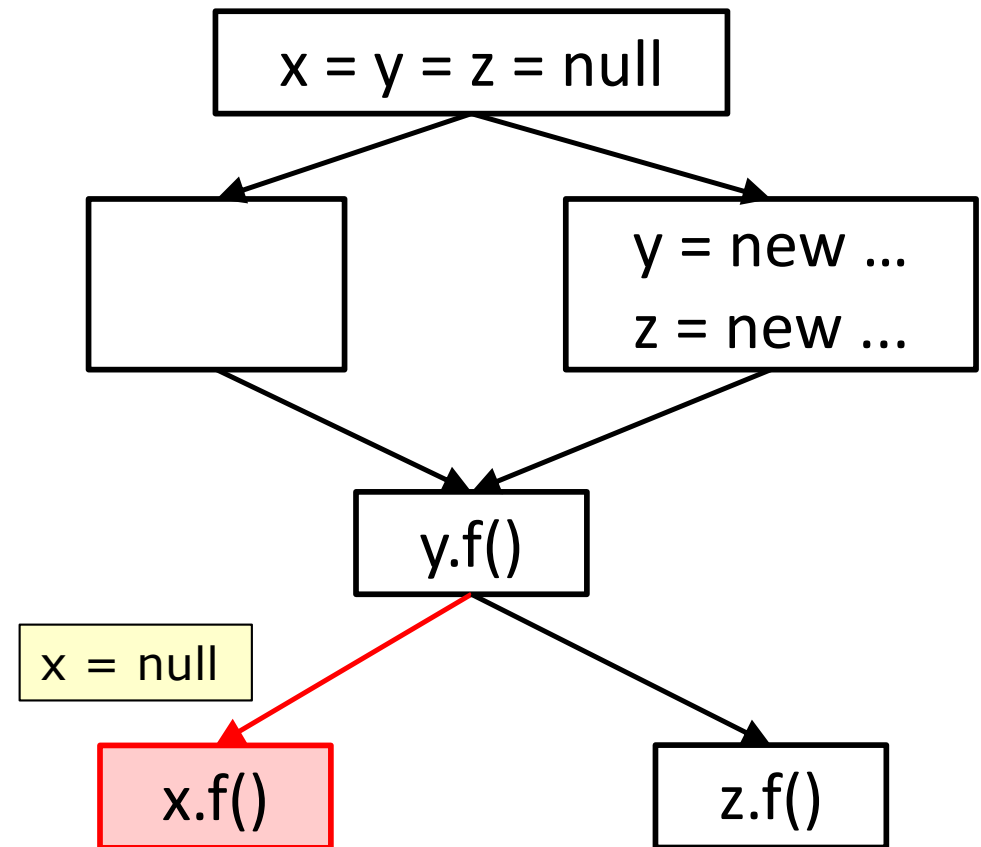
Null-pointer dataflow example

```
x = y = z = null;  
if (cond) {  
    y = new ...;  
    z = new ...;  
}  
y.f();  
if (cond2)  
    x.f();  
else  
    z.f();
```



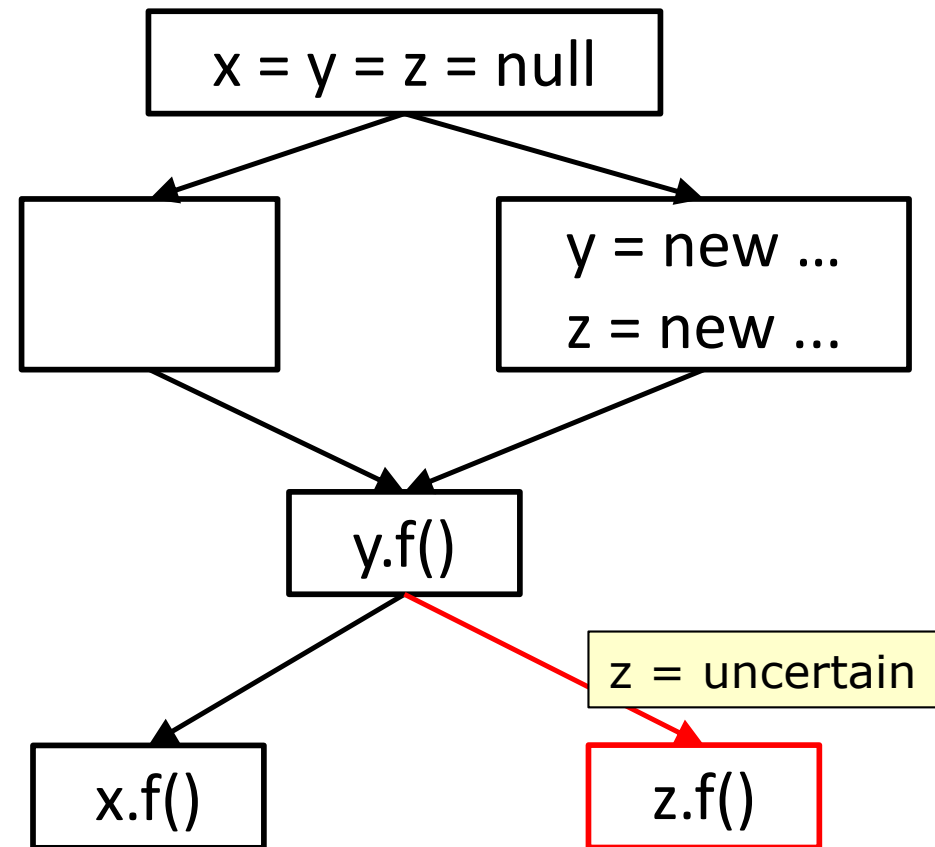
Null-pointer dataflow example

```
x = y = z = null;  
if (cond) {  
    y = new ...;  
    z = new ...;  
}  
y.f();  
if (cond2)  
    x.f();  
else  
    z.f();
```



Null-pointer dataflow example

```
x = y = z = null;  
if (cond) {  
    y = new ...;  
    z = new ...;  
}  
y.f();  
if (cond2)  
    x.f();  
else  
    z.f();
```



COMPARING QUALITY ASSURANCE STRATEGIES

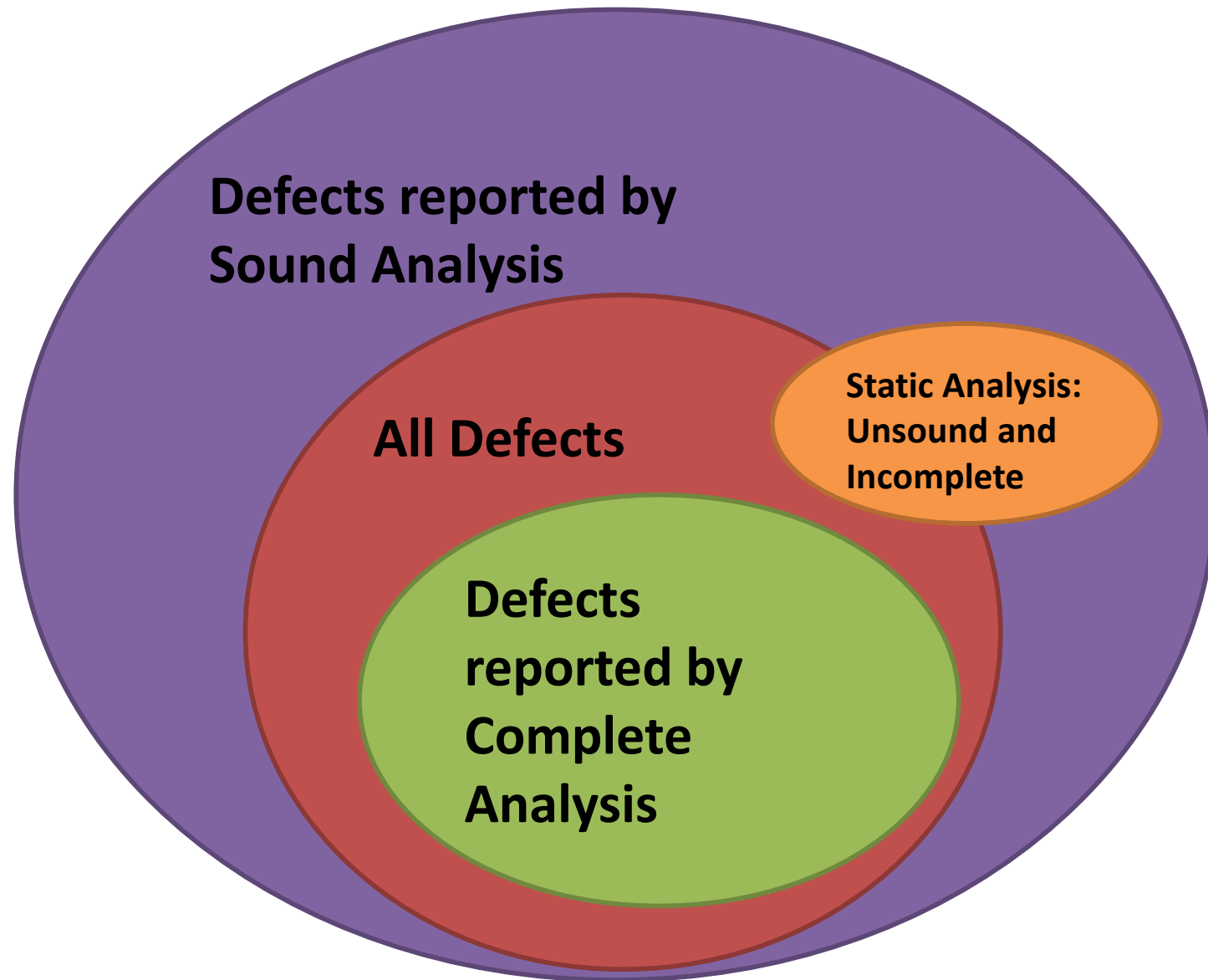
	Error exists	No error exists
Error Reported	True positive (correct analysis result)	False positive (annoying noise)
No Error Reported	False negative (false confidence)	True negative (correct analysis result)

Sound Analysis:
 reports all defects
 → no false negatives
 typically overapproximated

Complete Analysis:
 every reported defect is an actual defect
 → no false positives
 typically underapproximated

Check your understanding

- What is a trivial way to implement:
 - a sound analysis?
 - a complete analysis?



	Error exists	No error exists
Error Reported	True positive (correct analysis result)	False positive (annoying noise)
No Error Reported	False negative (false confidence)	True negative (correct analysis result)

Sound Analysis:
 reports all defects
 → no false negatives
 typically overapproximated

Complete Analysis:
 every reported defect is an actual defect
 → no false positives
 typically underapproximated

How does testing relate? And formal verification?

The Bad News: Rice's Theorem

"Any nontrivial property about the language recognized by a Turing machine is undecidable."

Henry Gordon Rice, 1953

- Every static analysis is necessarily incomplete or unsound or undecidable (or multiple of these)
- Each approach has different tradeoffs

Soundness / Completeness / Performance Tradeoffs

- Type checking does catch a specific class of problems (sound), but does not find all problems
- Compiler optimizations must err on the safe side (only perform optimizations when sure it's correct; -> complete)
- Many practical bug-finding tools analyses are unsound and incomplete
 - Catch typical problems
 - May report warnings even for correct code
 - May not detect all problems
- Overwhelming amounts of false negatives make analysis useless
- Not all "bugs" need to be fixed

Testing, Static Analysis, and Proofs

- Testing
 - Observable properties
 - Verify program for one execution
 - Manual development with automated regression
 - Most practical approach now
 - Does not find all problems (unsound)
- Static Analysis
 - Analysis of all possible executions
 - Specific issues only with conservative approx. and bug patterns
 - Tools available, useful for bug finding
 - Automated, but unsound and/or incomplete
- Proofs (Formal Verification)
 - Any program property
 - Verify program for all executions
 - Manual development with automated proof checkers
 - Practical for small programs, may scale up in the future
 - Sound and complete, but not automatically decidable

What strategy to use in your project?

Take-Home Messages

- There are many forms of quality assurance
- Testing should be integrated into development
 - possibly even test first
- Various coverage metrics can more or less approximate test suite quality
- Static analysis tools can detect certain patterns of problems
- Soundness and completeness to characterize analyses