Principles of Software Construction: Objects, Design, and Concurrency

Designing (sub-) systems

Incremental improvements

Charlie Garrod **Michael Hilton**

School of Computer Science

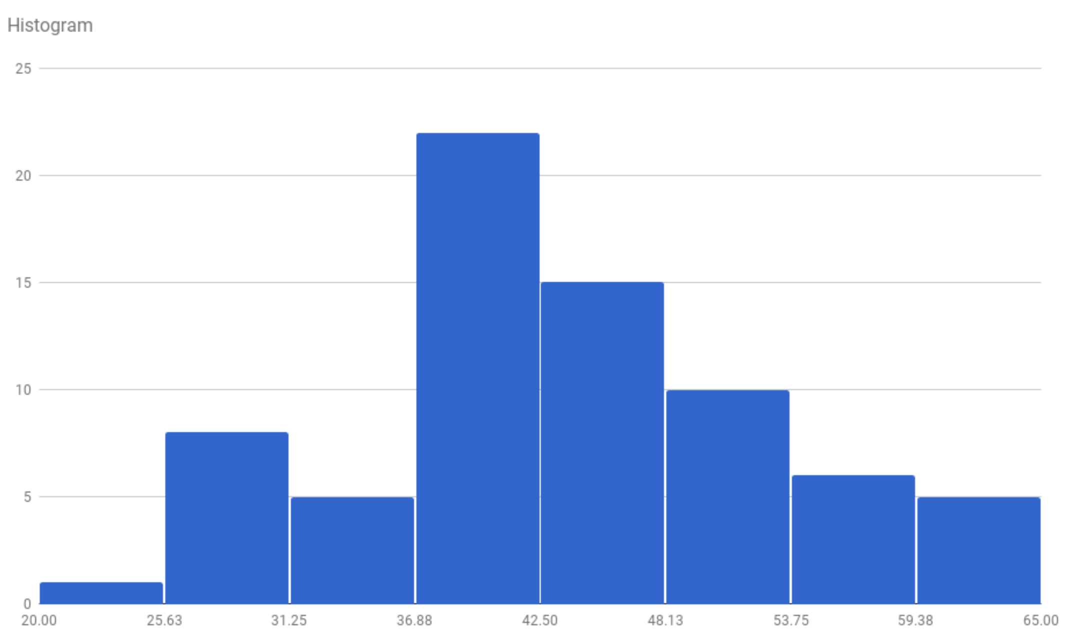
institute for SOFTWARE RESEARCH

Administriva

- HW4 Part A due Oct 5th
  - Mandatory design review meeting
- Final Friday, December 15, 2017 05:30-08:30 p.m.

Exam results Midterm 1

- Mean
  - 43.73/72
  - NOTE: This course does not have a fixed letter grade policy; i.e. the final letter grades will **not** be A=90-100%, B=80-90%, etc.
- Standard Deviation
  - 9.26



Review Strategy Pattern

- Problem: Clients need different variants of an algorithm
- Solution: Create an interface for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context
  - Introduces an extra interface and many classes:
    - Code can be harder to understand
    - Lots of overhead if the strategies are simple

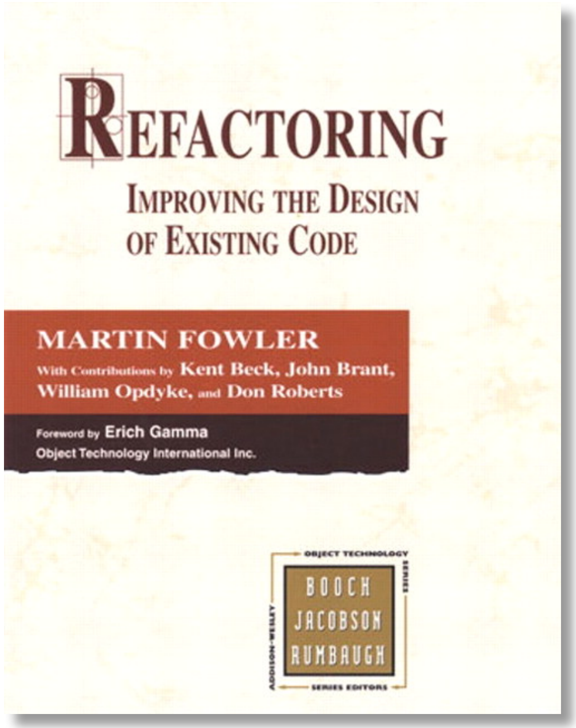
Design Problem

```
public class EgyptianTranslator {
    int n;
    EgyptianTranslator(int n) {
        this.n = n;
        …
    }
    public String translate() {
        …
    }
}
```

CODE SMELLS

Code Smells

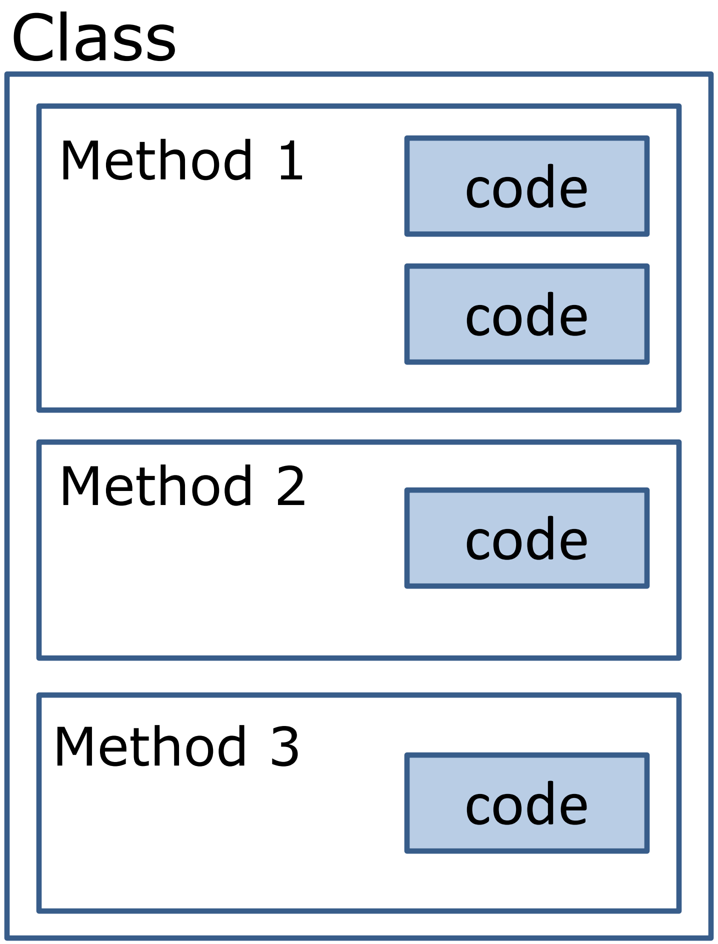
- A *code smell* is a hint that something has gone wrong somewhere in your code.
- A smell is *sniffable,* or something that is quick to spot.
- A smell doesn't *always* indicate a problem



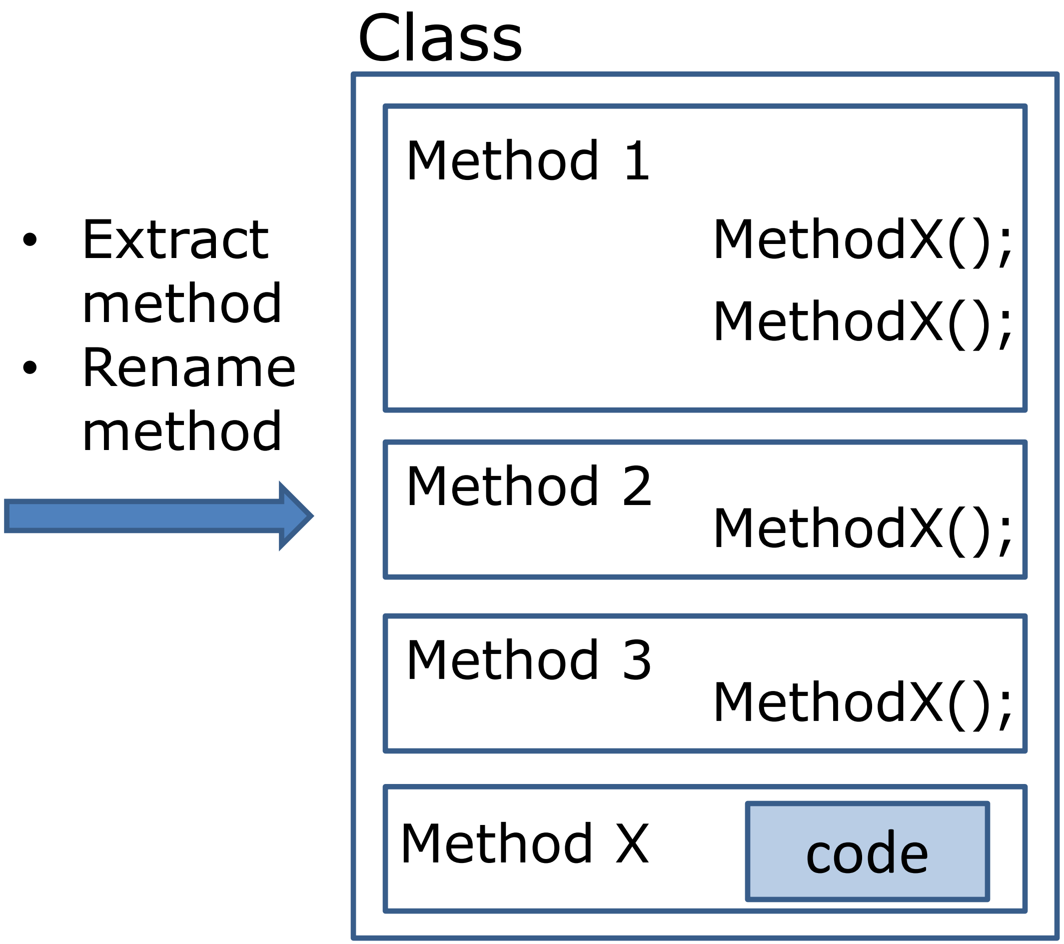
Bad Smells: Classification

- Most Common: **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

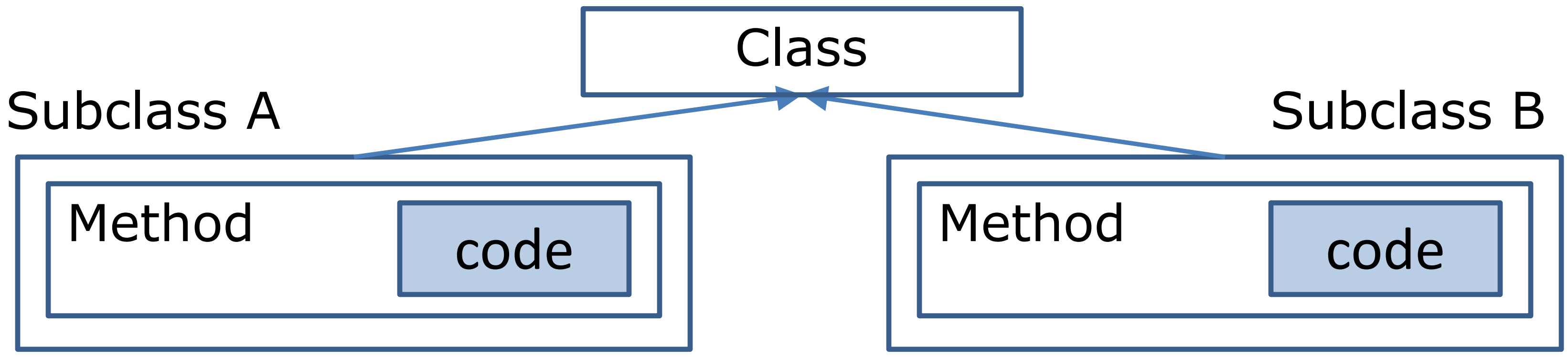
Code duplication (1)



- Extract method
- Rename method



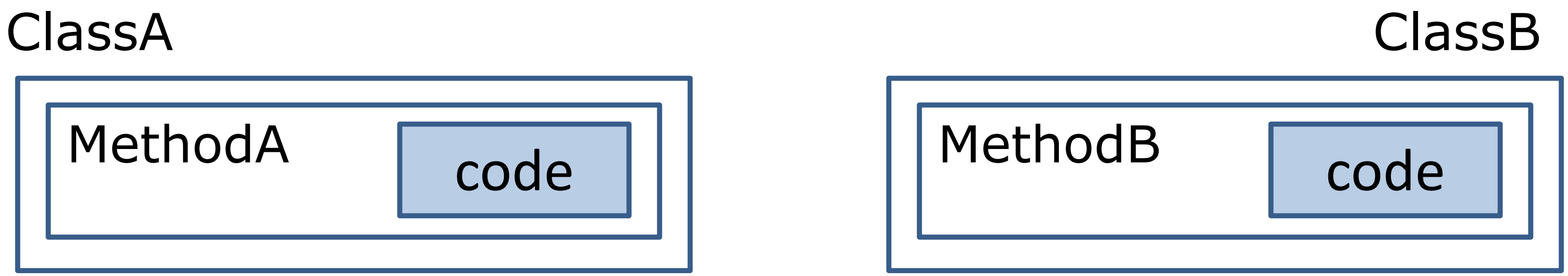
Code duplication (2)



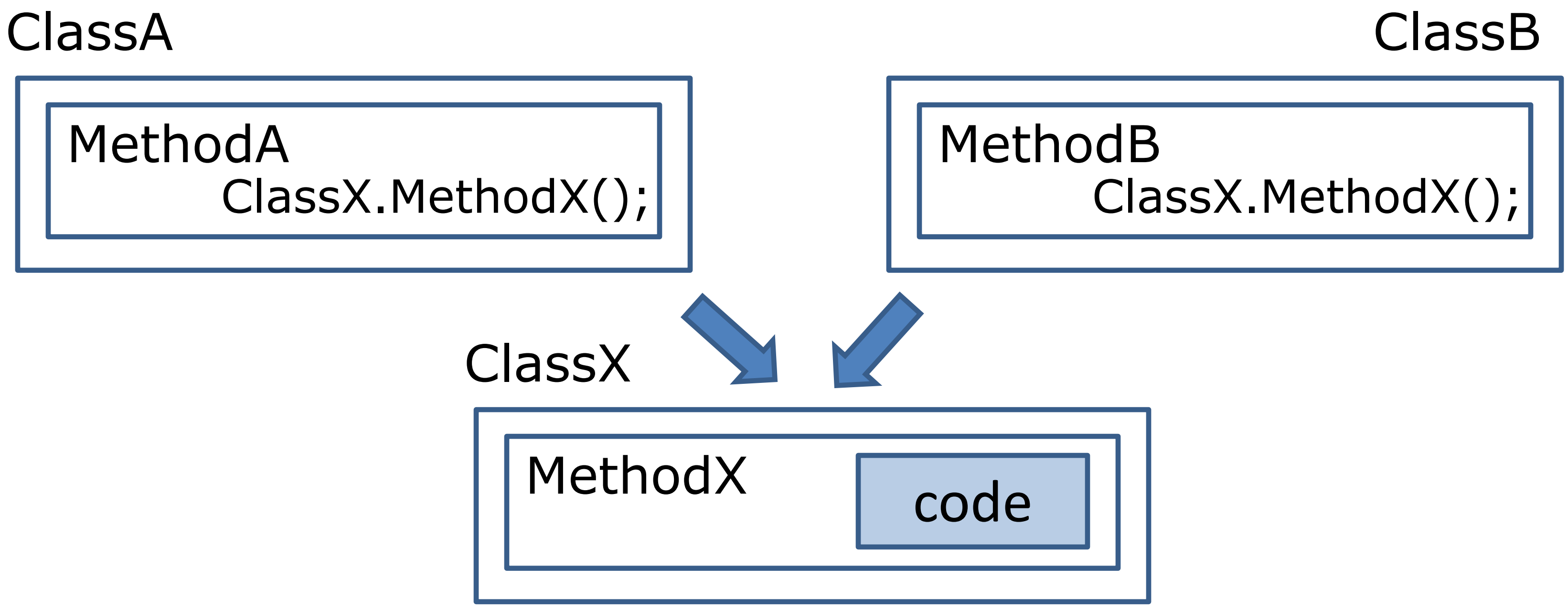
Same expression in two sibling classes:

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

Code duplication (3)



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 () {
                                        }
                                    }
                                    if () {
                                        if () {
                                            if () {
                                                for () {
                                                }
                                            }
                                        }
                                    } else {
                                    }
                                }
                            } else {
                            }
                        }
                    }
                }
            }
```

- 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

institute for SOFTWARE RESEARCH

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

institute for SOFTWARE RESEARCH

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

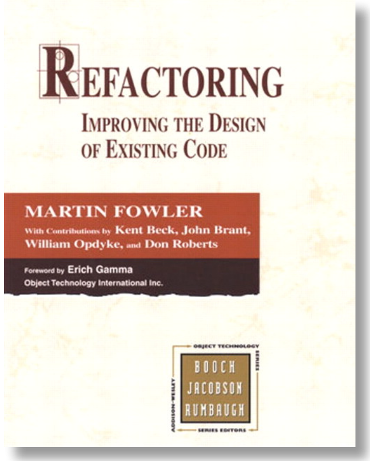
```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = getOutstanding();
    printBanner();
    printDetails(outstanding);
}
void printBanner(){…}
void printDetails(outstanding){…}

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

ANTI-PATTERNS

Anti-patterns

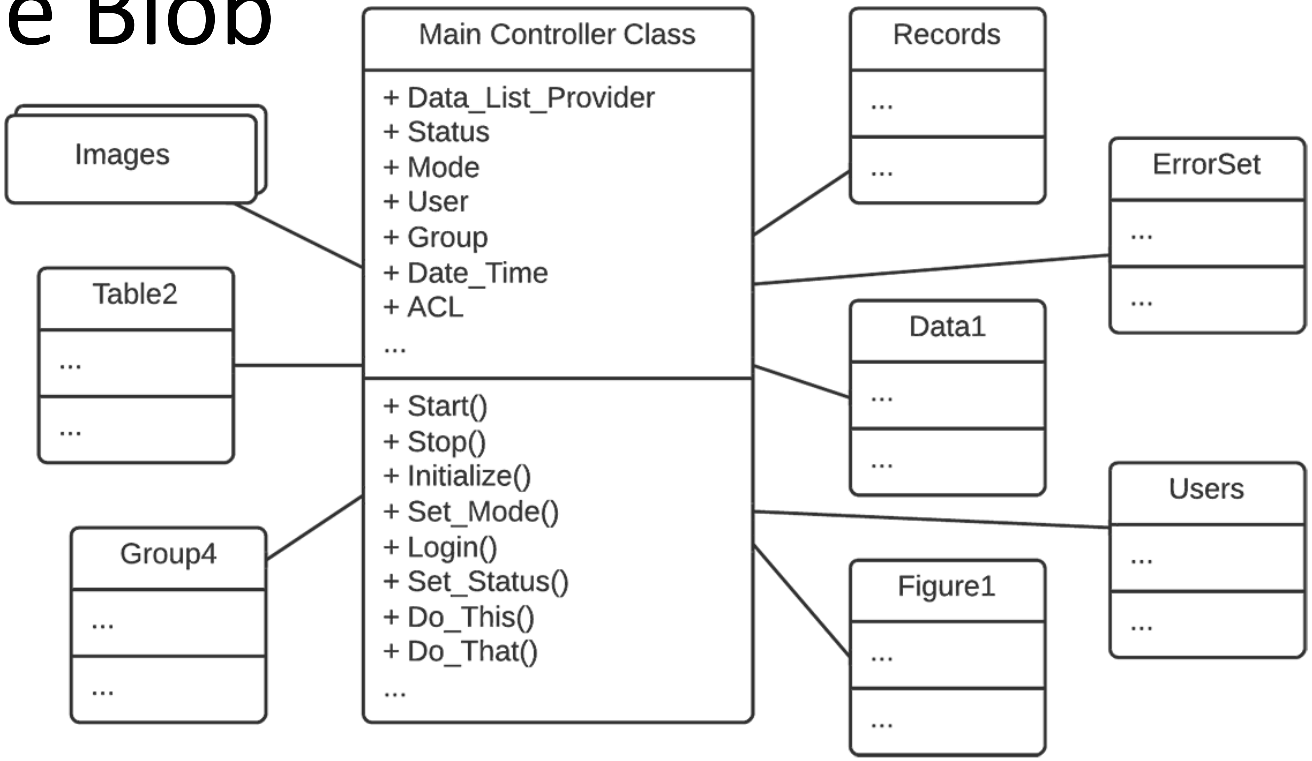
- “Anti”-pattern
- Patterns of things you should NOT do
- Often have memorable names.

Common anti-patterns

- Spaghetti code

Common anti-patterns

- Spaghetti code
- The Blob



Common anti-patterns

- Spaghetti code
- The Blob
- Golden Hammer

Common anti-patterns

- Spaghetti code
- The Blob
- Golden Hammer
- Lava Flow

Common anti-patterns

- Spaghetti code
- The Blob
- Golden Hammer
- Lava Flow
- Swiss Army Knife

EVALUATING 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?

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

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

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 15-313)

TEST COVERAGE

How much testing?

- You generally cannot test all inputs
  - too many, usually infinite
- But when it works, exhaustive testing is best!

- When to stop testing?
  - in practice, when you run out of money

What makes a good test suite?

- Provides high confidence that code is correct
- Short, clear, and non-repetitious
  - More difficult for test suites than regular code
  - Realistically, test suites will look worse
- Can be fun to write if approached in this spirit

Blackbox: Random Inputs
Next best thing to exhaustive testing

- Also know as *fuzz testing, torture testing*
- Try “random” inputs, as many as you can
  - Choose inputs to tickle interesting cases
  - Knowledge of implementation helps here
- Seed random number generator so tests repeatable
- Successful in some domains (parsers, network issues, …)
  - But, many tests execute similar paths
  - But, often finds only superficial errors

Blackbox: Covering Specifications

- Looking at specifications, not code:

- Test representative case
- Test boundary condition
- Test exception conditions
- (Test invalid case)

Textual Specification

public int read(byte[] b, int off, int len) throws IOException

- Reads up to len bytes of data from the input stream into an array of bytes. An attempt is made to read as many as len bytes, but a smaller number may be read. The number of bytes actually read is returned as an integer. This method blocks until input data is available, end of file is detected, or an exception is thrown.
- If len is zero, then no bytes are read and 0 is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value -1 is returned; otherwise, at least one byte is read and stored into b.
- The first byte read is stored into element b[off], the next one into b[off+1], and so on. The number of bytes read is, at most, equal to len. Let *k* be the number of bytes actually read; these bytes will be stored in elements b[off] through b[off+*k*-1], leaving elements b[off+*k*] through b[off+len-1] unaffected.
- In every case, elements b[0] through b[off] and elements b[off+len] through b[b.length-1] are unaffected.

- Throws:
  - IOException - If the first byte cannot be read for any reason other than end of file, or if the input stream has been closed, or if some other I/O error occurs.
  - NullPointerException - If b is null.
  - IndexOutOfBoundsException - If off is negative, len is negative, or len is greater than b.length - off

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 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?

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;

    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?

Does this return statement ever get reached?

Code coverage metrics

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

Method Coverage

- Trying to execute each method as part of at least one test

```java
    }
    public boolean equals(Object anObject) {
        if (isZero())
            if (anObject instanceof IMoney)
                return ((IMoney)anObject).isZero();
        if (anObject instanceof Money) {
            Money aMoney= (Money)anObject;
            return aMoney.currency().equals(currency())
                            && amount() == aMoney.amount();
        }
        return false;
    }
    public int hashCode() {
```

- Does this guarantee correctness?

Statement Coverage

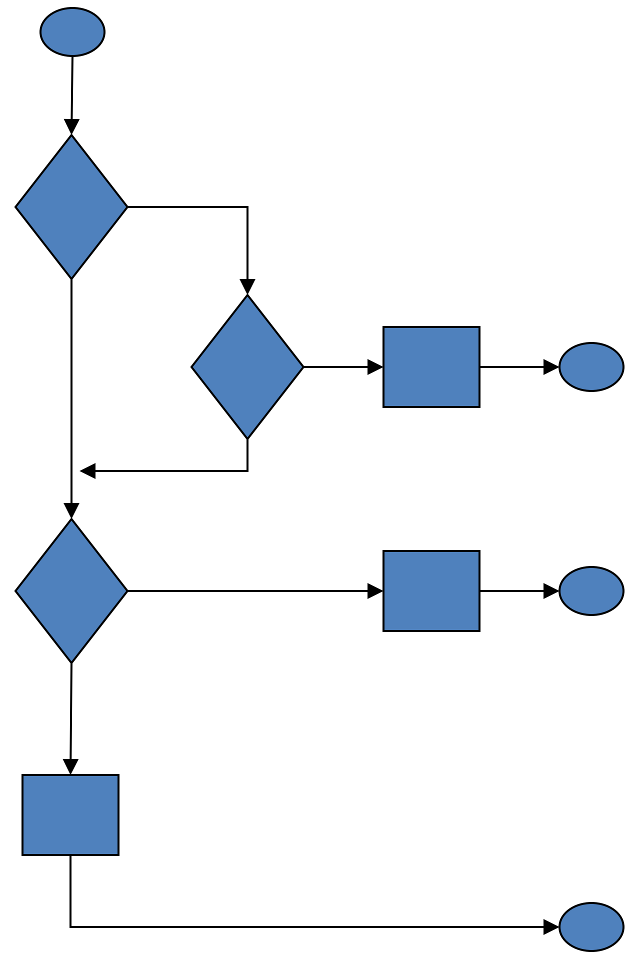
- Trying to test all parts of the implementation
- Execute every statement in at least one test

```java
    public boolean equals(Object anObject) {
        if (isZero())
            if (anObject instanceof IMoney)
                return ((IMoney)anObject).isZero();
        if (anObject instanceof Money) {
            Money aMoney= (Money)anObject;
            return aMoney.currency().equals(currency())
                                && amount() == aMoney.amount();
        }
        return false;
    }
```

- Does this guarantee correctness?

Structure of Code Fragment to Test

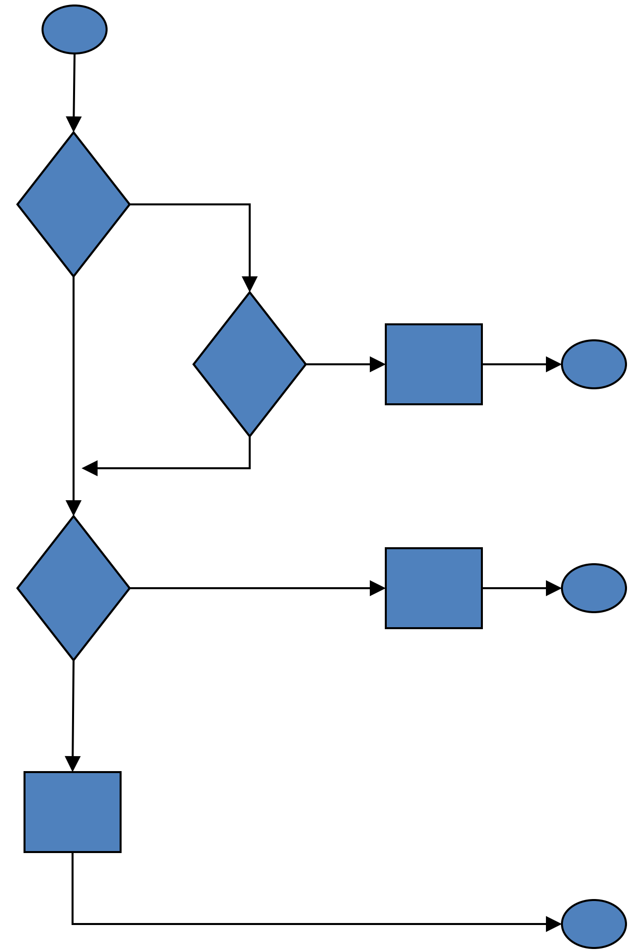
```java
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
                        && amount() == aMoney.amount();
    }
    return false;
}
```



Flow chart diagram for
`junit.samples.money.Money.equals`

Statement Coverage

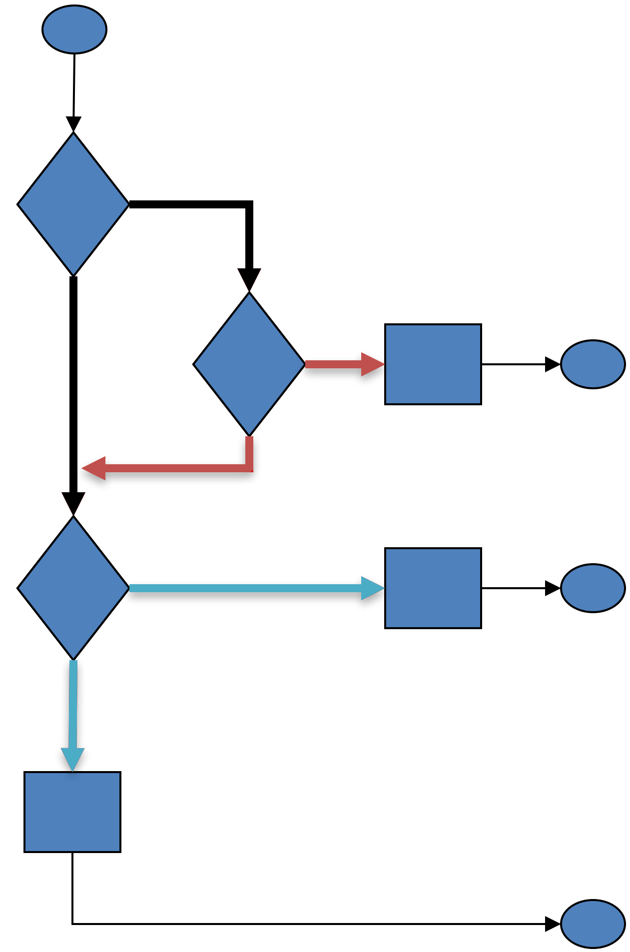
- Statement coverage
  - What portion of program statements (nodes) are touched by test cases
- Advantages
  - Test suite size linear in size of code
  - Coverage easily assessed
- Issues
  - Dead code is not reached
  - May require some sophistication to select input sets
  - Fault-tolerant error-handling code may be difficult to “touch”
  - Metric: Could create incentive to remove error handlers!



```java
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
                          && amount() == aMoney.amount();
    }
    return false;
```

Branch Coverage

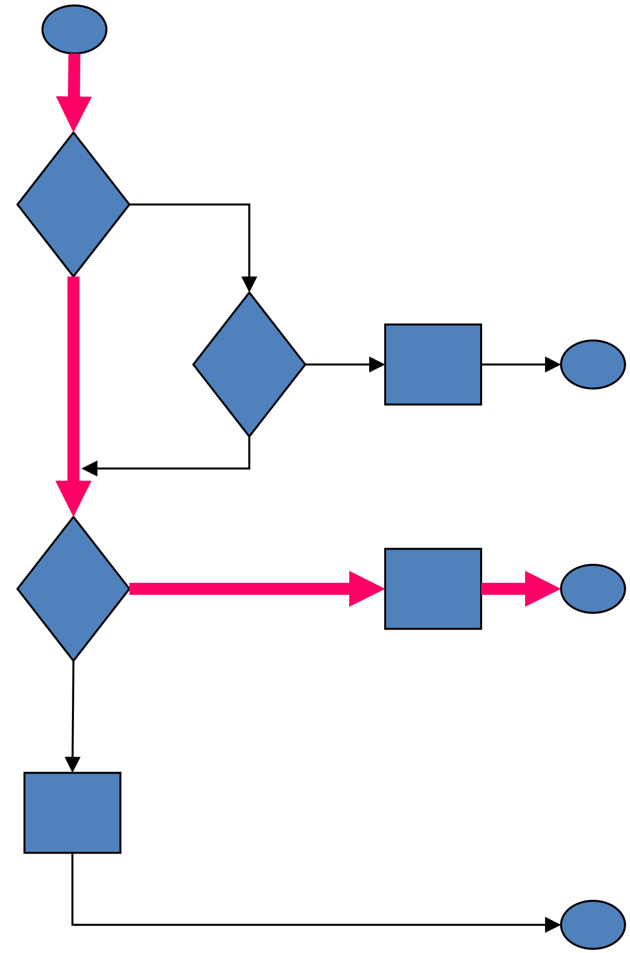
- Branch coverage
  - What portion of condition branches are covered by test cases?
  - Or: What portion of relational expressions and values are covered by test cases?
    - Condition testing (Tai)
  - Multicondition coverage – all boolean combinations of tests are covered
- Advantages
  - Test suite size and content derived from structure of boolean expressions
  - Coverage easily assessed
- Issues
  - Dead code is not reached
  - Fault-tolerant error-handling code may be difficult to “touch”



```java
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
                            && amount() == aMoney.amount();
    }
    return false;
```

Path Coverage

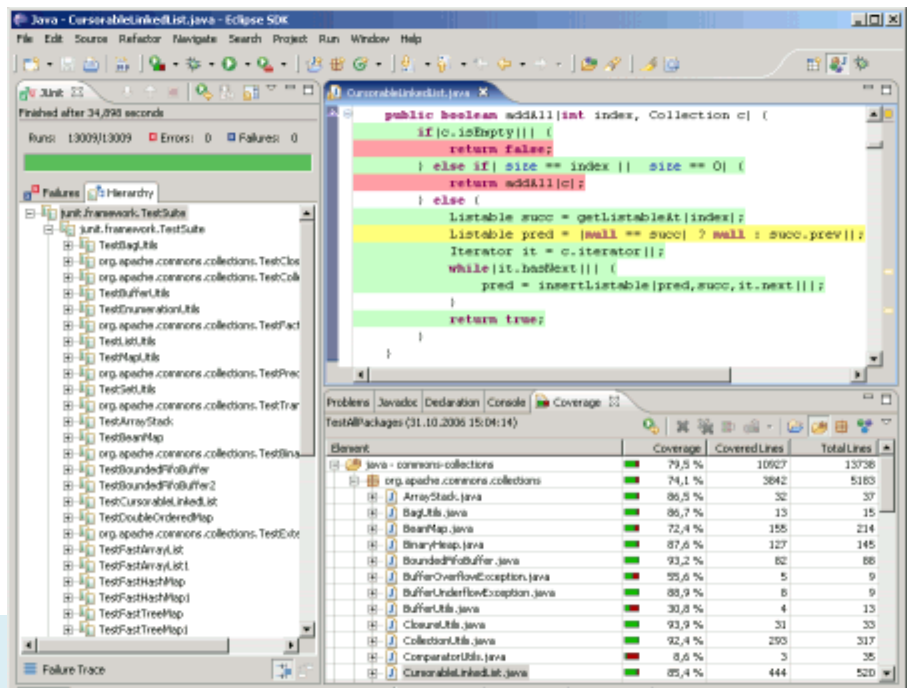
- Path coverage
  - What portion of all possible paths through the program are covered by tests?
  - Loop testing: Consider representative and edge cases:
    - Zero, one, two iterations
    - If there is a bound n: n-1, n, n+1 iterations
    - Nested loops/conditionals from inside out
- Advantages
  - Better coverage of logical flows
- Disadvantages
  - Infinite number of paths
  - Not all paths are possible, or necessary
    - What are the significant paths?
  - Combinatorial explosion in cases unless careful choices are made
    - E.g., sequence of n if tests can yield up to 2^n possible paths
  - Assumption that program structure is basically sound



```
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
                            && amount() == aMoney.amount();
    }
    return false;
```

Test Coverage Tooling

- Coverage assessment tools
  - Track execution of code by test cases
- Count visits to statements
  - Develop reports with respect to specific coverage criteria
  - Instruction coverage, line coverage, branch coverage
- Example: Cobertura and EclEmma for JUnit tests



Packages

[All](#)
[net.sourceforge.cobertura.ant](#)
[net.sourceforge.cobertura.check](#)
[net.sourceforge.cobertura.coveragedat](#)
[net.sourceforge.cobertura.instrument](#)
[net.sourceforge.cobertura.merge](#)
[net.sourceforge.cobertura.reporting](#)
[net.sourceforge.cobertura.reporting.htr](#)
[net.sourceforge.cobertura.reporting.htr](#)
[net.sourceforge.cobertura.reporting.xm](#)
[net.sourceforge.cobertura.util](#)

All Packages

Classes

- AntUtil *(88%)*
- Archive *(100%)*
- ArchiveUtil *(80%)*
- BranchCoverageData *(N/A)*
- CheckTask *(0%)*
- ClassData *(N/A)*
- ClassInstrumenter *(94%)*
- ClassPattern *(100%)*
- CoberturaFile *(73%)*
- CommandLineBuilder *(96%)*
- CommonMatchingTask *(88%)*
- ComplexityCalculator *(100%)*
- ConfigurationUtil *(50%)*
- CopyFiles *(87%)*
- CoverageData *(N/A)*
- CoverageDataContainer *(N/A)*
- CoverageDataFileHandler *(N/A)*
- CoverageRate *(0%)*
- ExcludeClasses *(100%)*
- FileFinder *(96%)*
- FileLocker *(0%)*
- FirstPassMethodInstrumenter *(100%)*
- HTMLReport *(94%)*
- HasBeenInstrumented *(N/A)*
- Header *(80%)*

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Compl
All Packages	55	75%	1625/2179	64%	472/738	
net.sourceforge.cobertura.ant	11	52%	170/330	43%	40/94	
net.sourceforge.cobertura.check	3	0%	0/150	0%	0/76	
net.sourceforge.cobertura.coveragedata	13	N/A	N/A	N/A	N/A	
net.sourceforge.cobertura.instrument	10	90%	460/510	75%	123/164	
net.sourceforge.cobertura.merge	1	86%	30/35	88%	14/16	
net.sourceforge.cobertura.reporting	3	87%	116/134	80%	43/54	
net.sourceforge.cobertura.reporting.html	4	91%	475/523	77%	156/202	
net.sourceforge.cobertura.reporting.html.files	1	87%	39/45	62%	5/8	
net.sourceforge.cobertura.reporting.xml	1	100%	155/155	95%	21/22	
net.sourceforge.cobertura.util	9	60%	175/291	69%	70/102	
someotherpackage	1	83%	5/6	N/A	N/A	

Report generated by Cobertura 1.9 on 6/9/07 12:37 AM.

Check your understanding

- Write test cases to achieve 100% line coverage but not 100% branch coverage

```
int foo(int a, int b) {
    if (a == b)
        a = a * 2;
    if (a + b > 10)
        return a - b;
    return a + b;
}
```

Check your understanding

- Write test cases to achieve 100% line coverage <u>and also</u> 100% branch coverage

```
int foo(int a, int b) {
    if (a == b)
        a = a * 2;
    if (a + b > 10)
        return a - b;
    return a + b;
}
```

Check your understanding

- Write test cases to achieve 100% line coverage <u>and</u> 100% branch coverage <u>and</u> 100% path coverage

```
int foo(int a, int b) {
    if (a == b)
        a = a * 2;
    if (a + b > 10)
        return a - b;
    return a + b;
}
```

Coverage metrics: useful but dangerous

- **Can give false sense of security**
- Examples of what coverage analysis could miss
  - Data values
  - Concurrency issues – race conditions etc.
  - Usability problems
  - Customer requirements issues
- **High branch coverage is *not* sufficient**

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

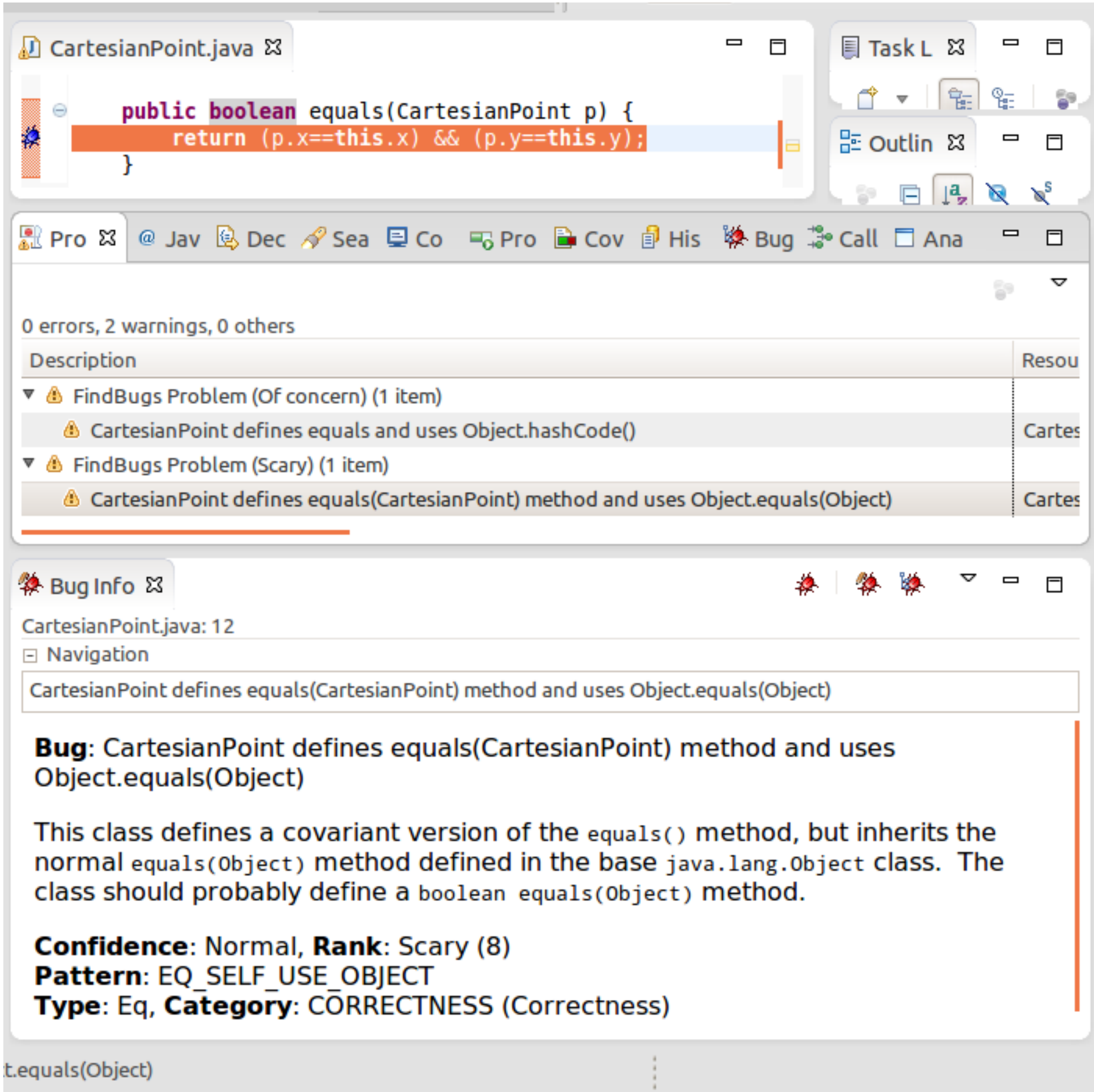
STATIC ANALYSIS

institute for SOFTWARE RESEARCH

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());
    }
}
```

FindBugs



CartesianPoint.java
public boolean equals(CartesianPoint p) {
return (p.x==this.x) && (p.y==this.y);
}
Task L
Outlin
Pro Jav Dec Sea Co Pro Cov His Bug Call Ana
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
CartesianPoint.java: 12
Navigation
CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)
Bug: CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)
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)
t.equals(Object)

~~Stupid~~ Subtle Bugs

```
public class Object {
    public boolean equals(Object other) { … }

    // other methods…
}

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());
    }
}
```

classes with no explicit superclass implicitly extend `Object`

can't change argument type when overriding

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

Fixing the Bug

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

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

        CartesianPoint that = (CartesianPoint) o;

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

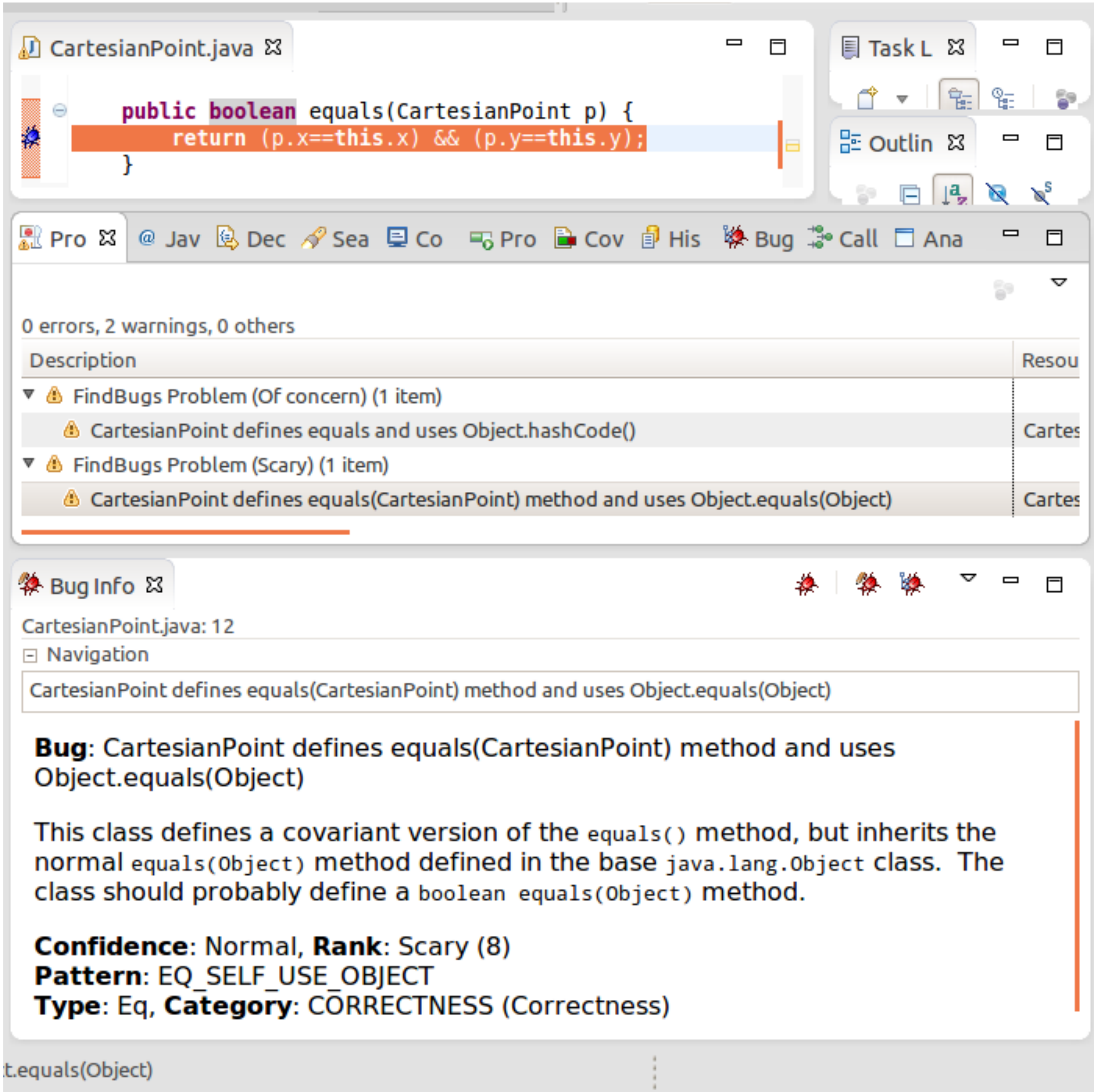
Declare our intent to override; Compiler checks that we did it

Use the same argument type as the method we are overriding

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

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

FindBugs

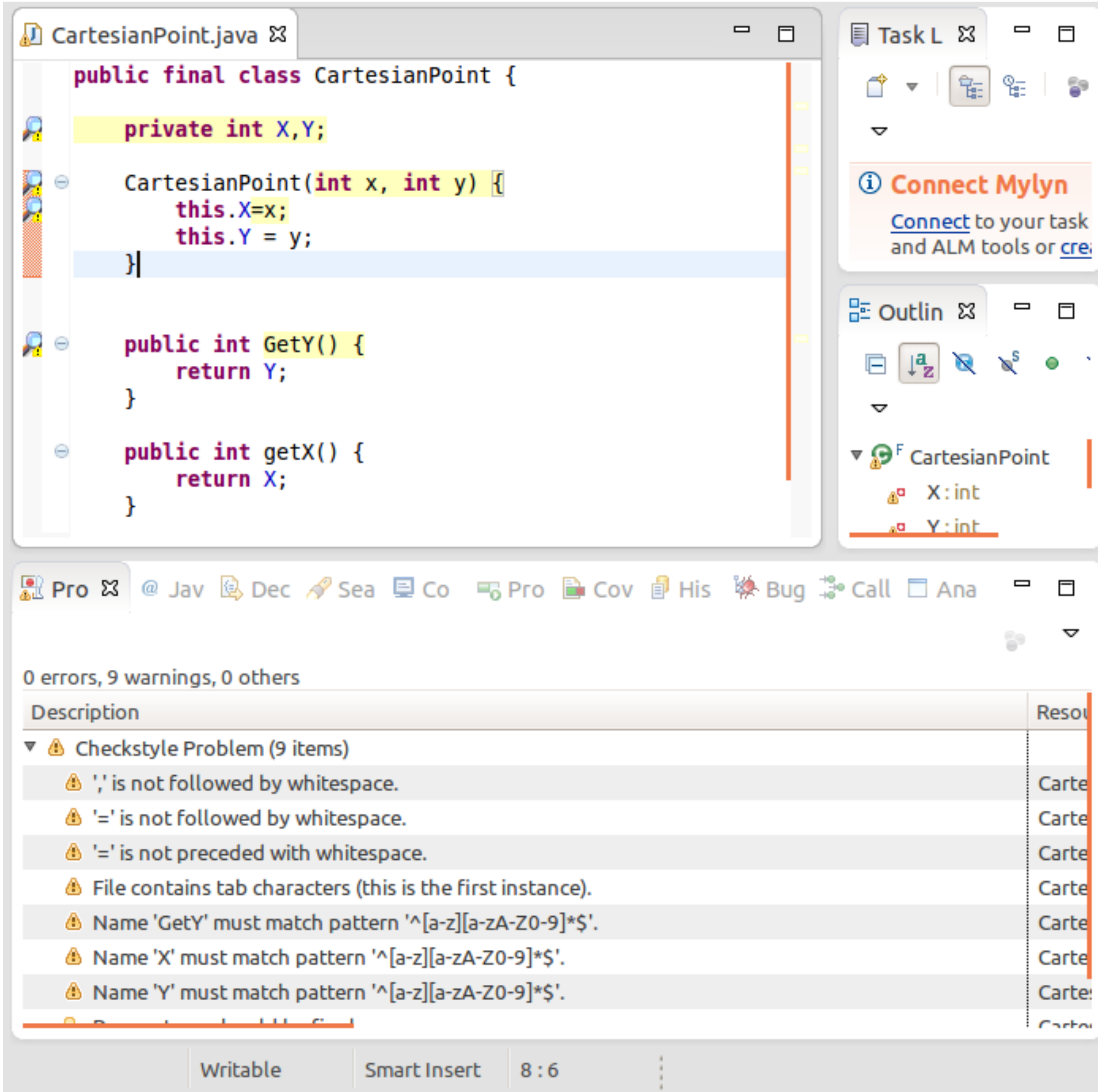


CartesianPoint.java
public boolean equals(CartesianPoint p) {
return (p.x==this.x) && (p.y==this.y);
}
Task L
Outlin
Pro
Jav
Dec
Sea
Co
Pro
Cov
His
Bug
Call
Ana
0 errors, 2 warnings, 0 others
Description
Resou
FindBugs Problem (Of concern) (1 item)
CartesianPoint defines equals and uses Object.hashCode()
Cartes
FindBugs Problem (Scary) (1 item)
CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)
Cartes
Bug Info
CartesianPoint.java: 12
Navigation
CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)
Bug: CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)
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)
t.equals(Object)

FindBugs

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

CheckStyle


CartesianPoint.java
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;
}
Task L
Connect Mylyn
Connect to your task and ALM tools or crea
Outlin
CartesianPoint
X : int
Y : int
Pro
Jav
Dec
Sea
Co
Pro
Cov
His
Bug
Call
Ana
0 errors, 9 warnings, 0 others
Description
Resou
Checkstyle Problem (9 items)
',' is not followed by whitespace.
'=' is not followed by whitespace.
'=' is not preceded with whitespace.
File contains tab characters (this is the first instance).
Name 'GetY' must match pattern '^[a-z][a-zA-Z0-9]*$'.
Name 'X' must match pattern '^[a-z][a-zA-Z0-9]*$'.
Name 'Y' must match pattern '^[a-z][a-zA-Z0-9]*$'.
Writable
Smart Insert
8 : 6

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

```java
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");
}
```

```java
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

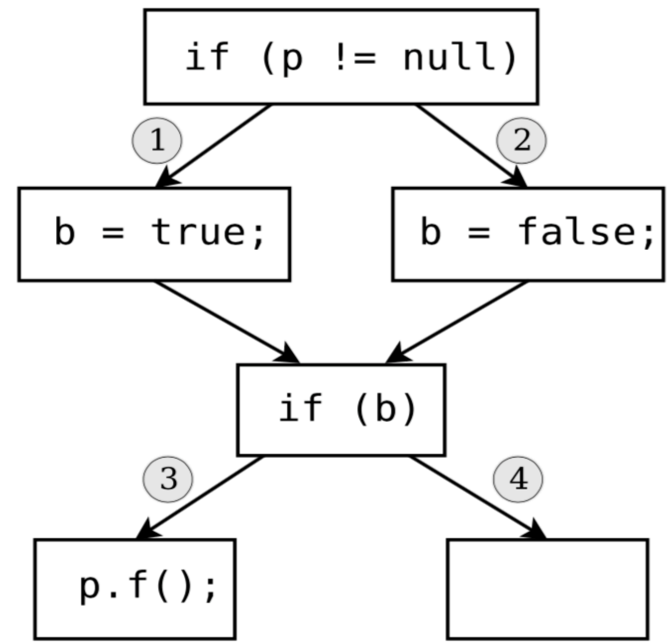
```
Detector foo = new Detector(…);
foo.execute();
```

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?

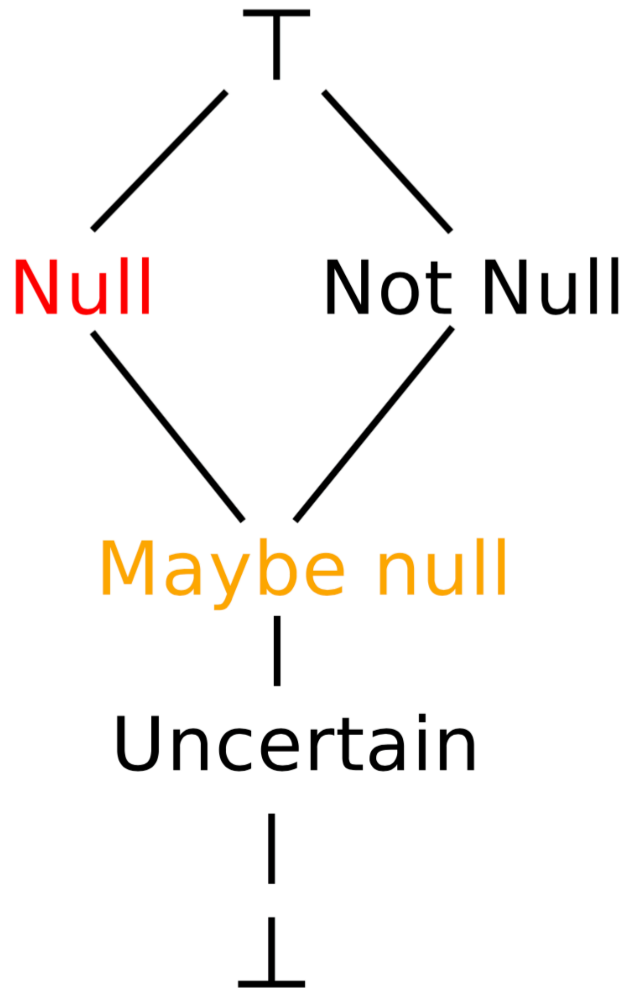
```
void foo(Object obj) {
        int x = obj.hashcode();
        …
}
```

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

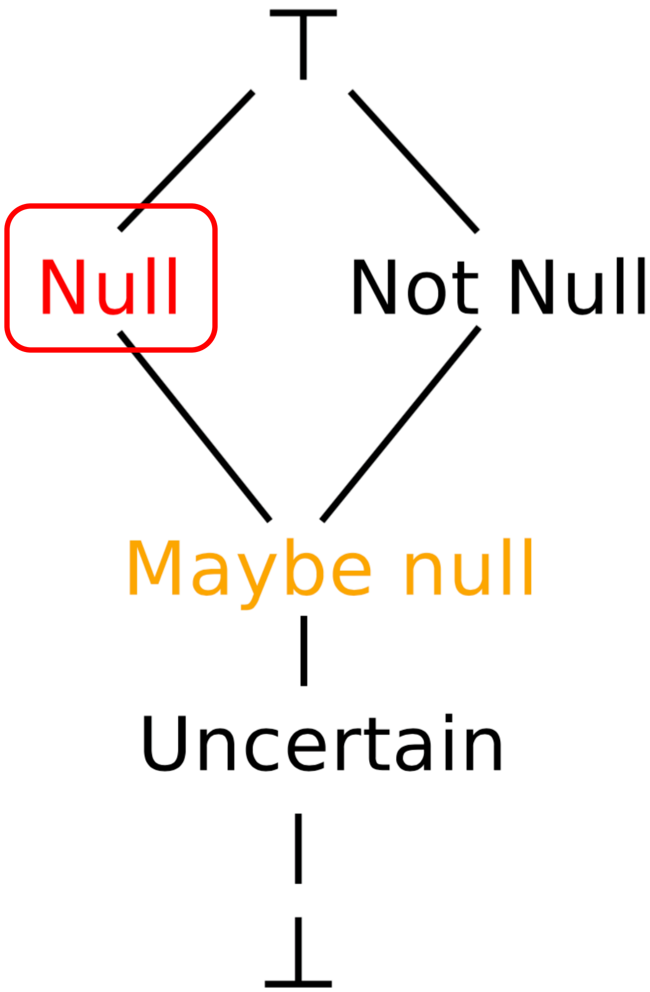
- Model values of local variables and stack operands using lattice of symbolic values
- When two control paths merge, use *meet* operator to combine values:



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:

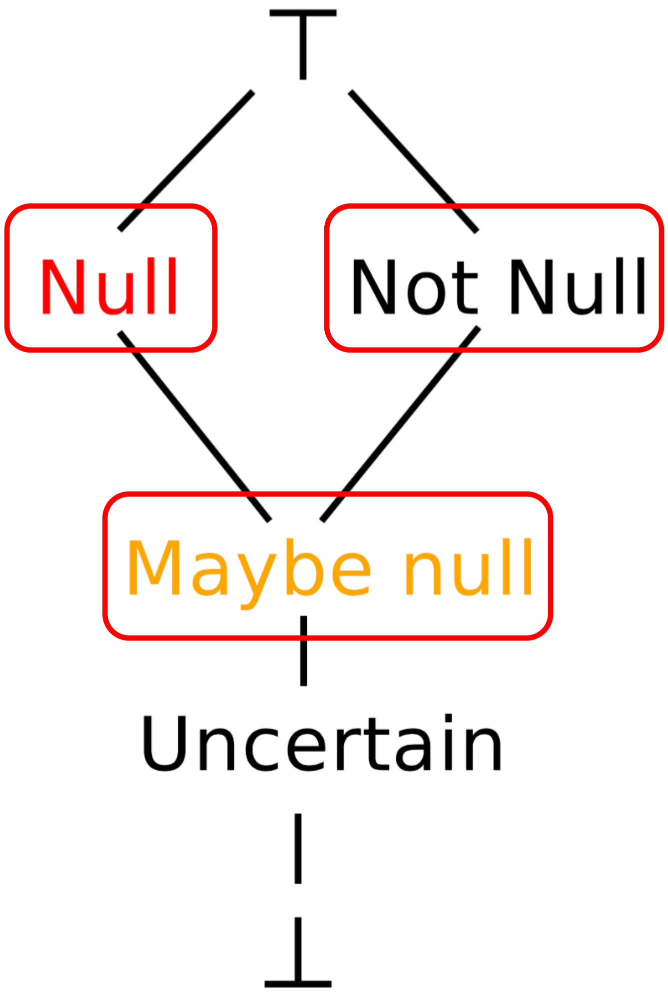
Null ◇ Null = Null



Dataflow values

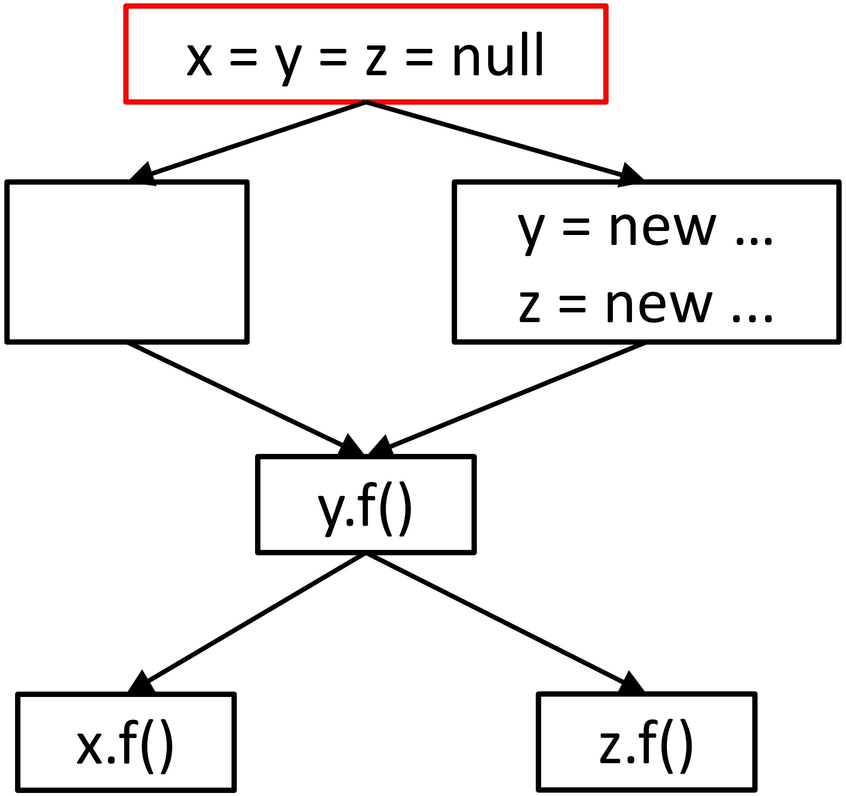
- Model values of local variables and stack operands using lattice of symbolic values
- When two control paths merge, use *meet* operator to combine values:

Null ◇ Not Null = Maybe Null



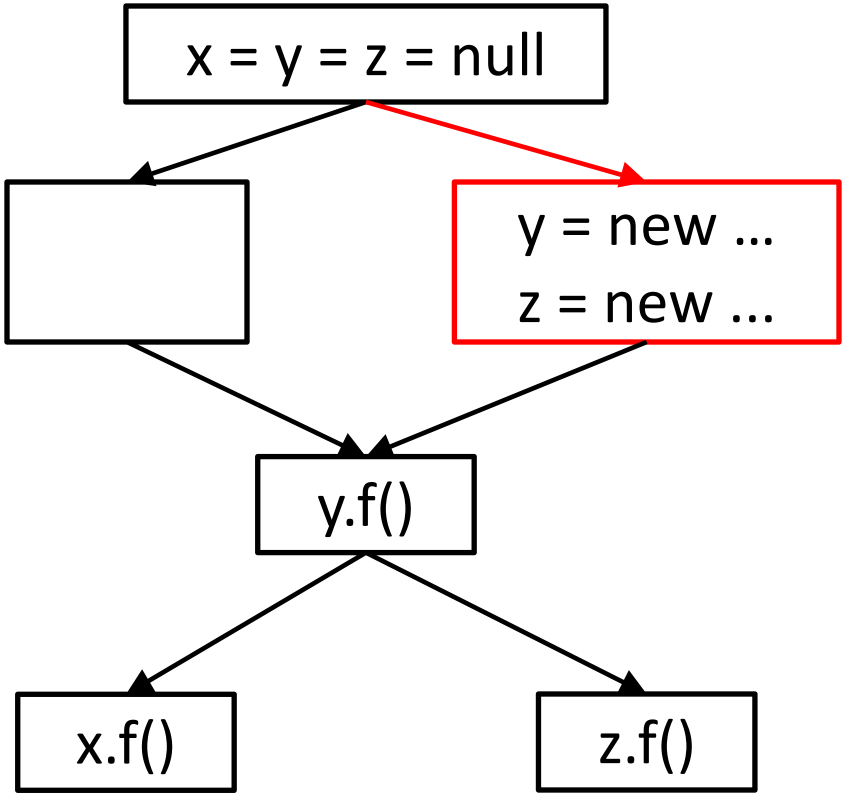
Null-pointer dataflow example

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



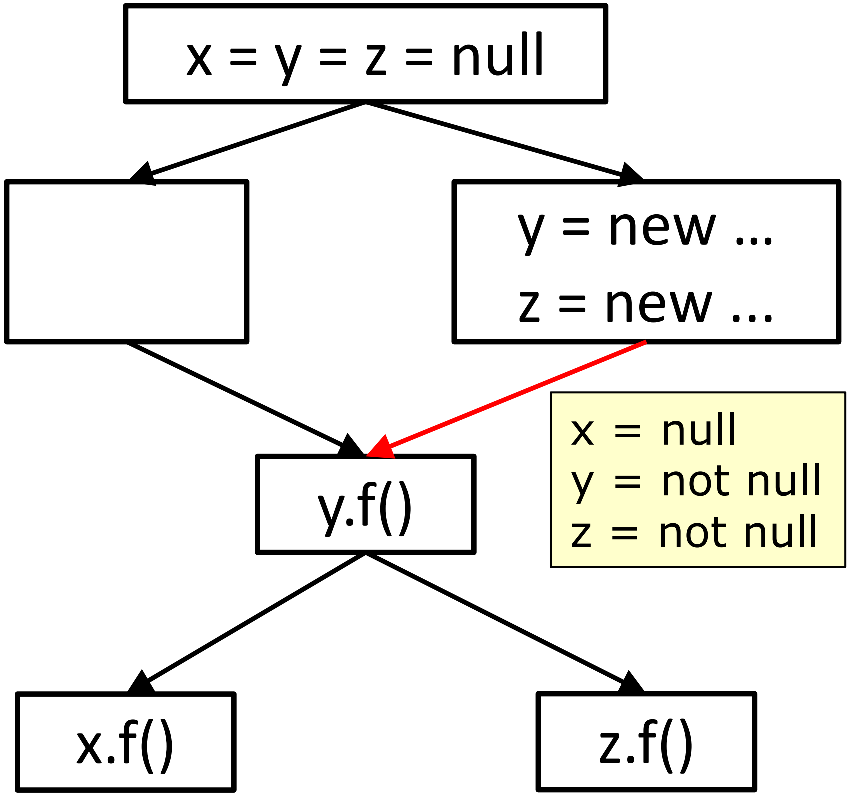
Null-pointer dataflow example

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



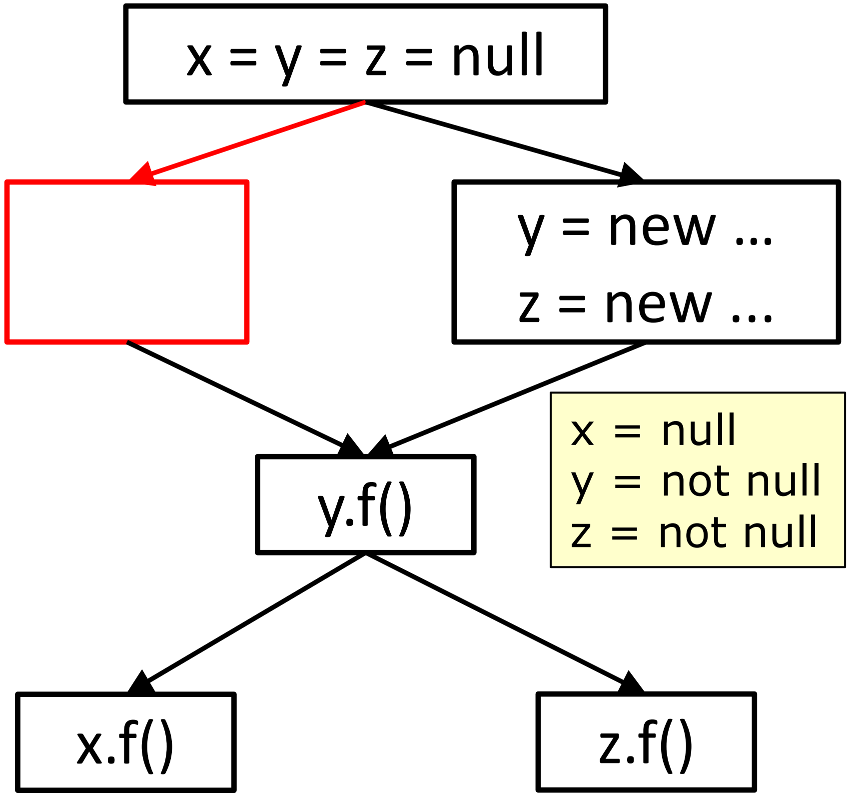
Null-pointer dataflow example

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



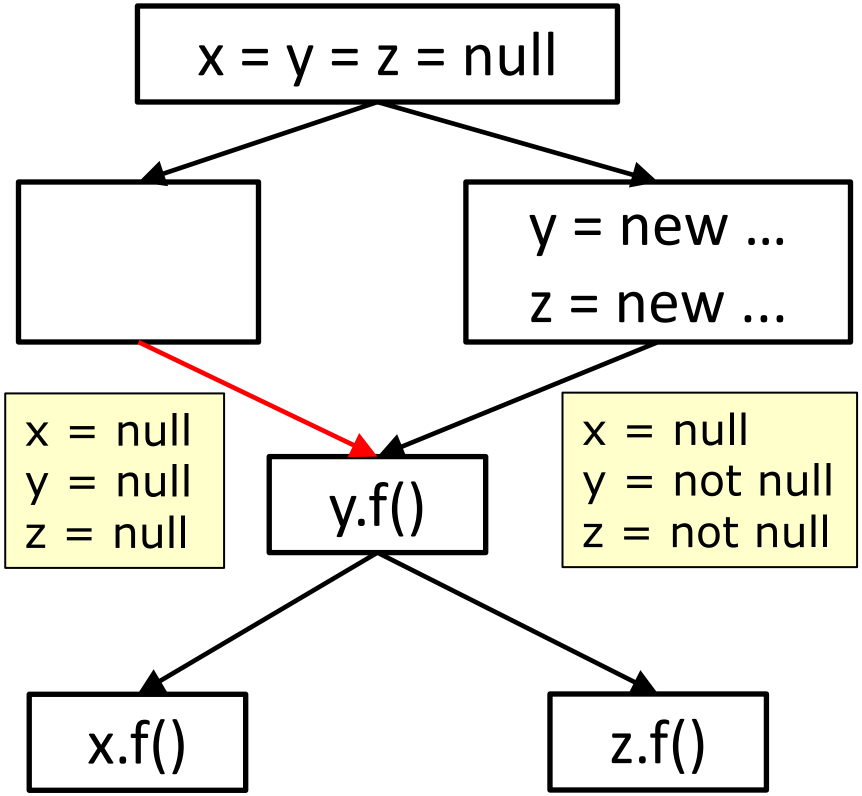
Null-pointer dataflow example

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



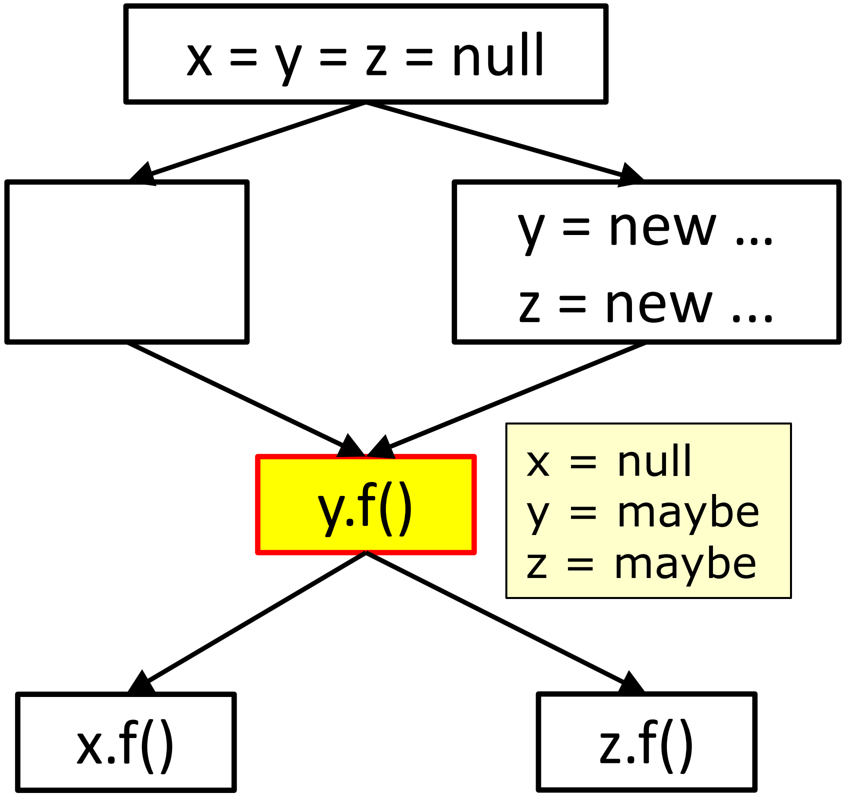
Null-pointer dataflow example

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



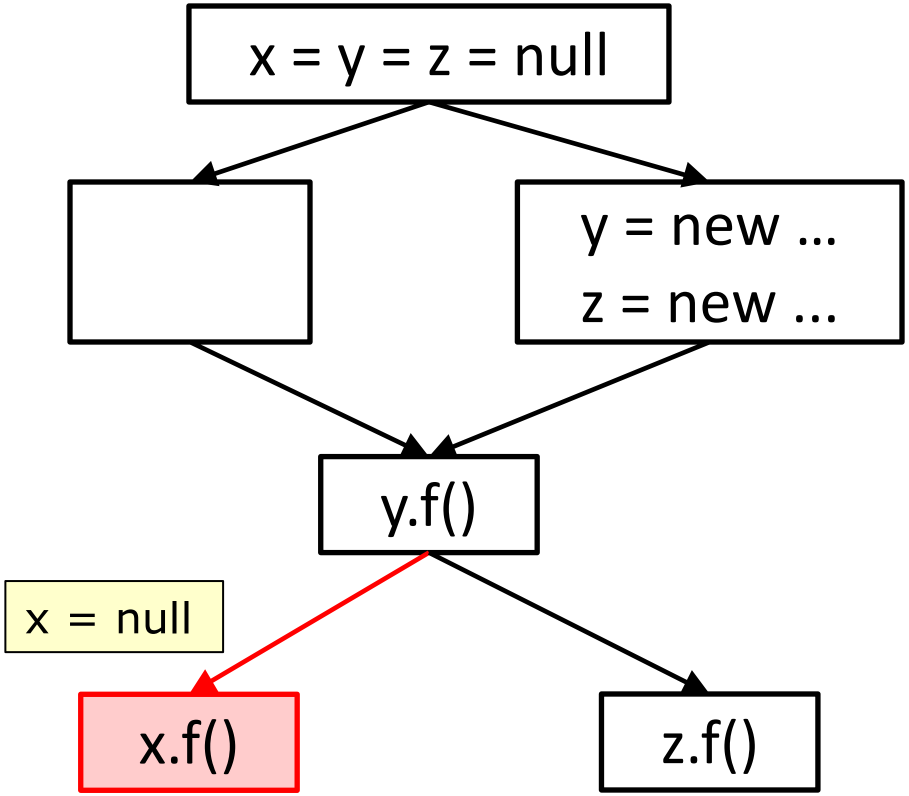
Null-pointer dataflow example

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



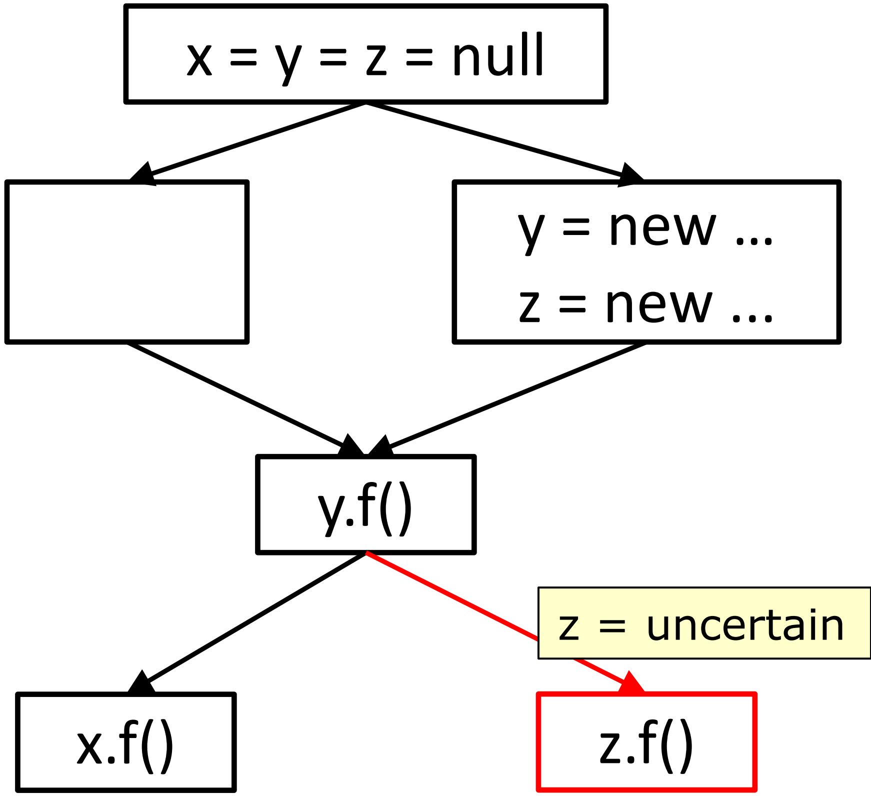
Null-pointer dataflow example

```
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();
```



Abstract Interpretation

- Static program analysis is the **systematic examination** of an **abstraction of a program's state space**
- Abstraction
  - Don't track everything! (that's normal interpretation)
  - Track an important abstraction
- Systematic
  - Ensure everything is checked in the same way

Details on how this works in 15-313

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:

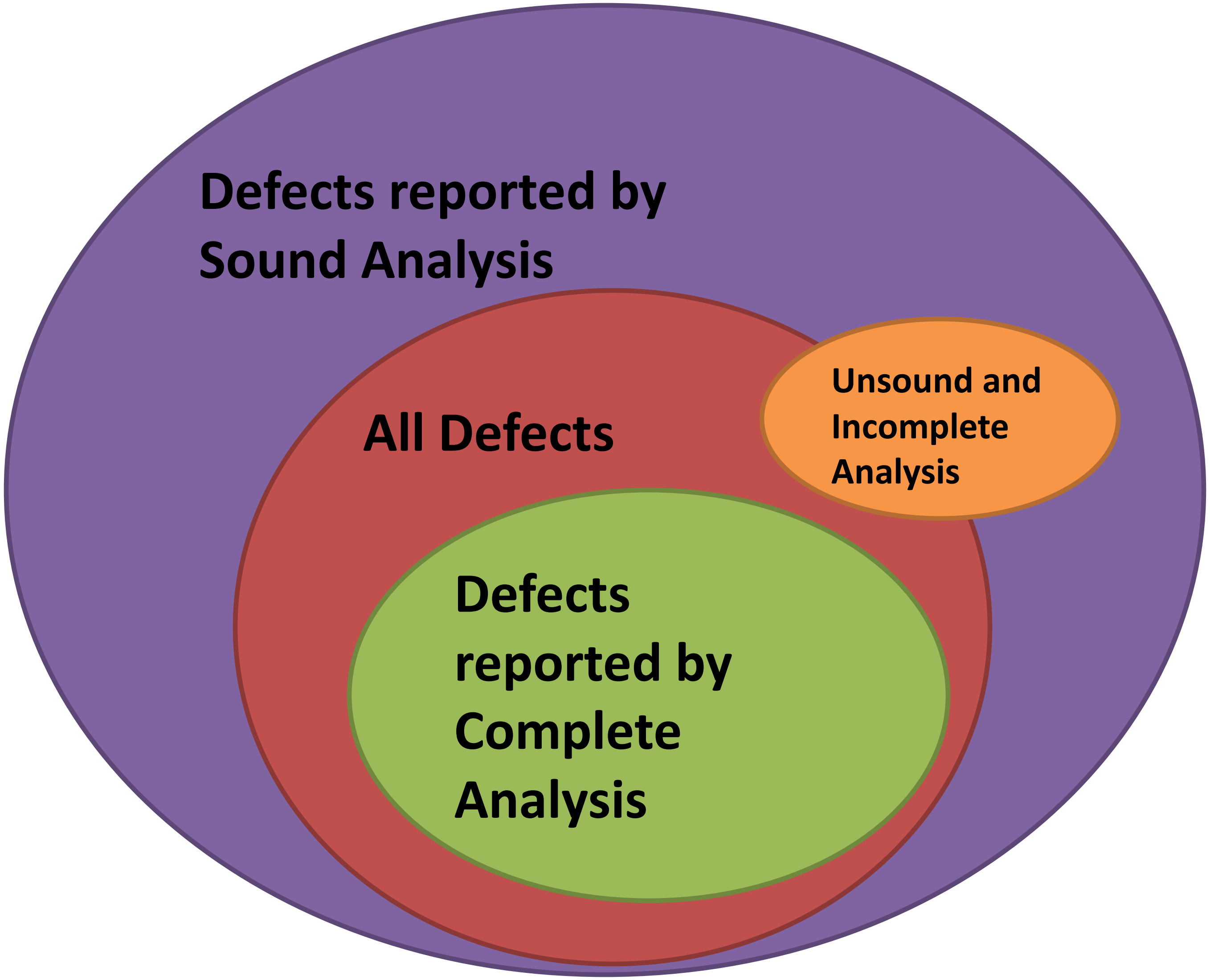
reports all defects
→ no false negatives
typically overapproximated

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?


Defects reported by Sound Analysis
All Defects
Unsound and Incomplete Analysis
Defects reported by 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