

Principles of Software Construction: Objects, Design, and Concurrency

Incremental Improvements

Michael Hilton

Bogdan Vasilescu



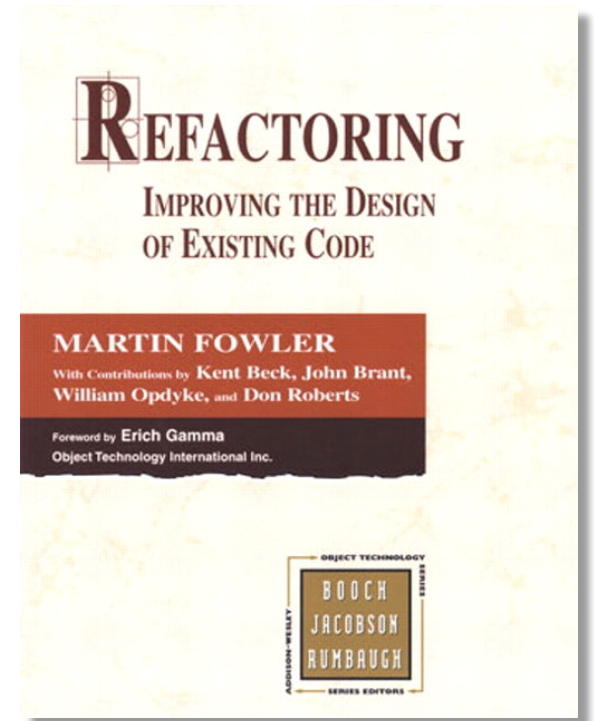
Administrivia

- HW 4a due tonight
- Midterms returned Tuesday (leftovers are here)

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

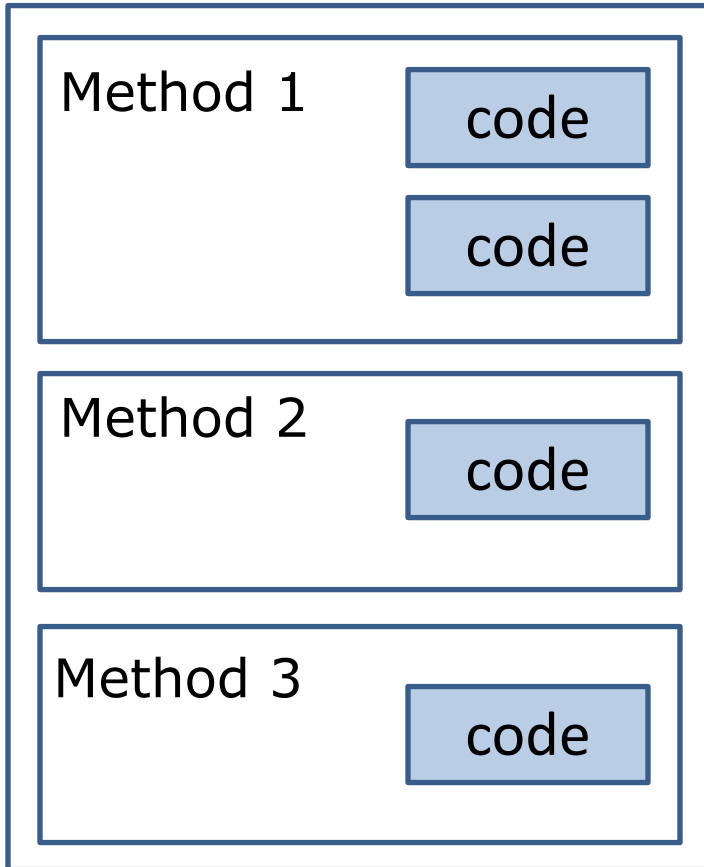


Bad Smells: Classification

- 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)

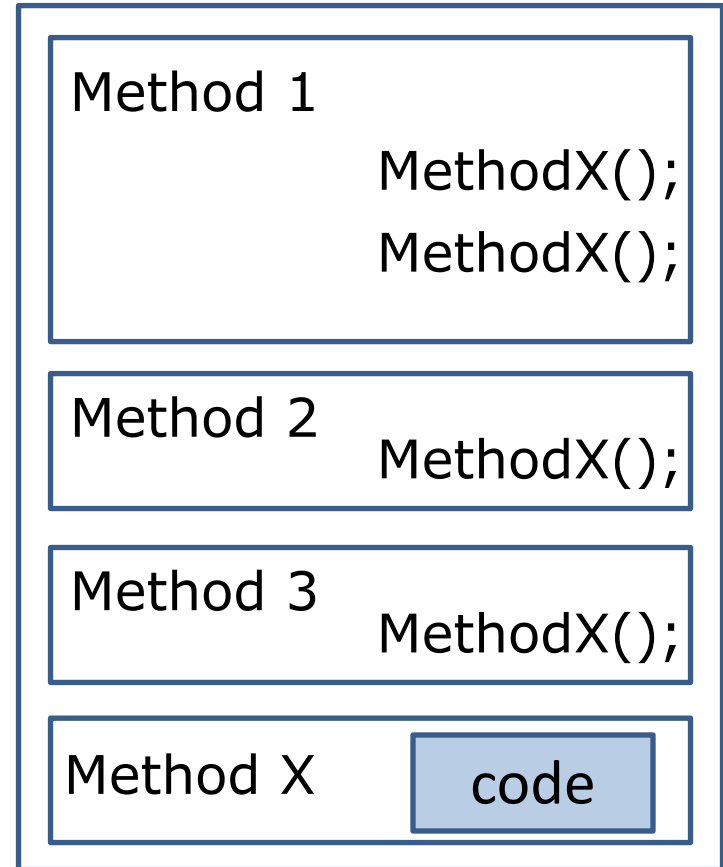
Class



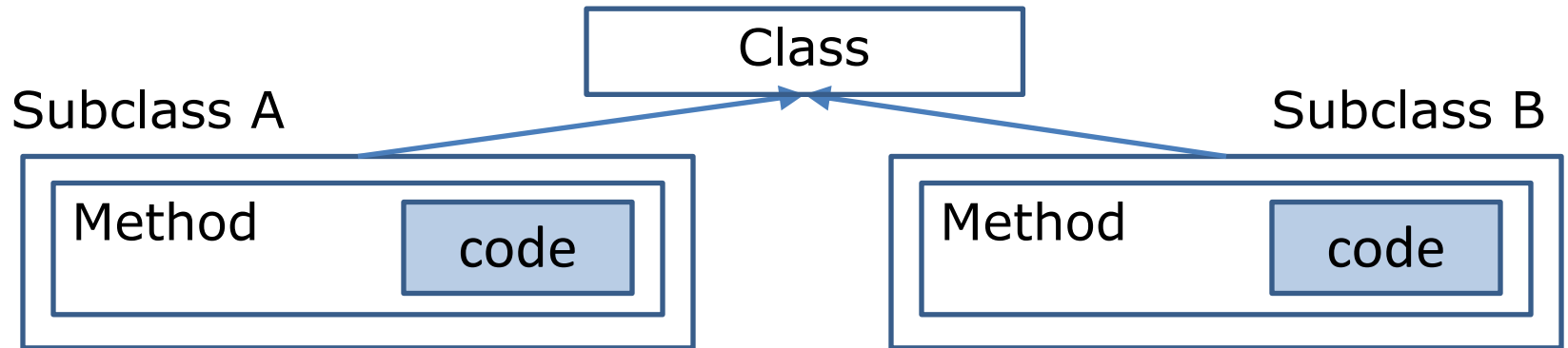
- Extract method
- Rename method



Class



Code duplication (2)

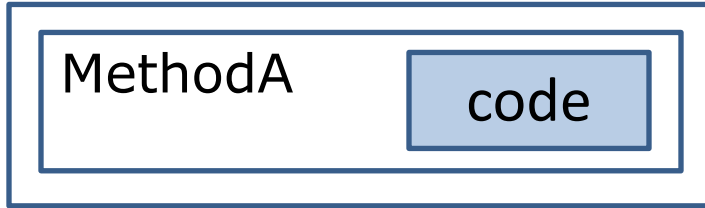


Same expression in two sibling classes:

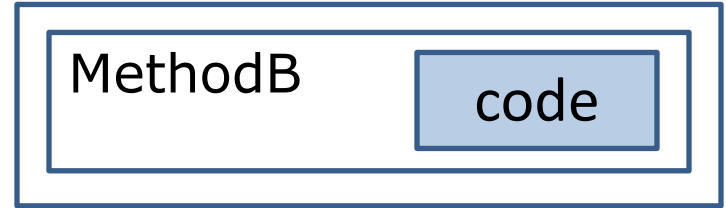
- Same code: Extract method + Pull up field
- Similar code: Extract method + Form Template Method
- Different algorithm: Substitute algorithm

Code duplication (3)

ClassA



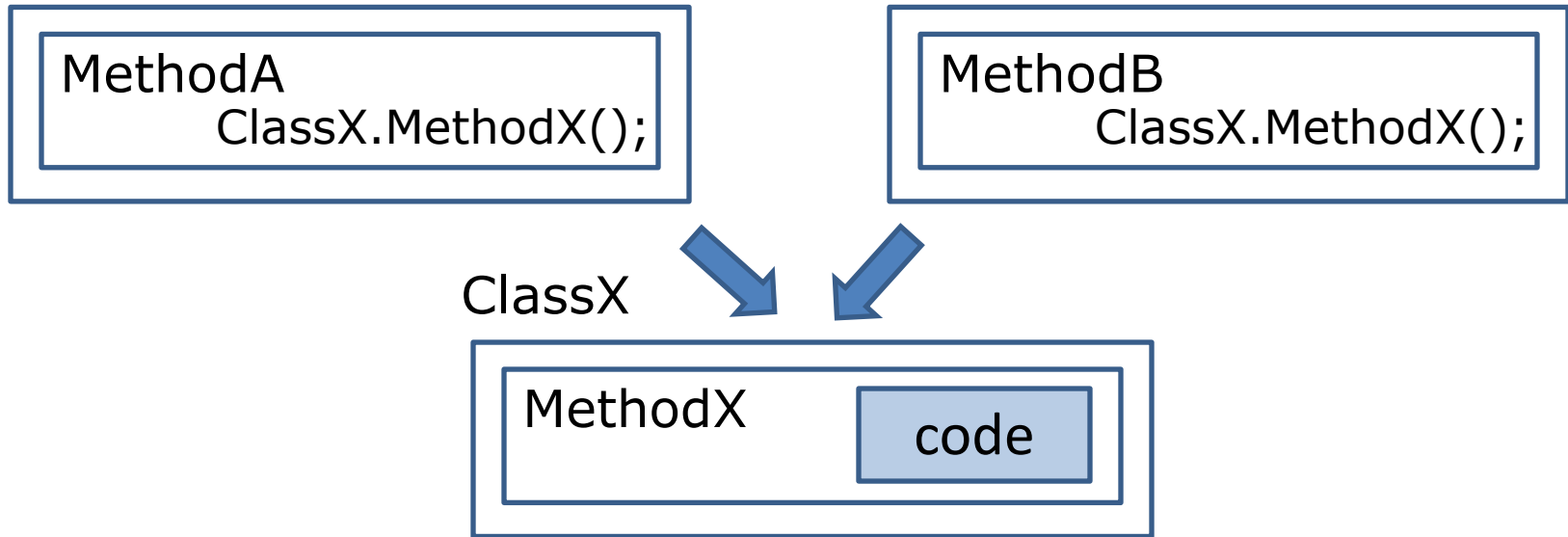
ClassB



Code duplication (3)

ClassA

ClassB



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


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){...}
```

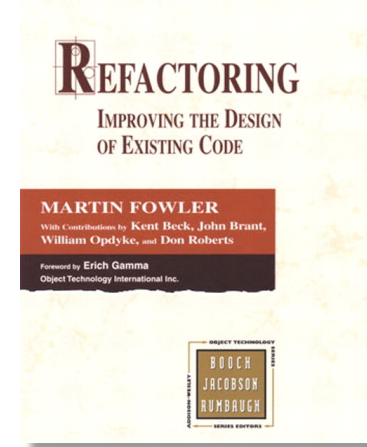
```
double getOutstanding() {  
    Enumeration e = _orders.elements();  
    double result = 0.0;  
    While (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        result += each.getAmount();  
    }  
    return result;  
}
```

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



ANTI-PATTERNS

Anti-patterns

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

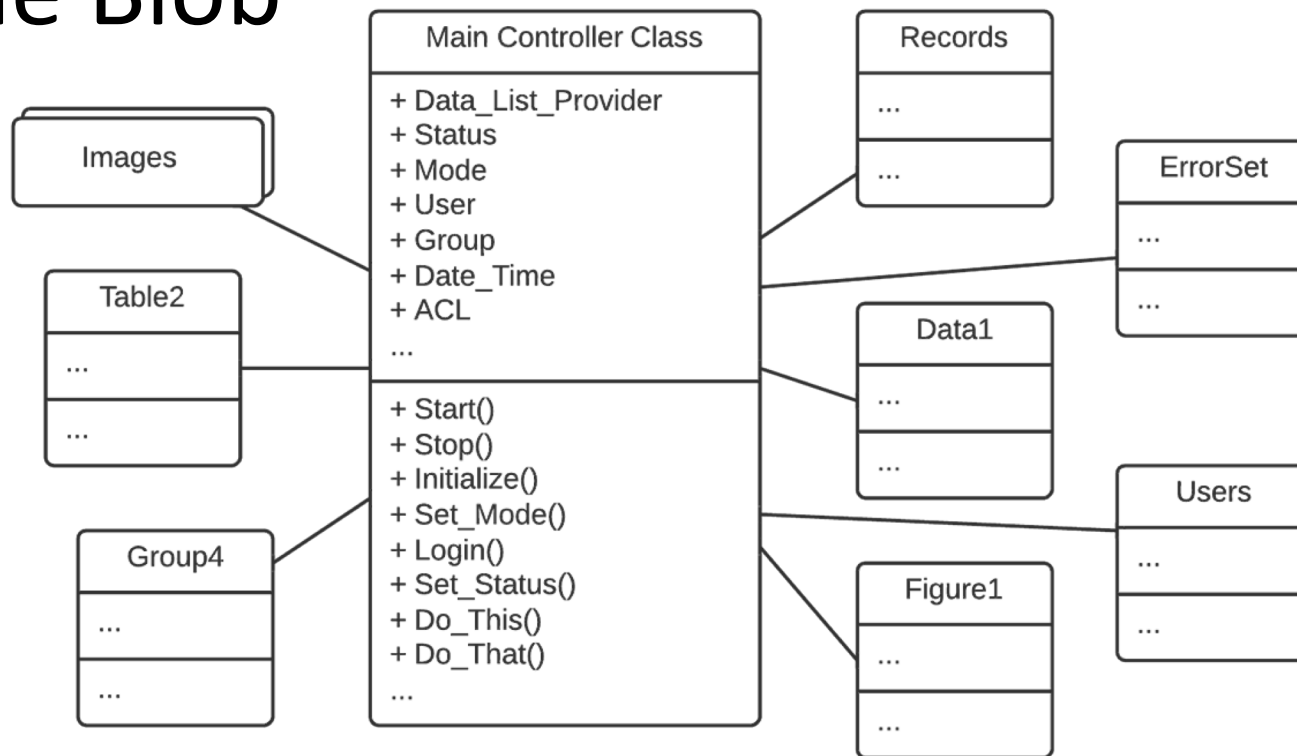
Common anti-patterns

- 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

```
38 }
39 public boolean equals(Object anObject) {
40     if (isZero())
41         if (anObject instanceof IMoney)
42             return ((IMoney)anObject).isZero();
43     if (anObject instanceof Money) {
44         Money aMoney= (Money)anObject;
45         return aMoney.currency().equals(currency())
46             && amount() == aMoney.amount();
47     }
48     return false;
49 }
```

- Does this guarantee correctness?

Statement Coverage

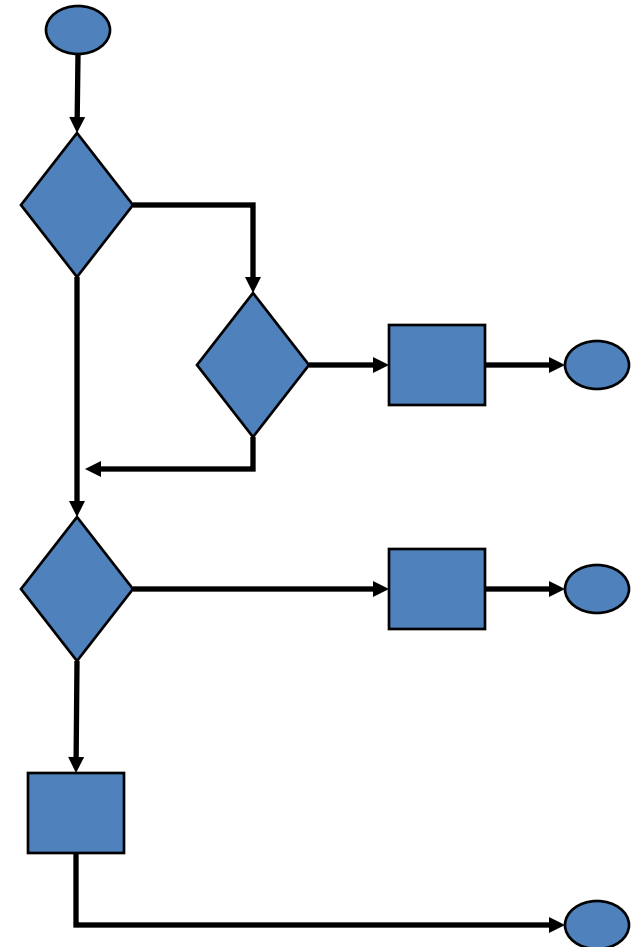
- Trying to test all parts of the implementation
- Execute every statement in at least one test

```
38     }
39     public boolean equals(Object anObject) {
40         if (isZero())
41             if (anObject instanceof IMoney)
42                 return ((IMoney)anObject).isZero();
43         if (anObject instanceof Money) {
44             Money aMoney= (Money)anObject;
45             return aMoney.currency().equals(currency())
46                    && amount() == aMoney.amount();
47         }
48         return false;
49     }
50     public int hashCode() {
```

- Does this guarantee correctness?

Structure of Code Fragment to Test

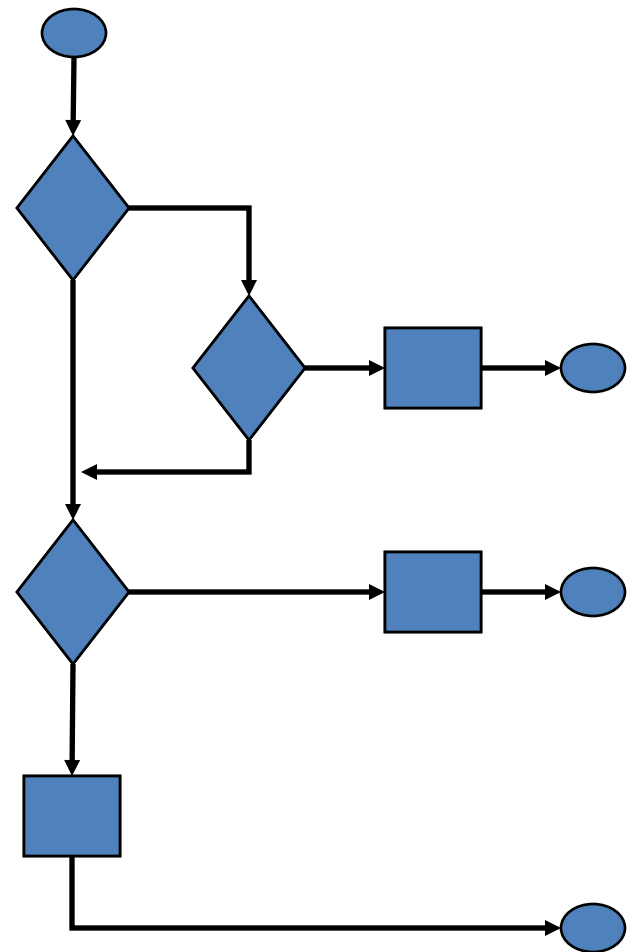
```
38 }
39 public boolean equals(Object anObject) {
40     if (isZero())
41         if (anObject instanceof IMoney)
42             return ((IMoney)anObject).isZero();
43     if (anObject instanceof Money) {
44         Money aMoney= (Money)anObject;
45         return aMoney.currency().equals(currency())
46             && amount() == aMoney.amount();
47     }
48     return false;
49 }
```



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

Statement Coverage

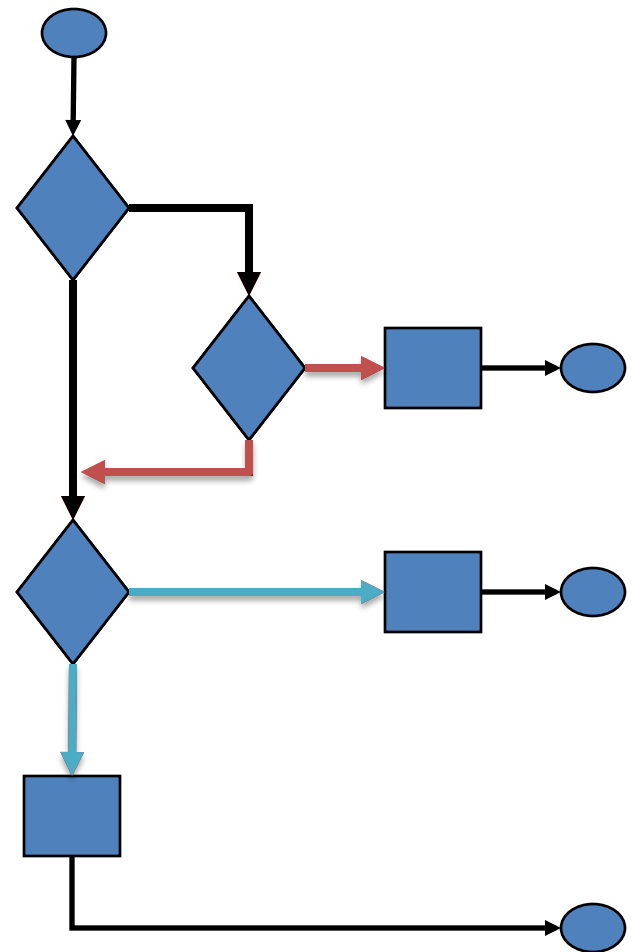
- Statement coverage
 - What portion of program statements (nodes) are touched by test cases
- Advantages
 - Test suite size linear in size of code
 - Coverage easily assessed
- Issues
 - Dead code is not reached
 - May require some sophistication to select input sets
 - Fault-tolerant error-handling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



```
}  
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

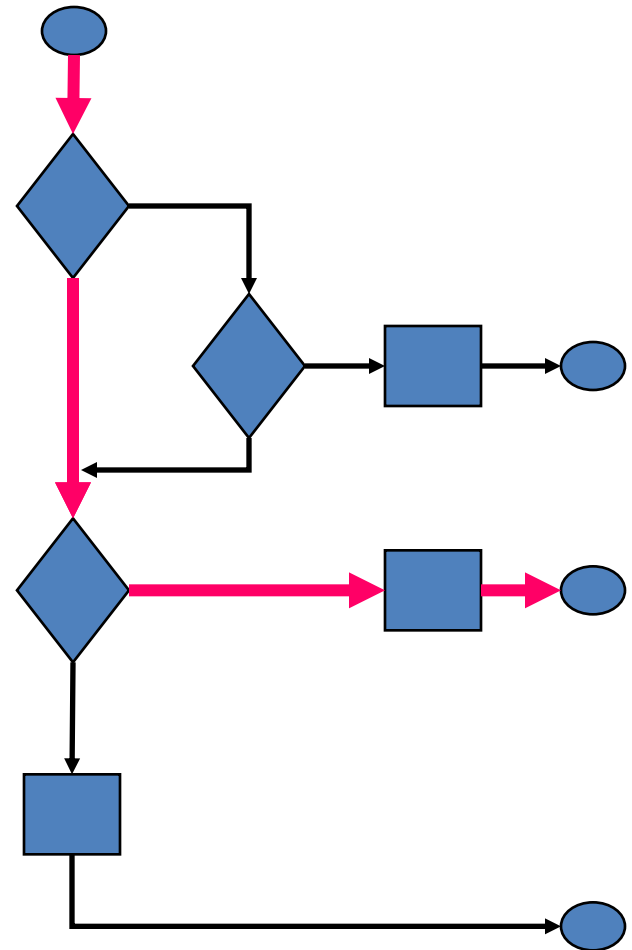
- Branch coverage
 - What portion of condition branches are covered by test cases?
 - Or: What portion of relational expressions and values are covered by test cases?
 - Condition testing (Tai)
 - Multicondition coverage – all boolean combinations of tests are covered
- Advantages
 - Test suite size 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”



```
}  
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;  
}
```

Packages

[All](#)

[net.sourceforge.cobertura.ant](#)

[net.sourceforge.cobertura.check](#)

[net.sourceforge.cobertura.coveragedata](#)

[net.sourceforge.cobertura.instrument](#)

[net.sourceforge.cobertura.merge](#)

[net.sourceforge.cobertura.reporting](#)

[net.sourceforge.cobertura.reporting.html](#)

[net.sourceforge.cobertura.reporting.html.files](#)

[net.sourceforge.cobertura.reporting.xml](#)

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

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 and also 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 and 100% branch coverage and 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

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());  
    }  
}
```

SpotBugs

The screenshot shows an IDE with the following components:

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

```
public boolean equals(CartesianPoint p) {  
    return (p.x==this.x) && (p.y==this.y);  
}
```
- Task List:** Shows a task labeled "Task L".
- Outline:** Shows the project structure.
- Problems:** A panel showing 0 errors, 2 warnings, and 0 others. It lists two FindBugs problems:
 - FindBugs Problem (Of concern) (1 item):** CartesianPoint defines equals and uses Object.hashCode().
 - FindBugs Problem (Scary) (1 item):** CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object).
- Bug Info:** A panel providing details for the selected bug:
 - 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)
 - Description:** 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)

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

```
    }  
}
```

SpotBugs

The screenshot shows an IDE with a Java file named `CartesianPoint.java`. The code defines a `public boolean equals(CartesianPoint p)` method that returns `(p.x==this.x) && (p.y==this.y);`. The IDE's bottom panel displays SpotBugs analysis results, indicating 0 errors, 2 warnings, and 0 others. The warnings are categorized as 'FindBugs Problem (Of concern)' and 'FindBugs Problem (Scary)'. The 'Bug Info' panel provides details for the 'Scary' bug: 'CartesianPoint defines equals(CartesianPoint) method and uses Object.equals(Object)'. The bug description states that the class defines a covariant version of the `equals()` method but inherits the normal `equals(Object)` method from the base `java.lang.Object` class. The bug is classified as 'Normal' confidence, 'Scary (8)' rank, with the pattern 'EQ_SELF_USE_OBJECT' and type 'Eq, Category: CORRECTNESS (Correctness)'.

```
public boolean equals(CartesianPoint p) {  
    return (p.x==this.x) && (p.y==this.y);  
}
```

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)

SpotBugs



```
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 with a Java file named `CartesianPoint.java`. The code defines a `CartesianPoint` class with private fields `X` and `Y`, a constructor, and getter methods. The IDE's right sidebar shows the 'Task List' and 'Outlin' panels. The 'Outlin' panel shows the class structure with fields `X:int` and `Y:int`. The bottom panel displays the 'Pro' (Problems) view, showing 0 errors, 9 warnings, and 0 others. The 'Checkstyle Problem (9 items)' section lists the following warnings:

- ' ' 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]*$`.

The bottom status bar indicates 'Writable', 'Smart Insert', and '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 SpotBugs 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;

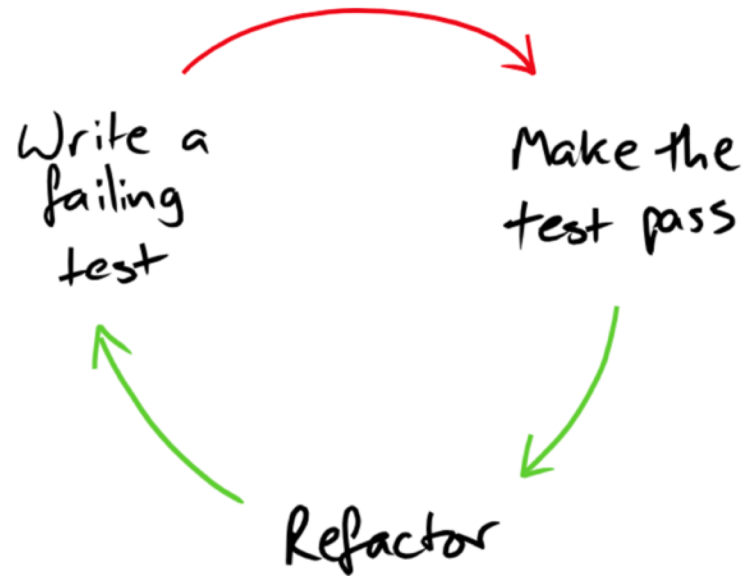
if (addressTypeCode.equals(one)) {
    System.out.println("equals");
} else {
    System.out.println("not equals");
}
```

TEST DRIVEN DEVELOPMENT (TDD)

Three simple rules

1. You are not allowed to write any production code unless it is to make a **failing unit test pass**.
2. You are not allowed to write any more of a unit test than is **sufficient to fail**; and compilation failures are failures.
3. You are not allowed to write any more production code than is **sufficient to pass** the one failing unit test.

TDD Cycle



From Growing Object-Oriented Software by Nat Pryce and Steve Freeman

<http://www.growing-object-oriented-software.com/figures.html>

@sebrose

<http://cucumber.io>

Why TDD?

“The act of writing a unit test is more an act of **design** than of verification.

It is also more an act of **documentation** than of verification.

The act of writing a unit test closes a remarkable number of feedback loops, the least of which is the one pertaining to **verification** of function”.

Advantages of TDD

- Clear place to start
- Much less code thrown away, less wasted effort
- Less Fear
- Side Effect: Robust test suite

A programming exercise that you repeat many many times, looking to make small, incremental improvements.

CODE KATA

Diamond Kata

- Given a letter, print a diamond starting with 'A' with the supplied letter at the widest point.
- For example: 'C' prints

```
  A
 B B
C   C
 B B
  A
```

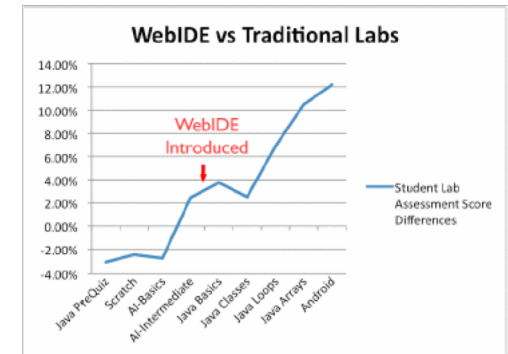


TDD Demo

IMPRESSIONS?

TDD Research

- Hilton et al.: Students learn better when forced to write tests first
- Bhat et al.: At Microsoft, projects using TDD had greater than two times code quality, but 15% more upfront setup time
- George et al.: TDD passed 18% more test cases, but took 16% more time
- Scanniello et al.: Perceptions of TDD include: novices believe TDD improves productivity at the expense of internal quality



More TDD Research

- Fucci et al.: Results: The Kruskal-Wallis tests did not show any significant difference between TDD and TLD in terms of testing effort (p-value = .27), external code quality (p-value = .82), and developers' productivity (p-value = .83).
- Fucci et al.: Conclusion: The claimed benefits of TDD may not be due to its distinctive test-first dynamic, but rather due to the fact that TDD-like processes encourage fine-grained, steady steps that improve focus and flow.

WHY IS THIS HARD?