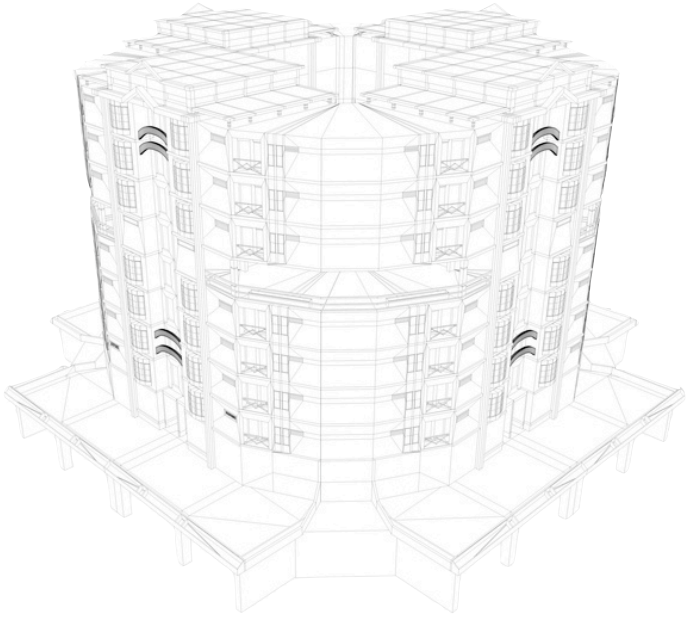
tdd:
test-driven *design*



NEAL FORD software architect / meme wrangler

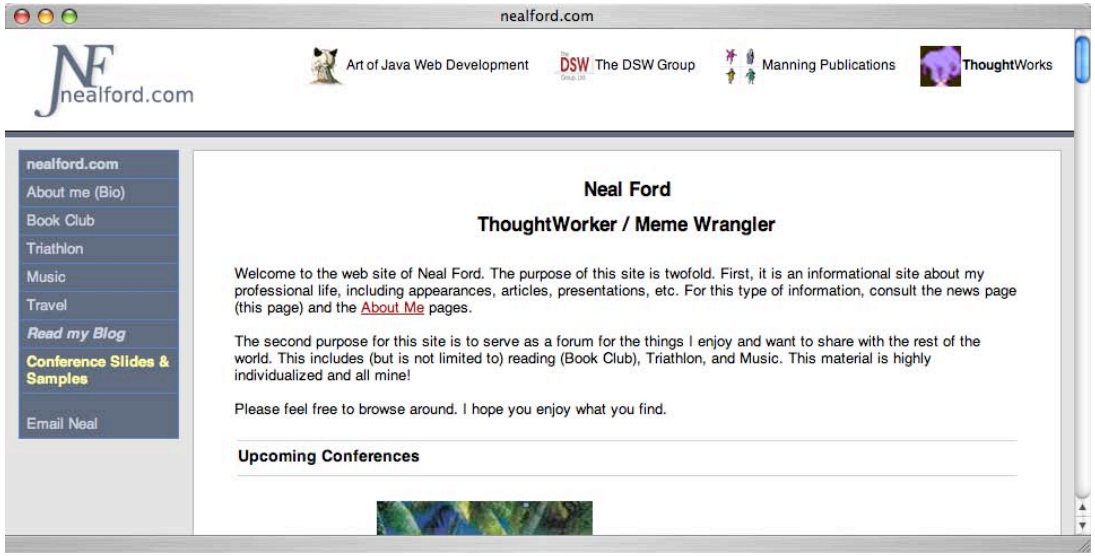
ThoughtWorks

nford@thoughtworks.com
3003 Summit Boulevard, Atlanta, GA 30319
www.nealford.com
www.thoughtworks.com
memeagora.blogspot.com

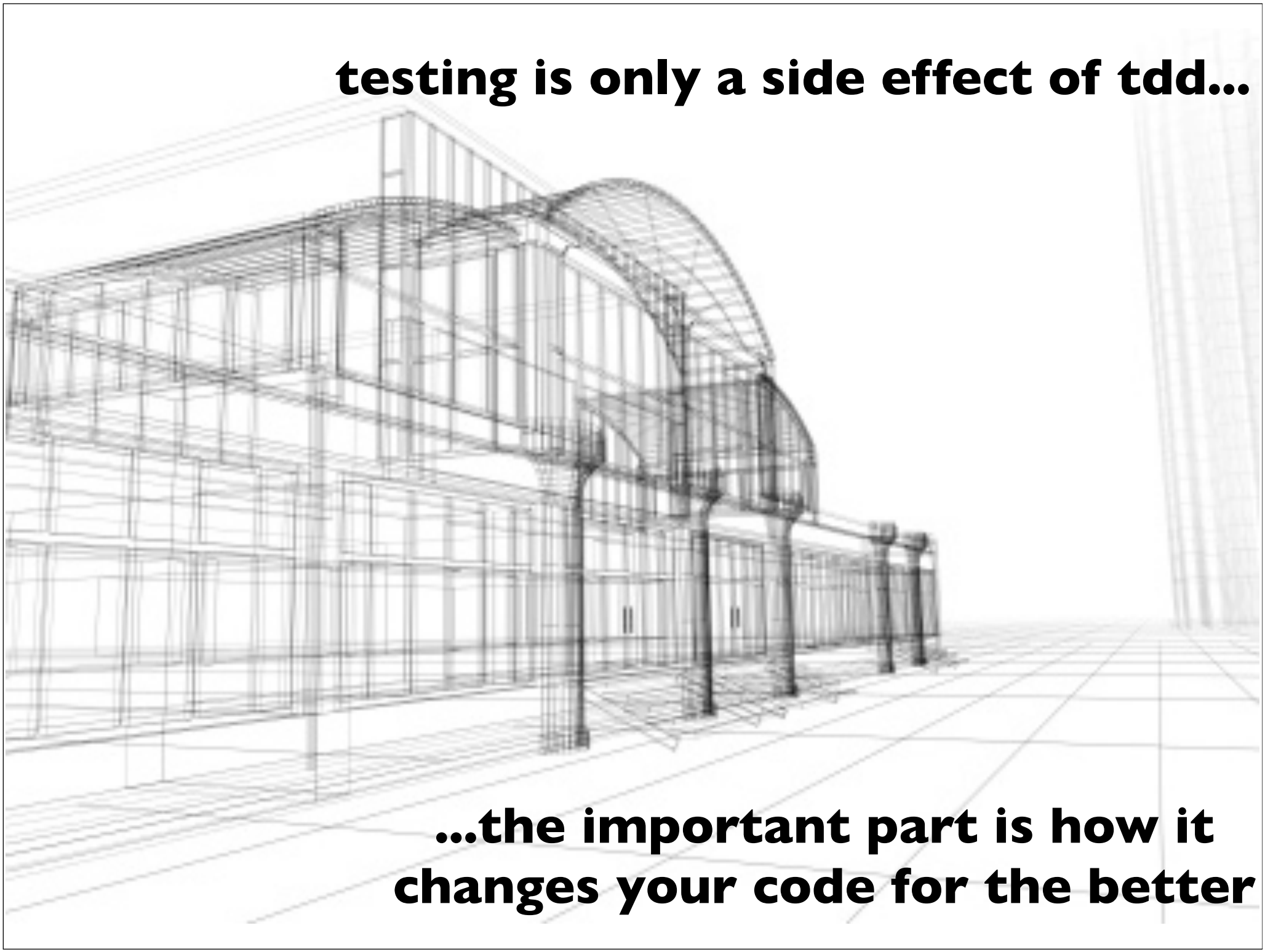
housekeeping

ask questions anytime

download slides from nealford.com



download samples from `github.com/nealford`


testing is only a side effect of tdd...
...the important part is how it changes your code for the better

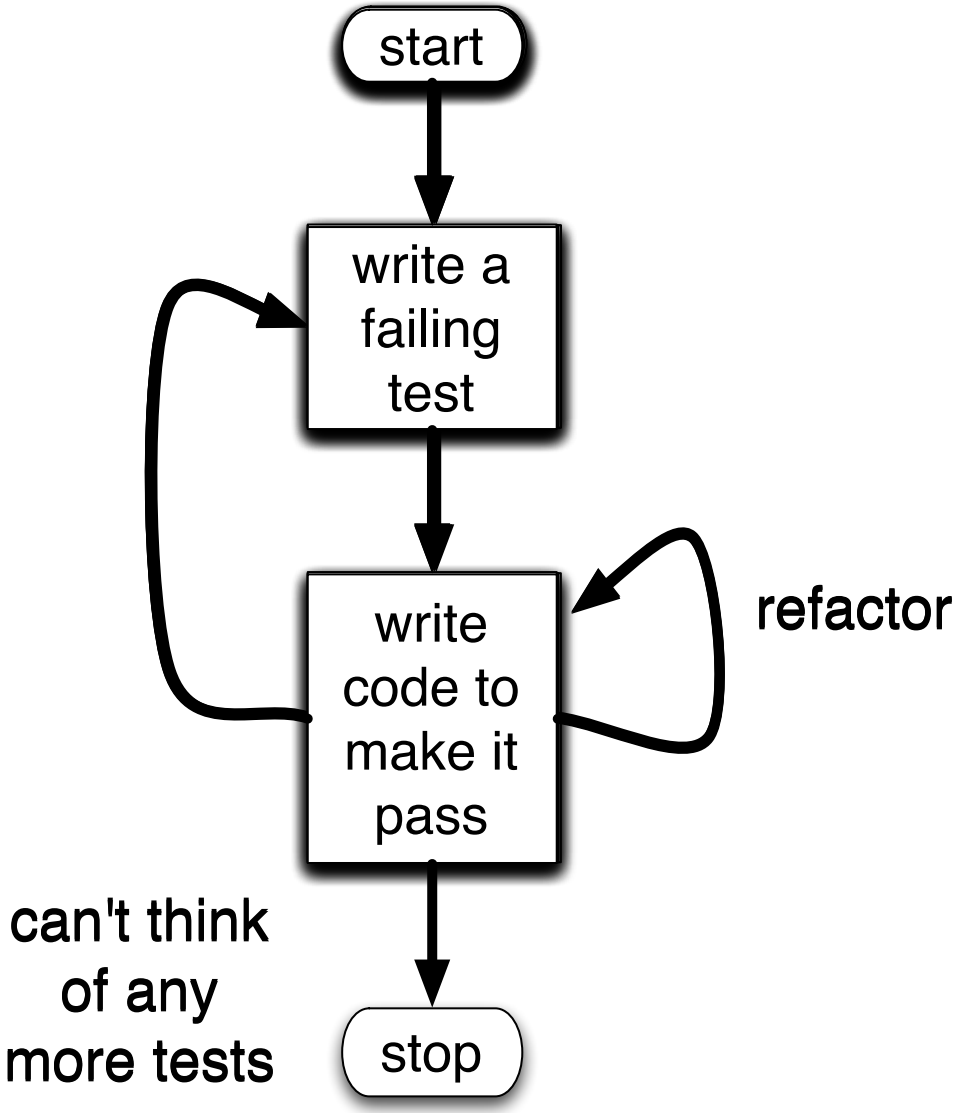
what i cover:

measuring code quality

the evolution of design

test-driven unit testing

testing what's hard to test



red green refactor

why watch it fail?

```
@Test public void sum() {
    Classifier6 c = new Classifier6(20);
    calculateFactors(c);
    int expected = 1 + 2 + 4 + 5 + 10 + 20;
    assertThat(expected, is(expected));
}
```

design benefits of tdd

think about how the rest of the world uses this class

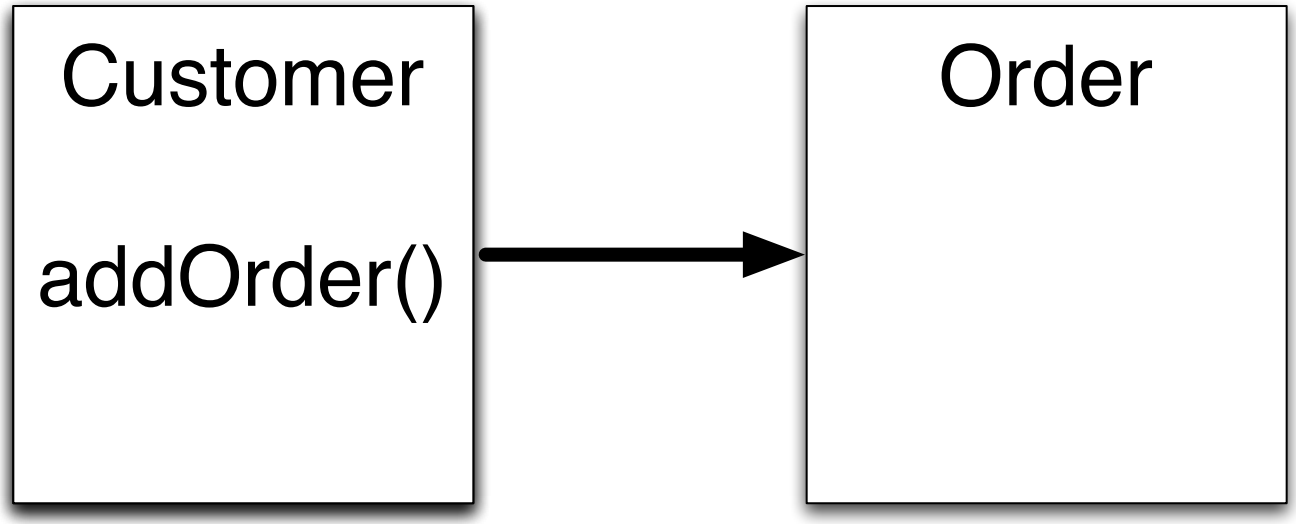
creates *consumption awareness*

design benefits of tdd

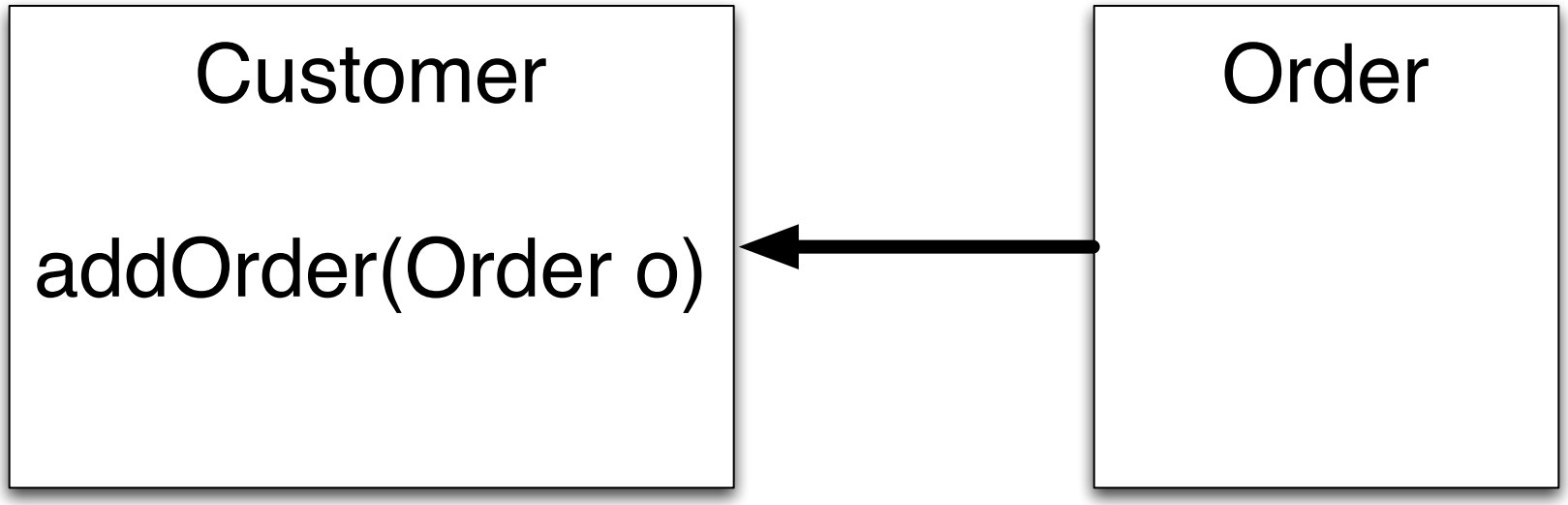
forces mocking of dependent objects

earliest possible object interaction decisions

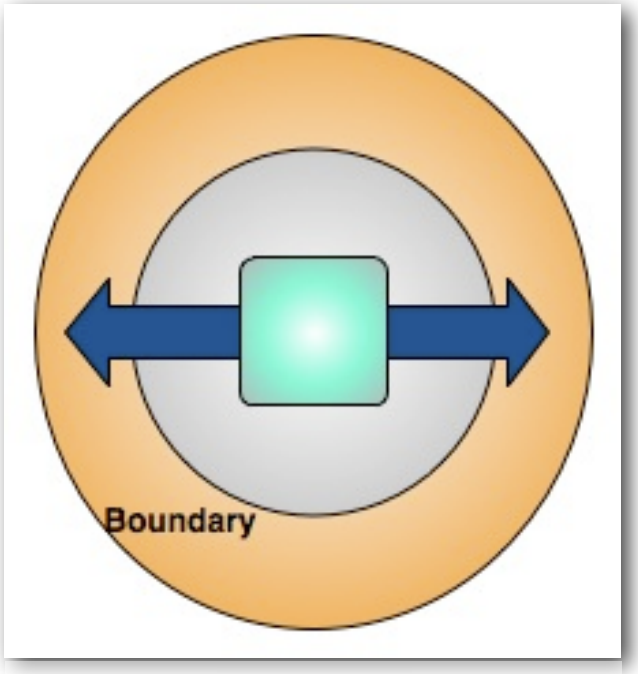
discourages embedded object creation

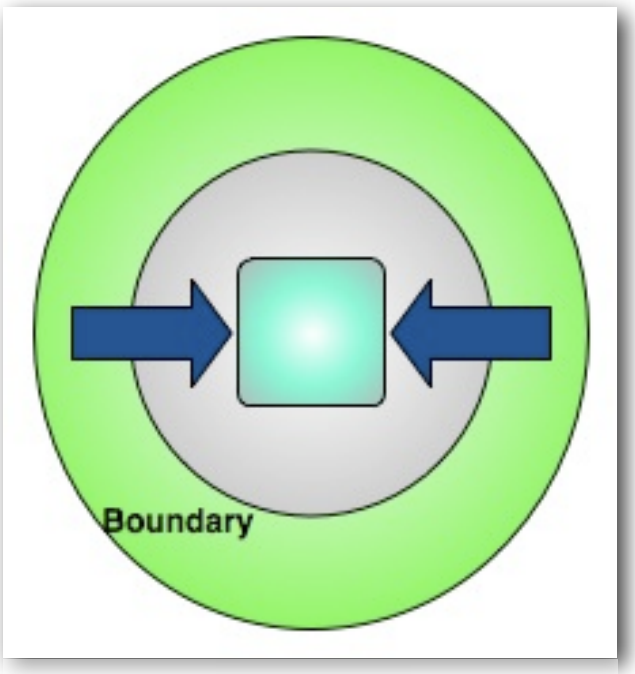


extroverted object



introverted object





extroverted objects

introverted objects

“reach out” to create objects

cleaner dependencies

scattered construction

moves object construction to a few simple places


unit tests

technology stack

java 5

junit 4.4

hamcrest matchers

```
assertThat(c.getFactors(), is(expected));
```


perfect
numbers

perfect number:

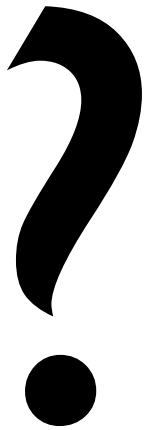
∑ of the factors == number
(not including the number)

∑ of the factors - # == #

```java
public class PerfectNumberFinder1 {
    public static boolean isPerfect(int number) {
        // get factors
        List<Integer> factors = new ArrayList<Integer>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i < number; i++)
            if (number % i == 0)
                factors.add(i);

        // sum factors
        int sum = 0;
        for (int n : factors)
            sum += n;

        // decide if it's perfect
        return sum - number == number;
    }
}
```

```
private static Integer[] PERFECT_NUMS = {6, 28, 496, 8128, 33550336};

@Test public void test_perfection() {
    for (int i : PERFECT_NUMS)
        assertTrue(PerfectNumberFinder1.isPerfect(i));
}

@Test public void test_non_perfection() {
    List<Integer> expected = new ArrayList<Integer>(
            Arrays.asList(PERFECT_NUMS));
    for (int i = 2; i < 100000; i++) {
        if (expected.contains(i))
            assertTrue(PerfectNumberFinder1.isPerfect(i));
        else
            assertFalse(PerfectNumberFinder1.isPerfect(i));
    }
}
```

```java
public static boolean isPerfect(int number) {
    // get factors
    List<Integer> factors = new ArrayList<Integer>();
    factors.add(1);
    factors.add(number);
    for (int i = 2; i < number; i++)
        if (number % i == 0)
            factors.add(i);

    // sum factors
    int sum = 0;
    for (int n : factors)
        sum += n;

    // decide if it's perfect
    return sum - number == number;
}
```



```
public class PerfectNumberFinder2 {
    public static boolean isPerfect(int number) {
        // get factors
        List<Integer> factors = new ArrayList<Integer>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i <= sqrt(number); i++)
            if (number % i == 0) {
                factors.add(i);
                factors.add(number / i);
            }

        // sum factors
        int sum = 0;
        for (int n : factors)
            sum += n;

        // decide if it's perfect
        return sum - number == number;
    }
}
```

whole number square roots

```
public class PerfectNumberFinder2 {
    public static boolean isPerfect(int number) {
        // get factors
        List<Integer> factors = new ArrayList<Integer>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i <= sqrt(number); i++)
            if (number % i == 0) {
                factors.add(i);
                // account for whole-number square roots
                if (number / i !=  i)
                    factors.add(number / i);
            }

        // sum factors
        int sum = 0;
        for (int n : factors)
            sum += n;

        // decide if it's perfect
        return sum - number == number;
    }
}
```

what's wrong?

the simplest thing?

`isPerfect()`?

go back to definition:

factors

is a number a factor

sum factors

first test

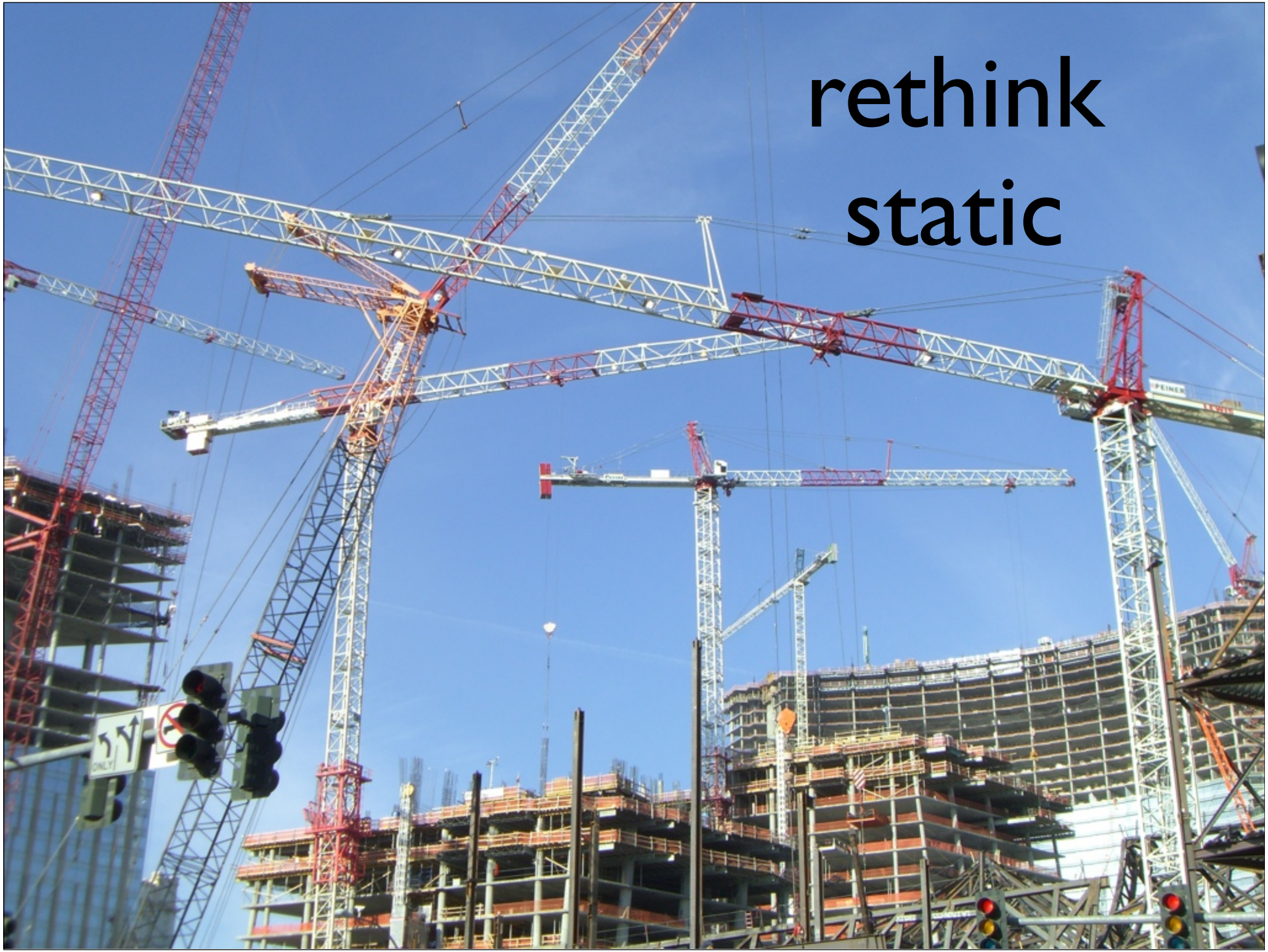
```
@Test public void is_factor() {
    assertTrue(Classifier1.isFactor(1, 10));
}
```

```
public class Classifier1 {
    public static boolean isFactor(int factor, int number) {
        return number % factor == 0;
    }
```

next test

```
@Test public void factors_for() {
    int[] expected = new int[] {1};
    assertThat(Classifier1.factorsFor(1), is(expected));
}
```

```
public static int[] factorsFor(int number) {
    return new int[] {number};
}
```


rethink
static

static tends to propagate

```
public class Classifier1 {
    public static boolean isFactor(int factor, int number) {
        return number % factor == 0;
    }

    public static int[] factorsFor(int number) {
        return new int[] {number};
    }
}
```

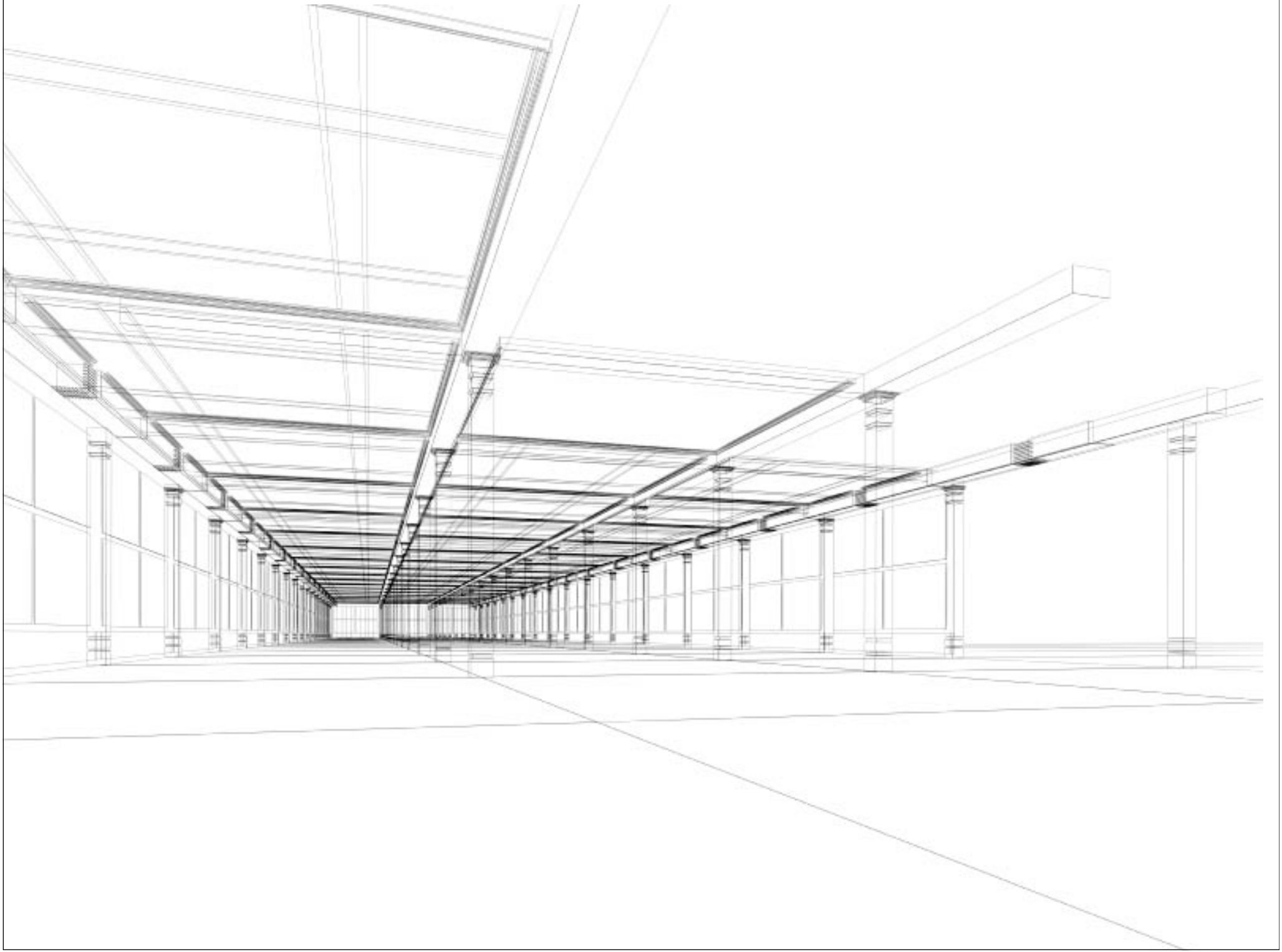
this code is starting to feel very procedural

take 2

```
@Test public void is_1_a_factor_of_10() {
    Classifier2 c = new Classifier2(10);
    assertTrue(c.isFactor(1));
}
```

```
public class Classifier2 {
    private int _number;

    public Classifier2(int number) {
        _number = number;
    }

    public boolean isFactor(int factor) {
        return _number % factor == 0;
    }
```

```
@Test public void is_1_a_factor_of_10() {
    Classifier2 c = new Classifier2(10);
    assertTrue(c.isFactor(1));
}

@Test public void is_5_a_factor_of_25() {
    Classifier2 c = new Classifier2(25);
    assertTrue(c.isFactor(5));
}

@Test public void is_3_not_a_factor_of_7() {
    Classifier2 c = new Classifier2(7);
    assertFalse(c.isFactor(3));
}
```



should we keep this test?

```
@Test public void is_1_a_factor_of_10() {
    Classifier2 c = new Classifier2(10);
    assertTrue(c.isFactor(1));
}
```


you've really
screwed up now!

writing insanely trivial tests allows you to get the interactions right

keep
trivial
tests as
canaries

next test

```
@Test public void factors_for_6() {
    int[] expected = new int[] {1, 2, 3, 6};
    Classifier2 c = new Classifier2(6);
    assertThat(c.getFactors(), is(expected));
}
```

yuck!

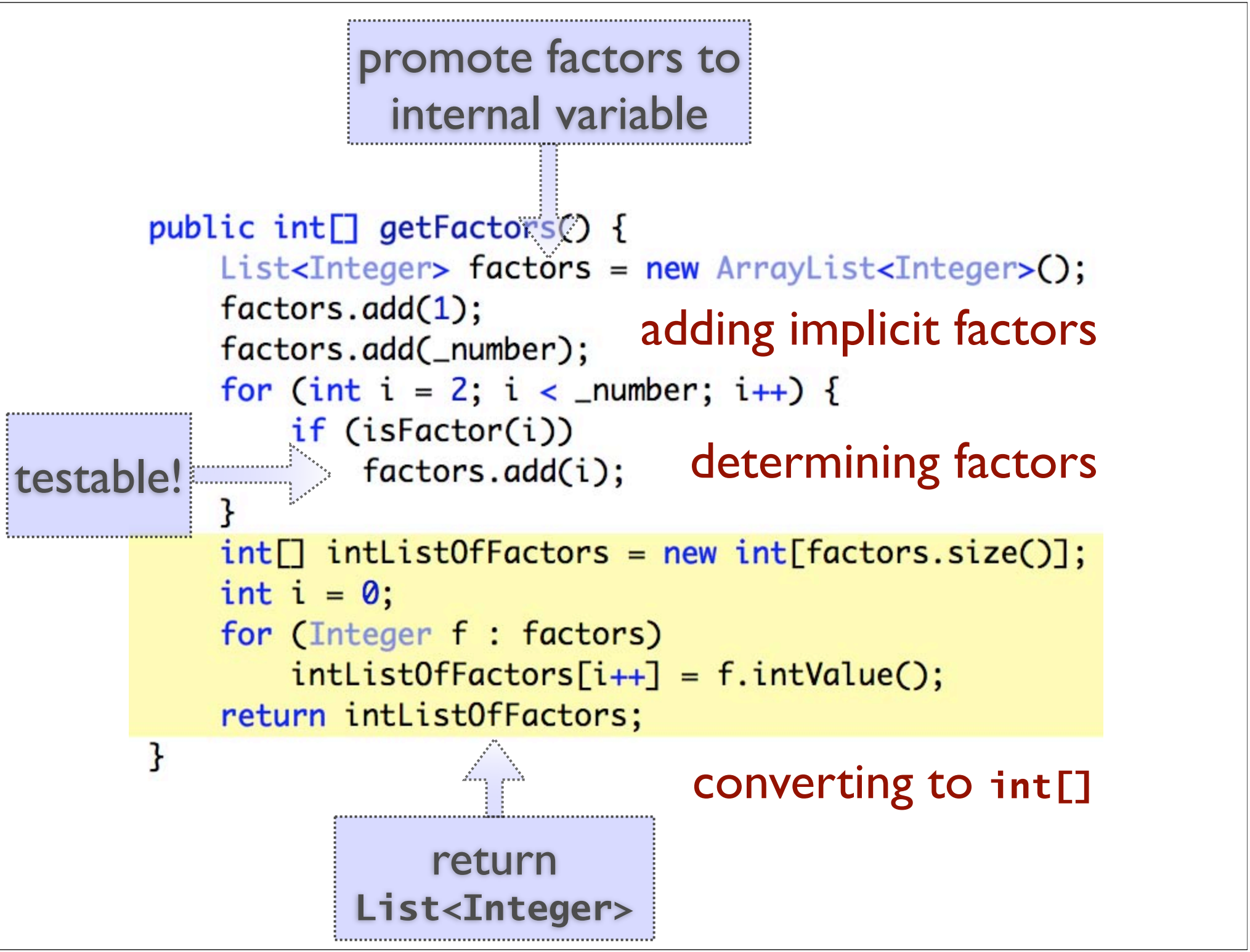
```
public int[] getFactors() {
    List<Integer> factors = new ArrayList<Integer>();
    factors.add(1);
    factors.add(_number);
    for (int i = 2; i < _number; i++) {
        if (isFactor(i))
            factors.add(i);
    }
    int[] intListOfFactors = new int[factors.size()];
    int i = 0;
    for (Integer f : factors)
        intListOfFactors[i++] = f.intValue();
    return intListOfFactors;
}
```

problems:

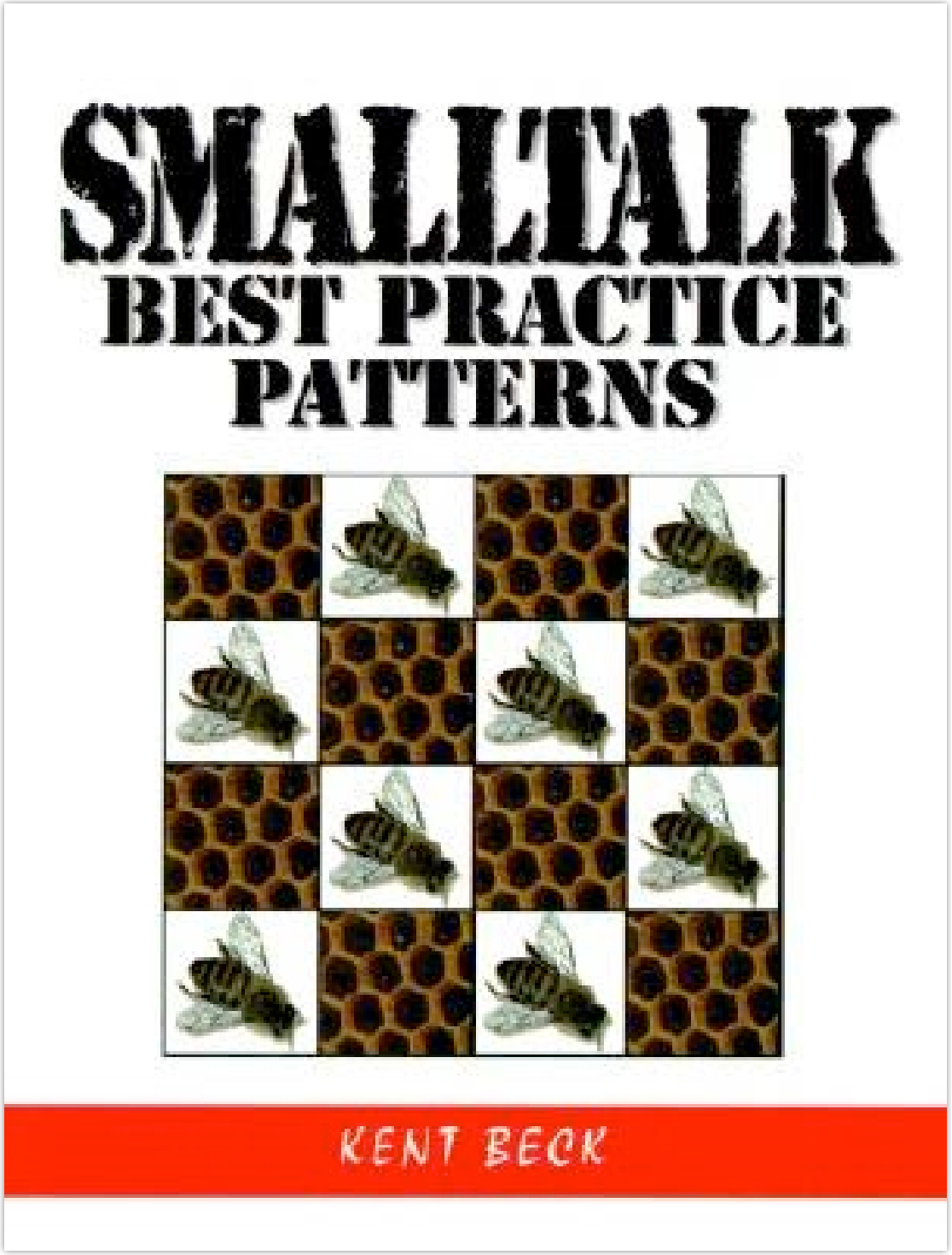
1. too much code!

2. switching from `List<Integer>` to `int[]` is noisy

3. more than one thing happening


promote factors to internal variable
public int[] getFactors() {
List<Integer> factors = new ArrayList<Integer>();
factors.add(1);
factors.add(_number);
for (int i = 2; i < _number; i++) {
if (isFactor(i))
factors.add(i);
}
int[] intListOfFactors = new int[factors.size()];
int i = 0;
for (Integer f : factors)
intListOfFactors[i++] = f.intValue();
return intListOfFactors;
}
adding implicit factors
testable!
determining factors
converting to int[]
return List<Integer>

composed method

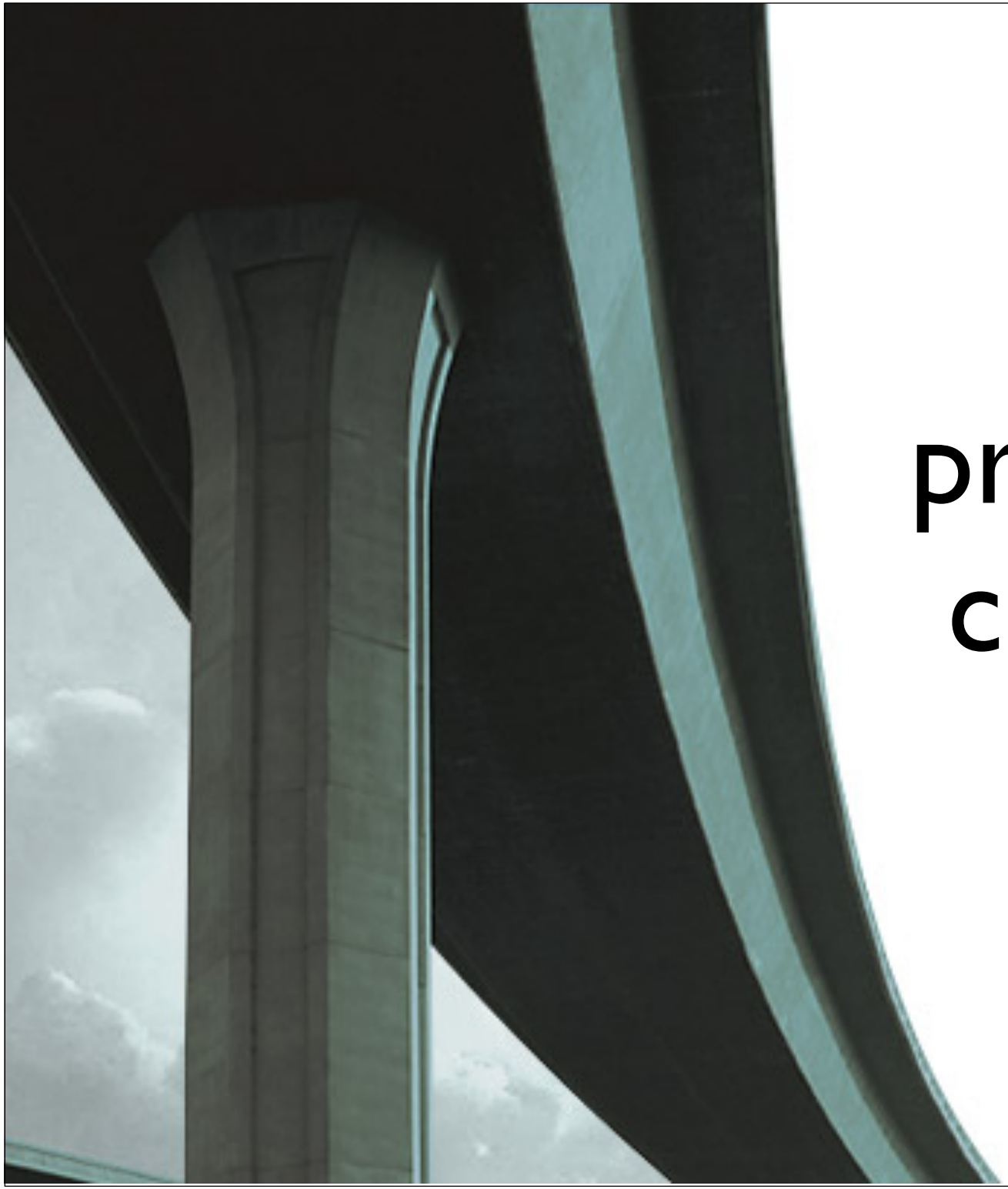


composed method

Divide your program into methods that perform one identifiable task.

Keep all of the operations in a method at the same level of abstraction.

This will naturally result in programs with many small methods, each a few lines long.



tdd
predisposes
composed
method

1 test generates lots of ?'s:

should factors be internal state?

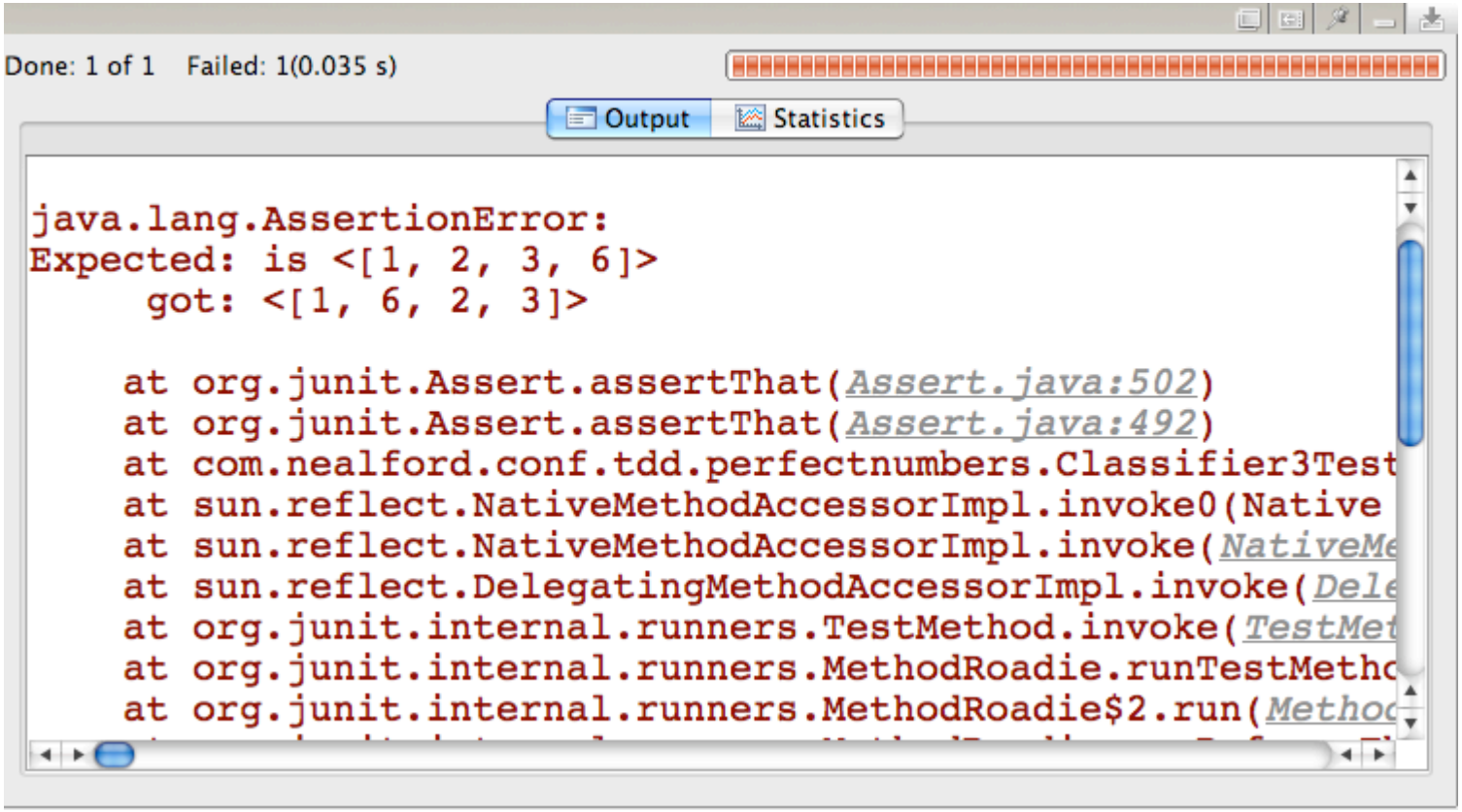
what should it return?


I test...
leads to another...
and another...

next test:
addFactors()

```
@Test public void add_factors() {
    Classifier3 c = new Classifier3(6);
    c.addFactor(2);
    c.addFactor(3);
    assertThat(c.getFactors(), is(Arrays.asList(1, 2, 3, 6)));
}
```

```
public class Classifier3 {
    private int _number;
    private List<Integer> _factors;

    public Classifier3(int number) {
        _number = number;
        _factors = new ArrayList<Integer>();
        this._factors.add(1);
        this._factors.add(number);
    }

    public boolean isFactor(int factor) {
        return _number % factor == 0;
    }

    public List<Integer> getFactors() {
        return _factors;
    }

    public void addFactor(int factor) {
        _factors.add(factor);
    }
}
```


Done: 1 of 1 Failed: 1(0.035 s)
Output
Statistics
java.lang.AssertionError:
Expected: is <[1, 2, 3, 6]>
got: <[1, 6, 2, 3]>
at org.junit.Assert.assertThat(Assert.java:502)
at org.junit.Assert.assertThat(Assert.java:492)
at com.nealford.conf.tdd.perfectnumbers.Classifier3Test
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMe
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Dele
at org.junit.internal.runners.TestMethod.invoke(TestMet
at org.junit.internal.runners.MethodRoadie.runTestMetho
at org.junit.internal.runners.MethodRoadie$2.run(Method

rethink the abstraction:

`java.util.List` is ordered

are factors ordered?

looking for the *set* of factors

refactor!

```
@Test public void add_factors() {
    Set<Integer> expected =
            new HashSet(Arrays.asList(1, 2, 3, 6));
    Classifier4 c = new Classifier4(6);
    c.addFactor(2);
    c.addFactor(3);
    assertThat(c.getFactors(), is(expected));
}
```

next: `calculateFactors()`

```
@Test public void factors_for_6() {
    Set<Integer> expected =
            new HashSet(Arrays.asList(1, 2, 3, 6));
    Classifier4 c = new Classifier4(6);
    c.calculateFactors();
    assertThat(c.getFactors(), is(expected));
}

public void calculateFactors() {
    for (int i = 2; i < _number; i++)
        if (isFactor(i))
            addFactor(i);
}
```

ok for tests to be moist...


...but not drenched

refactor

```
private Set<Integer> expectationSetWith(Integer... numbers) {
    return new HashSet<Integer>(Arrays.asList(numbers));
}
```

```
@Test public void factors_for_6() {
    Set<Integer> expected = expectationSetWith(1, 2, 3, 6);
    Classifier4 c = new Classifier4(6);
    c.calculateFactors();
    assertThat(c.getFactors(), is(expected));
}
```

more tests

```
@Test public void factors_for_12() {
    Classifier4 c = new Classifier4(12);
    c.calculateFactors();
    assertThat(c.getFactors(),
            is(expectationSetWith(1, 2, 3, 4, 6, 12)));
}

@Test public void factors_for_100() {
    Classifier4 c = new Classifier4(100);
    c.calculateFactors();
    assertThat(c.getFactors(),
            is(expectationSetWith(1, 100, 2, 50, 4, 25, 5, 20, 10)));
}
```

boundary conditions

```
@Test public void factors_for_1() {
    Classifier5 c = new Classifier5(1);
    c.calculateFactors();
    assertThat(c.getFactors(), is(expectationSetWith(1)));
}

@Test(expected = InvalidNumberException.class)
public void cannot_classify_negative_numbers() {
    new Classifier5(-20);
}

@Test public void factors_for_max_int() {
    Classifier5 c = new Classifier5(Integer.MAX_VALUE);
    c.calculateFactors();
    assertThat(c.getFactors(), is(expectationSetWith(1, 2147483647)));
}
```

consumption awareness encourages exploration of boundary conditions

```
public class Classifier5 {
    private Set<Integer> _factors;
    private int _number;
    private static final String NEGATIVE_NUMBER_ERROR =
            "Can't classify negative numbers";

    public Classifier5(int number) {
        if (number < 1)
            throw new InvalidNumberException(NEGATIVE_NUMBER_ERROR);
        _number = number;
        _factors = new HashSet<Integer>();
        _factors.add(1);
        _factors.add(_number);
    }
```

next test: `sumOfFactors()`

```
@Test public void sum() {
    Classifier5 c = new Classifier5(20);
    c.calculateFactors();
    int expected = 1 + 2 + 4 + 5 + 10 + 20;
    assertThat(c.sumOfFactors(), is(expected));
}

public int sumOfFactors() {
    int sum = 0;
    for (int i : _factors)
        sum += i;
    return sum;
}
```

next test: perfection!

```
@Test public void perfection() {
    int[] perfectNumbers =
        new int[] {6, 28, 496, 8128, 33550336};
    for (int number : perfectNumbers)
        assertTrue(classifierFor(number).isPerfect());
}
```

```
public boolean isPerfect() {
    calculateFactors();
    return sumOfFactors() - _number == _number;
}
```


what about
a negative test?

```
private static final Integer[] PERFECT_NUMS =
        {6, 28, 496, 8128, 33550336};


@Test public void test_a_bunch_of_numbers() {
    Set<Integer> expected = new HashSet<Integer>(
            Arrays.asList(PERFECT_NUMS));
    for (int i = 2; i < 33550340; i++) {
        if (expected.contains(i))
            assertTrue(classifierFor(i).isPerfect());
        else
            assertFalse(classifierFor(i).isPerfect());
    }
}
```

optimize

now!

unoptimized

```
public void calculateFactors() {
    for (int i = 2; i < _number; i++)
        if (isFactor(i))
            addFactor(i);
}

public void addFactor(int factor) {
    _factors.add(factor);
}
```

optimized

```
public void calculateFactors() {
    for (int i = 2; i < sqrt(_number) + 1; i-
        if (isFactor(i))
            addFactor(i);
}

public void addFactor(int factor) {
    _factors.add(factor);
    _factors.add(_number / factor);
}
```

design implications

remember this?

```
for (int i = 2; i <= sqrt(number); i++)
    if (number % i == 0) {
        factors.add(i);
        // account for whole-number square roots
        if (number / i !=  i)
            factors.add(number / i);
    }
```

vs.

perfect number finder

```
for (int i = 2; i <= sqrt(number); i++)
    if (number % i == 0) {
        factors.add(i);
        // account for whole-number square roots
        if (number / i !=  i)
            factors.add(number / i);
    }
```

classifier

```
private void calculateFactors() {
    for (int i = 2; i < sqrt(_number) + 1; i++)
        if (isFactor(i))
            addFactor(i);
}

private void addFactor(int factor) {
    _factors.add(factor);
    _factors.add(_number / factor);
}
```

tdd vs test-after

test after doesn't expose design flaws early

because you think at the wrong abstraction level

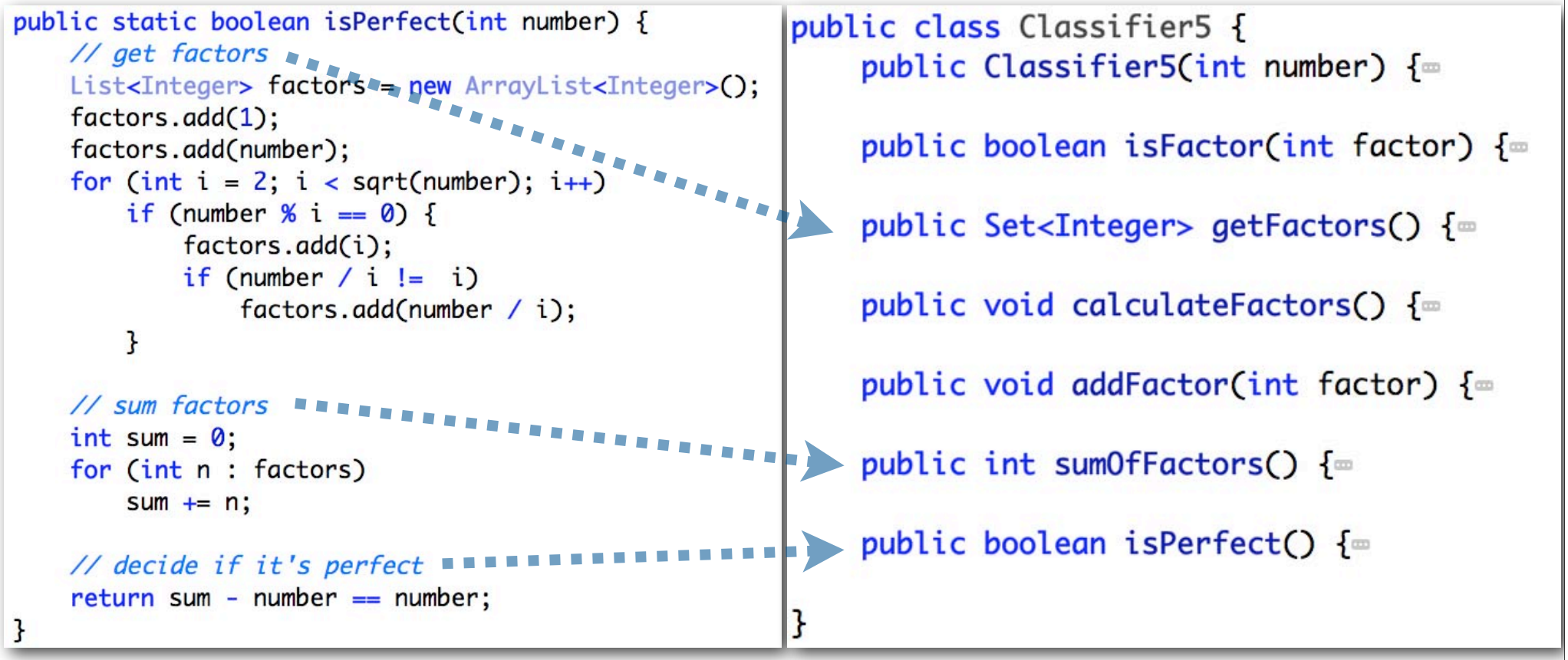
tdd forces you to think about every little thing

and encourages refactoring what's not right

if its hard to test,
something's wrong

```
public class Classifier5 {
    private Set<Integer> _factors;
    private int _number;
    private static final String NEGATIVE_NUMBER_ERROR =
            "Can't classify negative numbers";

    public Classifier5(int number) {…

    public boolean isFactor(int factor) {…

    public Set<Integer> getFactors() {…

    public void calculateFactors() {…

    public void addFactor(int factor) {…

    public int sumOfFactors() {…

    public boolean isPerfect() {…

}
```

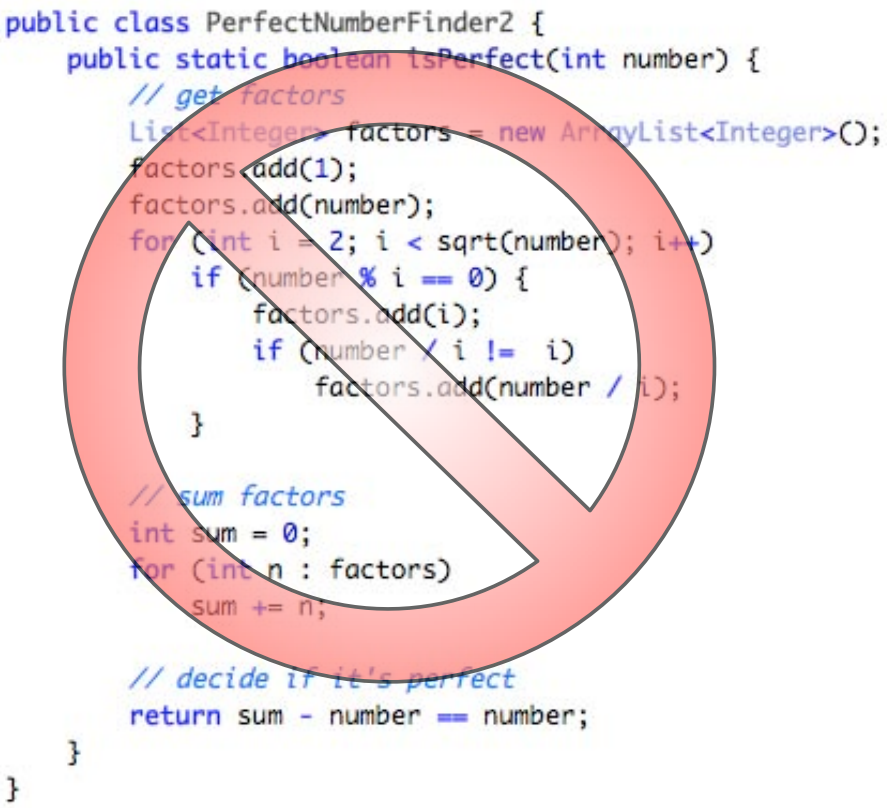
t d d
generates
cohesive
methods



refactor comments to methods

new requirement:

handle abundant & deficient numbers



```
public class PerfectNumberFinder2 {
    public static boolean isPerfect(int number) {
        // get factors
        List<Integer> factors = new ArrayList<Integer>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i < sqrt(number); i++)
            if (number % i == 0) {
                factors.add(i);
                if (number / i != i)
                    factors.add(number / i);
            }

        // sum factors
        int sum = 0;
        for (int n : factors)
            sum += n;

        // decide if it's perfect
        return sum - number == number;
    }
}
```

```
public class Classifier6 {
    private Set<Integer> _factors;
    private int _number;

    public Classifier6(int number) {…

    private boolean isFactor(int factor) {…

    public Set<Integer> getFactors() {…

    private void calculateFactors() {…

    private void addFactor(int factor) {…

    private int sumOfFactors() {…

    public boolean isPerfect() {…

    public boolean isAbundant() {…

    public boolean isDeficient() {…
}
```

abundant & deficient

```
public boolean isAbundant() {
    calculateFactors();
    return sumOfFactors() - _number > _number;
}

public boolean isDeficient() {
    calculateFactors();
    return sumOfFactors() - _number < _number;
}
```

composed method
yields reusable code

Classifier

```
public class Classifier6 {
    private Set<Integer> _factors;
    private int _number;

    public Classifier6(int number) {
        if (number < 1)
            throw new InvalidNumberException(
            "Can't classify negative numbers");
        _number = number;
        _factors = new HashSet<Integer>();
        _factors.add(1);
        _factors.add(_number);
    }

    private boolean isFactor(int factor) {
        return _number % factor == 0;
    }
```

```
public Set<Integer> getFactors() {
    return _factors;
}

private void calculateFactors() {
    for (int i = 2; i < sqrt(_number) + 1; i++)
        if (isFactor(i))
            addFactor(i);
}

private void addFactor(int factor) {
    _factors.add(factor);
    _factors.add(_number / factor);
}

private int sumOfFactors() {
    int sum = 0;
    for (int i : _factors)
        sum += i;
    return sum;
}
```

```
public boolean isPerfect() {
    calculateFactors();
    return sumOfFactors() - _number == _number;
}

public boolean isAbundant() {
    calculateFactors();
    return sumOfFactors() - _number > _number;
}

public boolean isDeficient() {
    calculateFactors();
    return sumOfFactors() - _number < _number;
}
```

complete tests

```
public class Classifier6Test {
    private static final Integer[] PERFECT_NUMS =
            {6, 28, 496, 8128, 33550336};

    @Test
    public void is_1_a_factor_of_10() {
        Classifier6 c = new Classifier6(10);
        assertTrue(isFactor(c, 1));
    }

    @Test public void is_5_a_factor_of_25() {
        Classifier6 c = new Classifier6(25);
        assertTrue(isFactor(c, 5));
    }
```

```
@Test public void is_3_not_a_factor_of_7() {
    Classifier6 c = new Classifier6(7);
    assertFalse(isFactor(c, 3));
}

@Test public void add_factors() {
    Set<Integer> expected =
            new HashSet<Integer>(Arrays.asList(1, 2, 3, 6));
    Classifier6 c = new Classifier6(6);
    addFactor(c, 2);
    addFactor(c, 3);
    assertThat(c.getFactors(), is(expected));
}

@Test public void factors_for_6() {
    Set<Integer> expected = expectationSetWith(1, 2, 3, 6);
    Classifier6 c = new Classifier6(6);
    calculateFactors(c);
    assertThat(c.getFactors(), is(expected));
}
```

```
@Test public void factors_for_12() {
    Classifier6 c = new Classifier6(12);
    calculateFactors(c);
    assertThat(c.getFactors(),
            is(expectationSetWith(1, 2, 3, 4, 6, 12)));
}

@Test public void factors_for_100() {
    Classifier6 c = new Classifier6(100);
    calculateFactors(c);
    assertThat(c.getFactors(),
            is(expectationSetWith(1, 100, 2, 50, 4, 25, 5, 20, 10)));
}

@Test public void factors_for_prime() {
    Classifier6 c = new Classifier6(23);
    calculateFactors(c);
    assertThat(c.getFactors(), is(expectationSetWith(1, 23)));
}
```

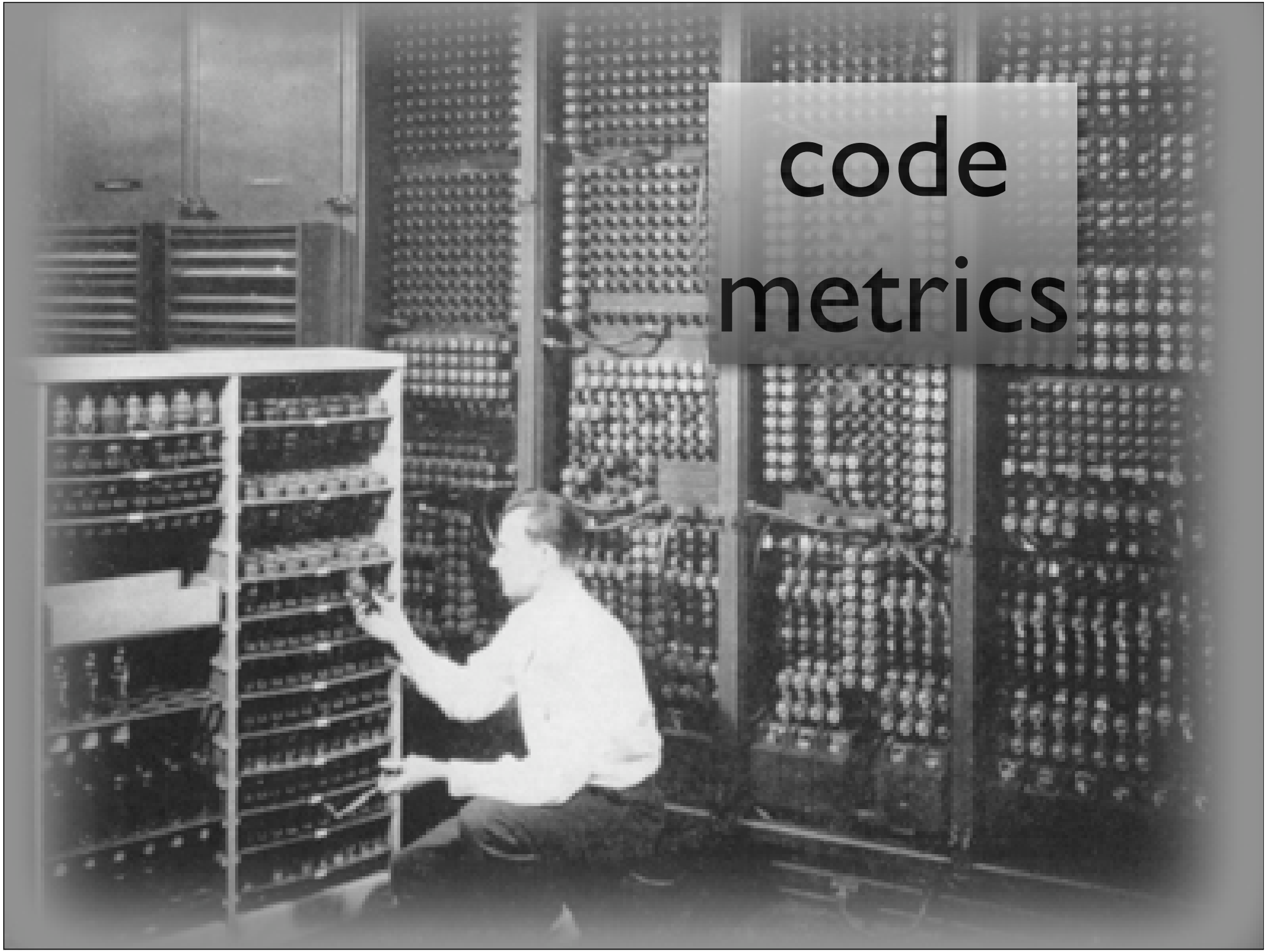
```
@Test public void test_a_bunch_of_numbers() {
    Set<Integer> expected = new HashSet<Integer>(
            Arrays.asList(PERFECT_NUMS));
    for (int i = 2; i < 33550340; i++) {
        if (expected.contains(i))
            assertTrue(classifierFor(i).isPerfect());
        else
            assertFalse(classifierFor(i).isPerfect());
    }
}

@Test(expected = InvalidNumberException.class)
public void cannot_classify_negative_numbers() {
    new Classifier6(-20);
}

@Test public void sum() {
    Classifier6 c = new Classifier6(20);
    calculateFactors(c);
    int expected = 1 + 2 + 4 + 5 + 10 + 20;
    assertThat(sumOfFactors(c), is(expected));
}
```

```java
@Test public void perfection() {
    Integer[] perfectNumbers = PERFECT_NUMS;
    for (int number : perfectNumbers)
        assertTrue(classifierFor(number).isPerfect());
}

@Test public void abundance() {
    int[] abundantNumbers = new int[] {12, 18, 20, 24, 30};
    for (int i : abundantNumbers)
        assertTrue(new Classifier6(i).isAbundant());
}

@Test public void deficicency() {
    int[] deficientNumbers = new int[] {3, 5, 7, 9, 10, 11};
    for (int i : deficientNumbers) {
        assertTrue(new Classifier6(i).isDeficient());
    }
}
```

helpers

```
private Set<Integer> expectationSetWith(Integer... numbers) {
    return new HashSet<Integer>(Arrays.asList(numbers));
}

private Classifier6 classifierFor(int number) {
    Classifier6 c = new Classifier6(number);
    calculateFactors(c);
    return c;
}
```


measuring code
quality
ERROR

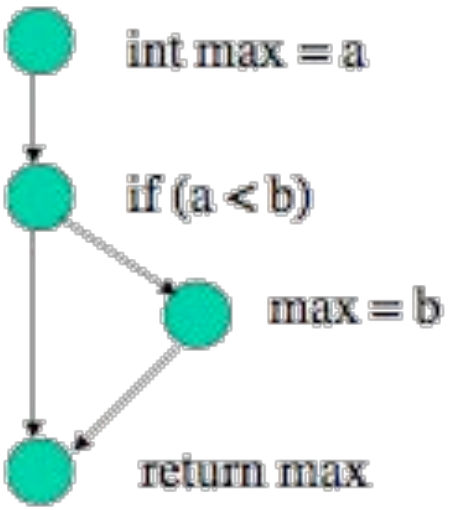

code
metrics

cyclomatic complexity

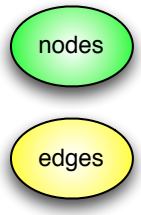
measures method complexity

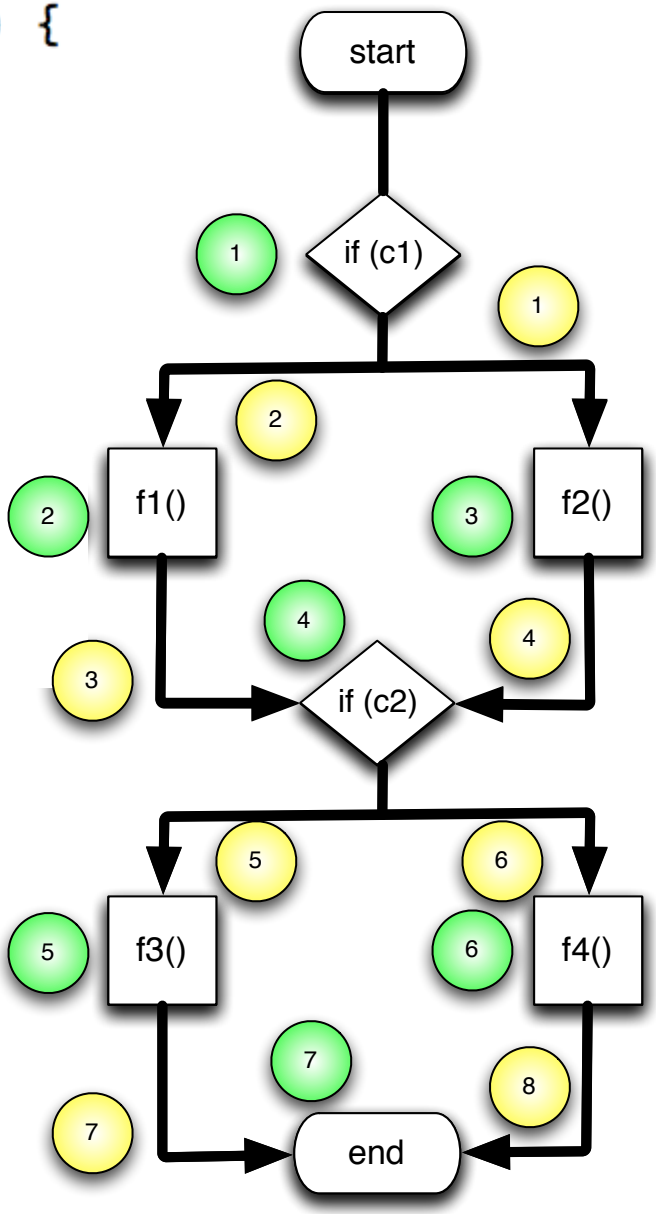
```
V(G)= e - n + 2
V(G) = cyclomatic complexity of G
e= # edges
n= # of nodes
```

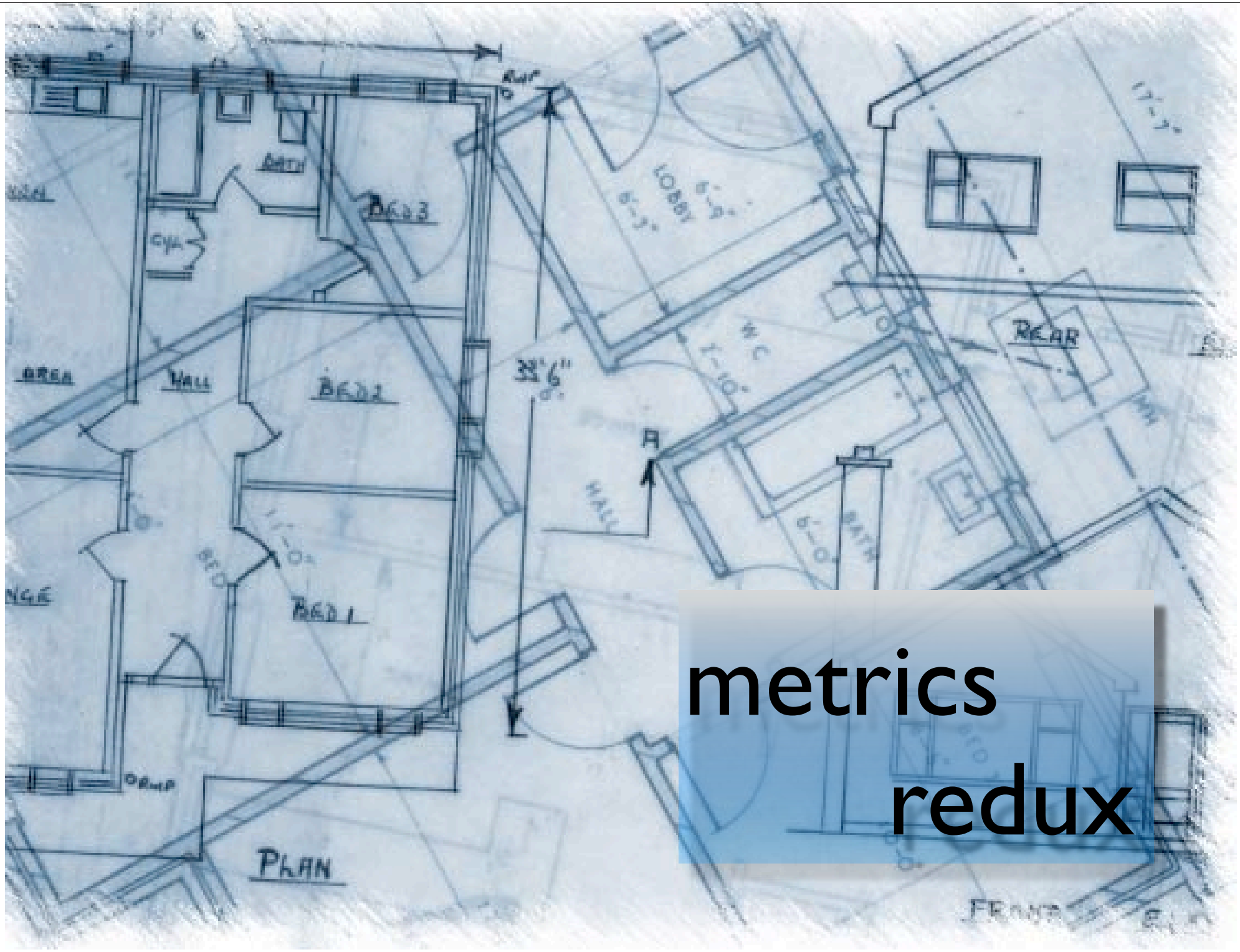
```
int max (int a, int b) {
    int max = a;
    if (a < b) {
        max = b;
    }
    return max;
}
```

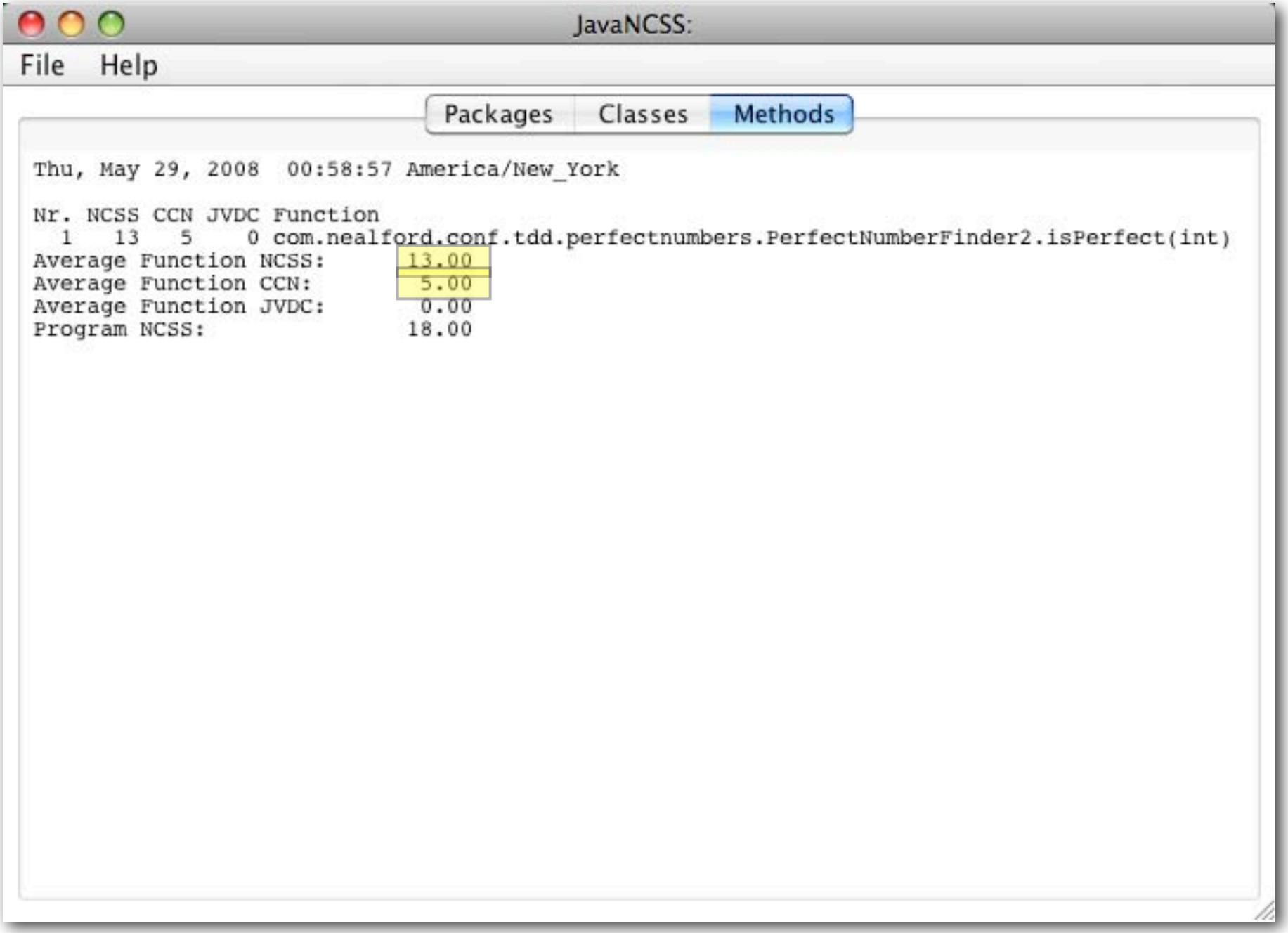


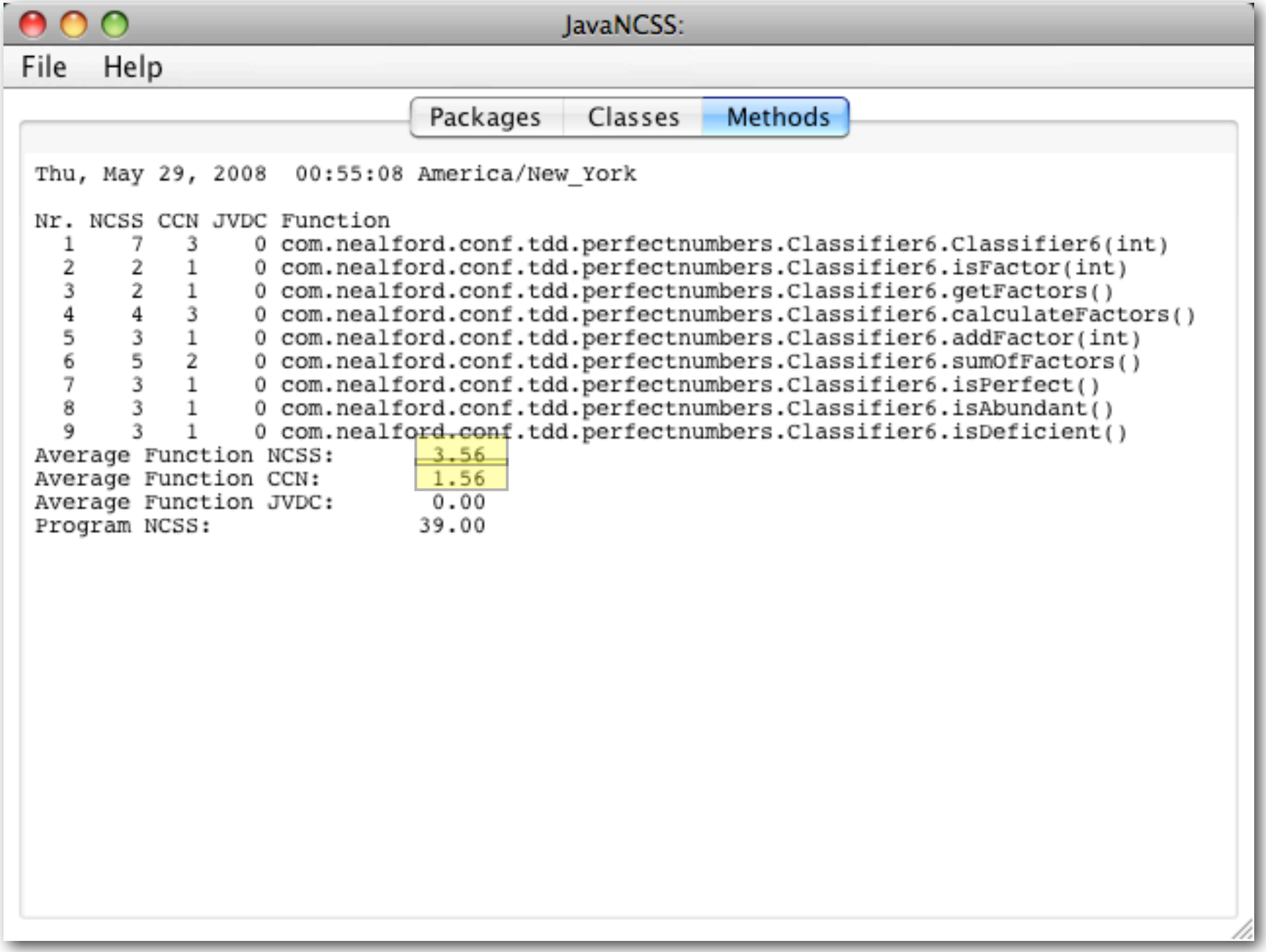
```
public void doIt() {
    if (c1) {
        f1();
    } else {
        f2();
    }
    if (c2) {
        f3();
    } else {
        f4();
    }
}
```






metrics
redux


JavaNCSS:
File Help
Packages Classes Methods
Thu, May 29, 2008 00:58:57 America/New_York
Nr. NCSS CCN JVDC Function
1 13 5 0 com.nealford.conf.tdd.perfectnumbers.PerfectNumberFinder2.isPerfect(int)
Average Function NCSS: 13.00
Average Function CCN: 5.00
Average Function JVDC: 0.00
Program NCSS: 18.00

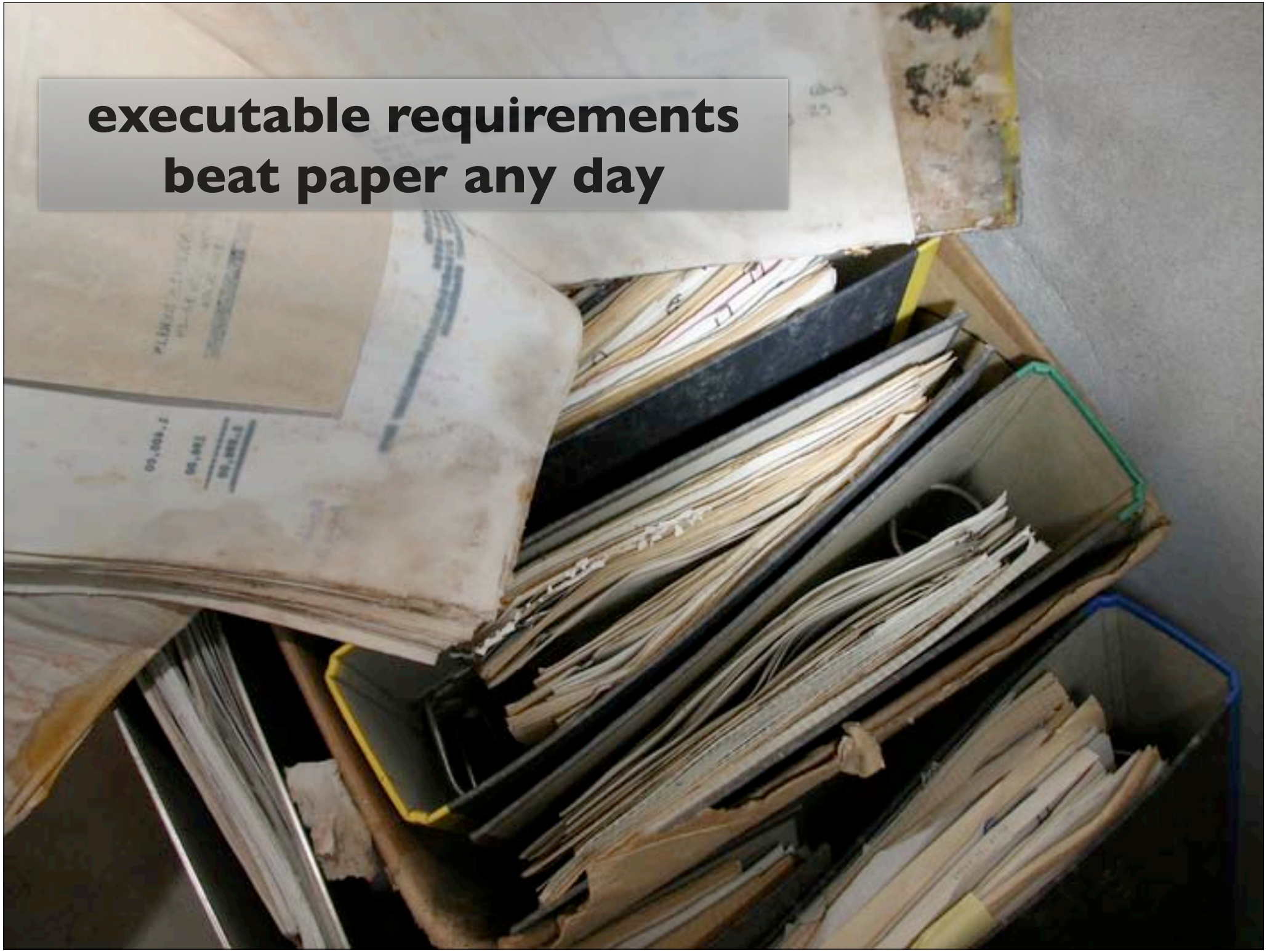

JavaNCSS:
File
Help
Packages
Classes
Methods
Thu, May 29, 2008 00:55:08 America/New_York
Nr. NCSS CCN JVDC Function
1 7 3 0 com.nealford.conf.tdd.perfectnumbers.Classifier6.Classifier6(int)
2 2 1 0 com.nealford.conf.tdd.perfectnumbers.Classifier6.isFactor(int)
3 2 1 0 com.nealford.conf.tdd.perfectnumbers.Classifier6.getFactors()
4 4 3 0 com.nealford.conf.tdd.perfectnumbers.Classifier6.calculateFactors()
5 3 1 0 com.nealford.conf.tdd.perfectnumbers.Classifier6.addFactor(int)
6 5 2 0 com.nealford.conf.tdd.perfectnumbers.Classifier6.sumOfFactors()
7 3 1 0 com.nealford.conf.tdd.perfectnumbers.Classifier6.isPerfect()
8 3 1 0 com.nealford.conf.tdd.perfectnumbers.Classifier6.isAbundant()
9 3 1 0 com.nealford.conf.tdd.perfectnumbers.Classifier6.isDeficient()
Average Function NCSS: 3.56
Average Function CCN: 1.56
Average Function JVDC: 0.00
Program NCSS: 39.00

are these number sustainable for a large project?

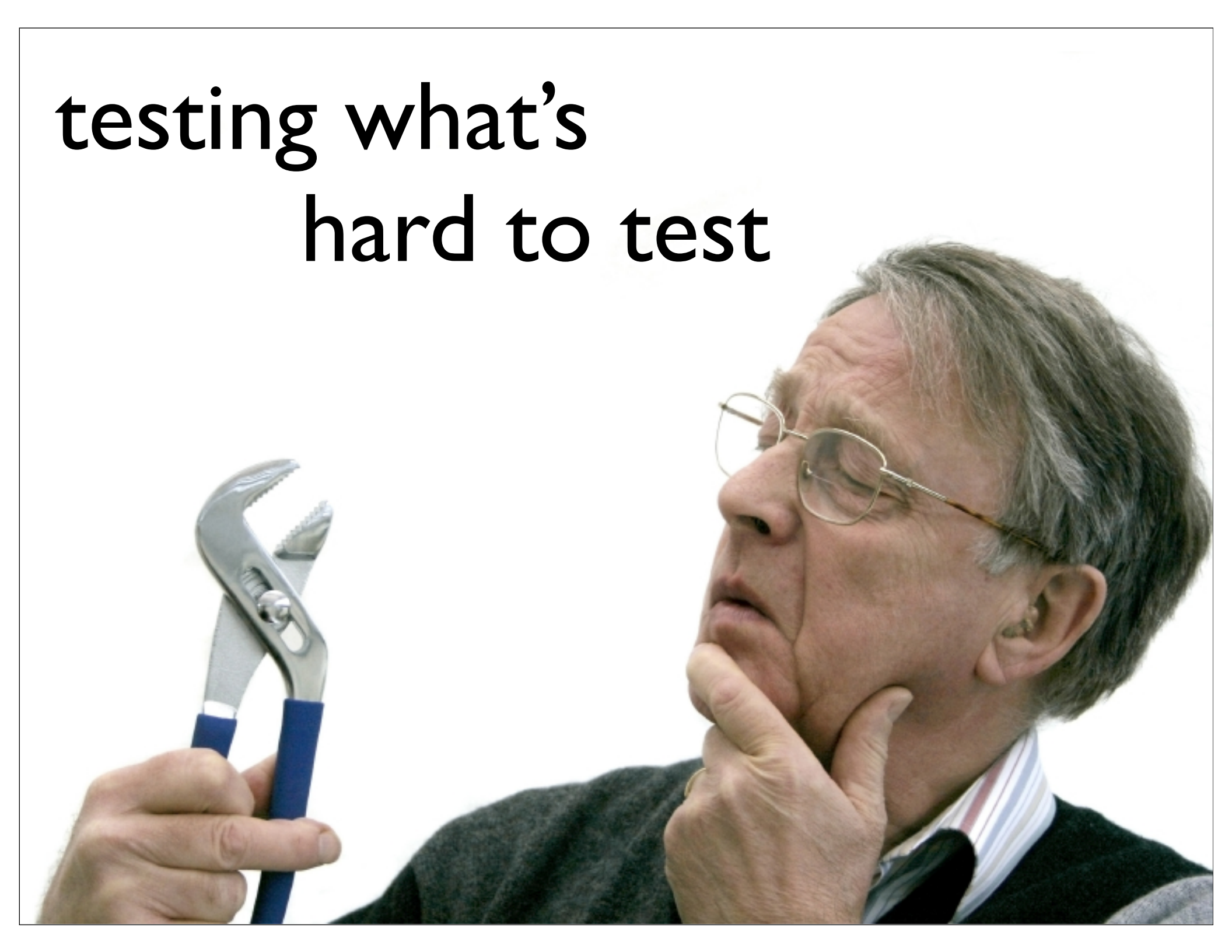
stay tuned!


tests
as
documentation

```
@Test public void is_1_a_factor_of_10() {…

@Test public void is_5_a_factor_of_25() {…

@Test public void is_3_not_a_factor_of_7() {…

@Test public void add_factors() {…

@Test public void factors_for_6() {…

@Test public void factors_for_12() {…

@Test public void factors_for_100() {…

@Test public void factors_for_1() {…

@Test(expected = InvalidNumberException.class)
public void cannot_classify_negative_numbers() {…

@Test public void factors_for_max_int() {…

@Test public void sum() {…

@Test public void perfection() {…
```


executable requirements
beat paper any day

testing what's hard to test



tdd private methods?

```
private void calculateFactors() {
    for (int i = 2; i < sqrt(_number) + 1; i++)
        if (isFactor(i))
            addFactor(i);
}

private void addFactor(int factor) {
    _factors.add(factor);
    _factors.add(_number / factor);
}

private int sumOfFactors() {
    int sum = 0;
    for (int i : _factors)
        sum += i;
    return sum;
}
```

solution #1:

make them all public
(or package) scope

solution #2:

use reflection

reflection helpers

```
@Test
public void is_1_a_factor_of_10() {
    Classifier6 c = new Classifier6(10);
    assertTrue(isFactor(c, 1));
}

private boolean isFactor(Classifier6 c, int factor) {
    try {
        Method m = Classifier6.class.getDeclaredMethod(
                "isFactor", int.class);
        m.setAccessible(true);
        return (Boolean) m.invoke(c, factor);
    } catch (Throwable t) {
        fail();
    }
    return false;
}
```


groovy

reflection

```
@Test public void is_factor_via_reflection() {
  def m = Classifier6.class.getDeclaredMethod("isFactor", int.class)
  m.accessible = true
  assertTrue m.invoke(new Classifier6(10), 10)
  assertTrue m.invoke(new Classifier6(25), 5)
  assertFalse m.invoke(new Classifier6(25), 6)
}
```

dirty secret: private is ignored!

```
@Test public void is_factor() {
    assertTrue new Classifier6(10).isFactor(1)
    assertTrue new Classifier6(25).isFactor(5)
    assertFalse new Classifier6(25).isFactor(6)
}
```

technically, a bug...

...no great hurry to fix it (insanely useful!)


mocking


mocking in
JAVA™

technology stack

java 5

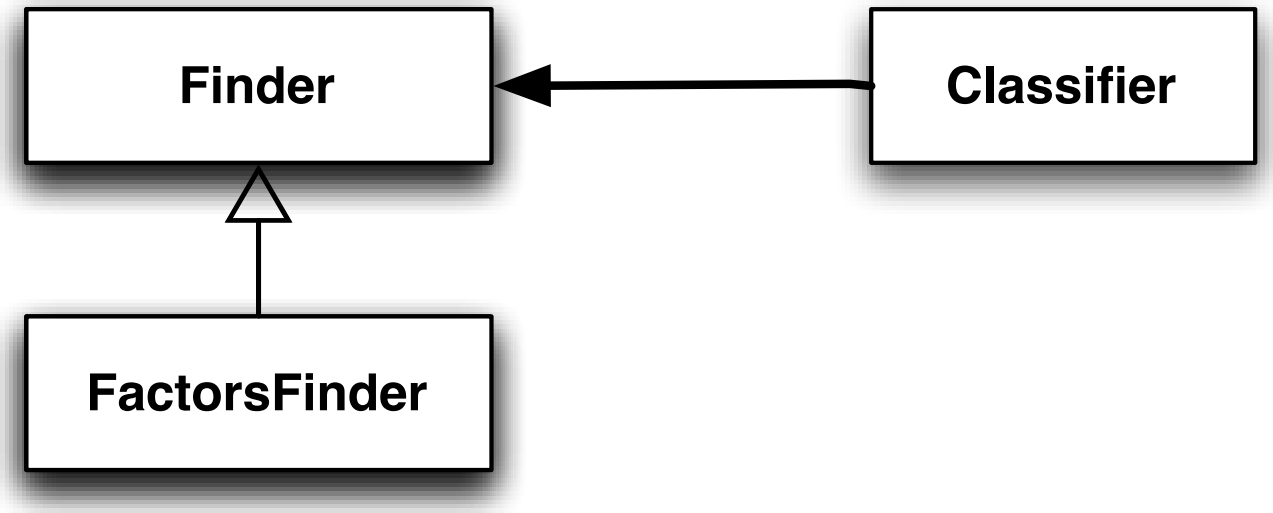
junit 4.x

hamcrest matchers

jmock 2.4

changes:

added a `FactorsFinder` class which harvests factors



refactored `Classifier` to use external factors

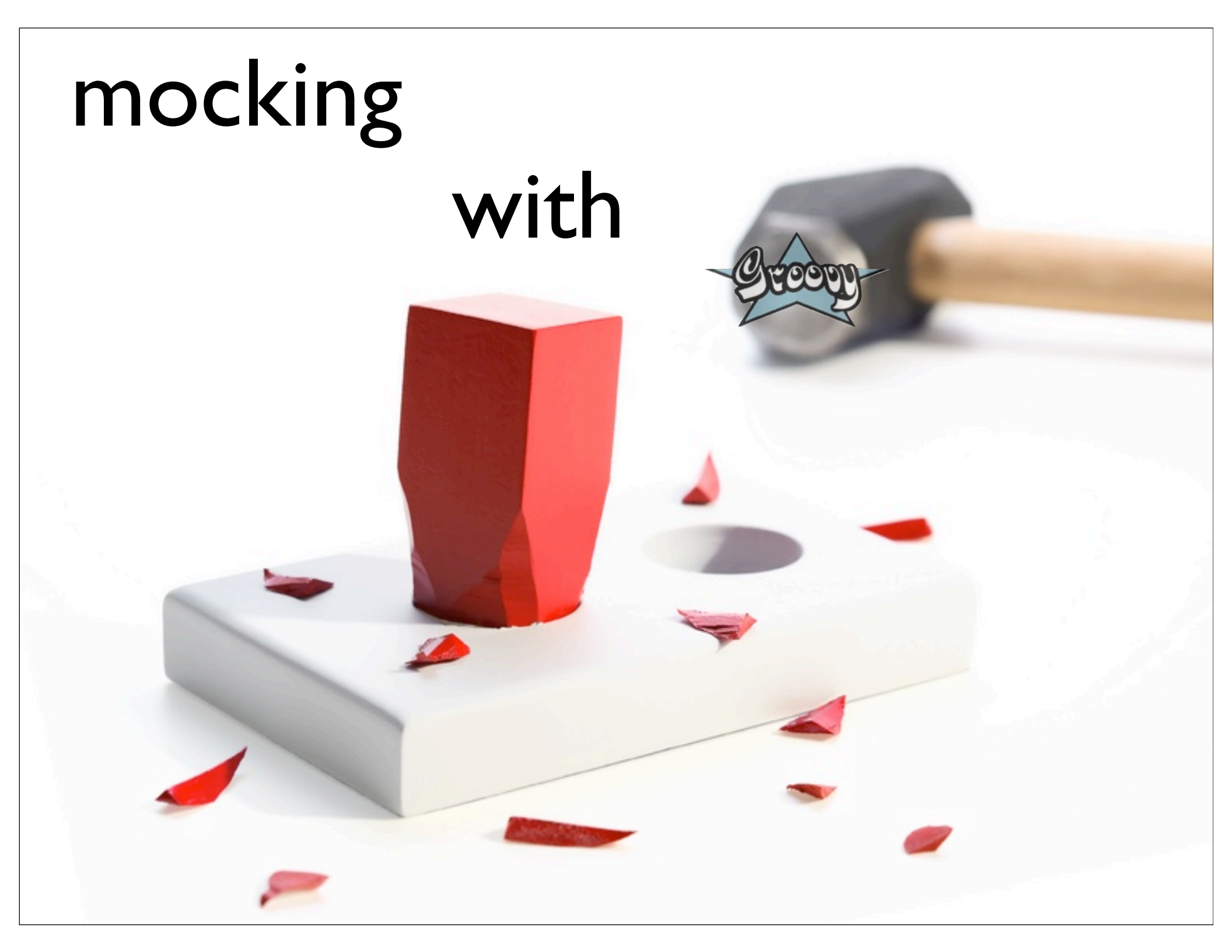
```
public class Classifier7 {
    private int _number;
    private Finder _factors;

    public Classifier7(int number, Finder factors) {
        _number = number;
        _factors = factors;
    }

    public boolean isPerfect() {
        return sumOfFactors() - _number == _number;
    }

    public int sumOfFactors() {
        int sum = 0;
        for (int i : _factors.factors())
            sum += i;
        return sum;
    }
}
```

```
public class FactorsFinder implements Finder {
    private int _number;
    private Set<Integer> _factors = new HashSet<Integer>();

    public FactorsFinder(int number) {
        _number = number;
    }

    public Set<Integer> factors() {
        calculateFactors();
        return _factors;
    }

    private boolean isFactor(int factor) {
        return _number % factor == 0;
    }

    public void calculateFactors() {
        for (int i = 2; i < sqrt(_number) + 1; i++)
            if (isFactor(i))
                addFactor(i);
    }

    private void addFactor(int factor) {
        _factors.add(factor);
        _factors.add(_number / factor);
    }
}
```

jmock


```
@RunWith(JMock.class)
public class ClassifierWithMockTest {
    Mockery context = new JUnit4Mockery() {{
        setImposteriser(ClassImposteriser.INSTANCE);
    }};

    @Test public void external_factors() {
        final Finder facts = context.mock(Finder.class);
        Classifier7 c = new Classifier7(42, facts);
        final Set<Integer> expected =
                new HashSet(Arrays.asList(1, 2, 3, 6, 7, 21, 14, 42));
        context.checking(new Expectations() {{
            one(facts).factors(); will(returnValue(expected));
        }});
        assertThat(c.sumOfFactors(), is(1 + 2 + 3 + 6 + 7 + 21 + 14 + 42));
        context.assertIsSatisfied();
    }
}
```

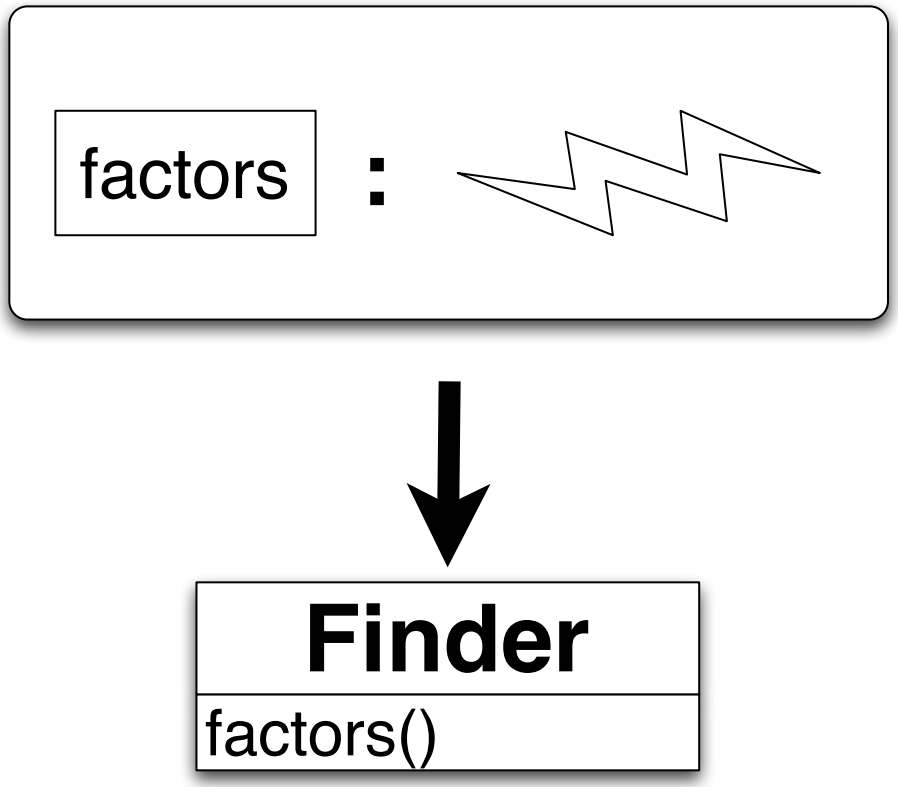
mocking with



java-like syntax

```
@Test public void test_mock_with_simple_syntax() {
  Set expected = new HashSet(Arrays.asList(1, 2, 3, 6, 7, 21, 14, 42))
  def facts = [factors: {expected}] as Finder
  Classifier7 c = new Classifier7(42, facts)
  def sumOfFactors = 0
  expected.each { num ->
    sumOfFactors += num
  }
  assertThat c.sumOfFactors(), is(sumOfFactors)
}
```

```
def facts = [factors: {expected}] as Finder
```



idiomatic


```
@Test public void test_mock_with_simple_syntax() {
  Set expected = new HashSet(Arrays.asList(1, 2, 3, 6, 7, 21, 14, 42))
  def facts = [factors: {expected}] as Finder
  Classifier7 c = new Classifier7(42, facts)
  def sumOfFactors = 0
  expected.each { num ->
    sumOfFactors += num
  }
  assertThat c.sumOfFactors(), is(sumOfFactors)
}
```

```
@Test public void test_mock_with_factors_finder() {
  def expected = [1, 2, 3, 6, 7, 21, 14, 42]
  def facts = [factors: {expected as Set}] as Finder
  Classifier7 c = new Classifier7ssifier7(42, facts)
  assertThat c.sumOfFactors(),
      is(expected.inject(0) { sum, n -> sum += n})
}
```

```
@Test public void test_mock_with_factors_finder() {
  def expected = [1, 2, 6, 7, 23, 46]
  def facts = [factors: {expected as Set}] as Finder
  Classifier7 c = new Classifier7(46, facts)
  assertThat c.sumOfFactors(),
      is(expected.inject(0) { sum, n -> sum += n})
}
```


```java
@Test public void external_factors() {
    final Finder facts = context.mock(Finder.class);
    Classifier7 c = new Classifier7(46, facts);
    final Set<Integer> expected =
            new HashSet(Arrays.asList(1, 2, 6, 7, 23, 46));
    context.checking(new Expectations() {{
        one(facts).factors(); will(returnValue(expected));
    }});
    assertThat(c.sumOfFactors(), is(1 + 2 + 6 + 7+ 23 + 46));
    context.assertIsSatisfied();
}
```

ceremony

essence

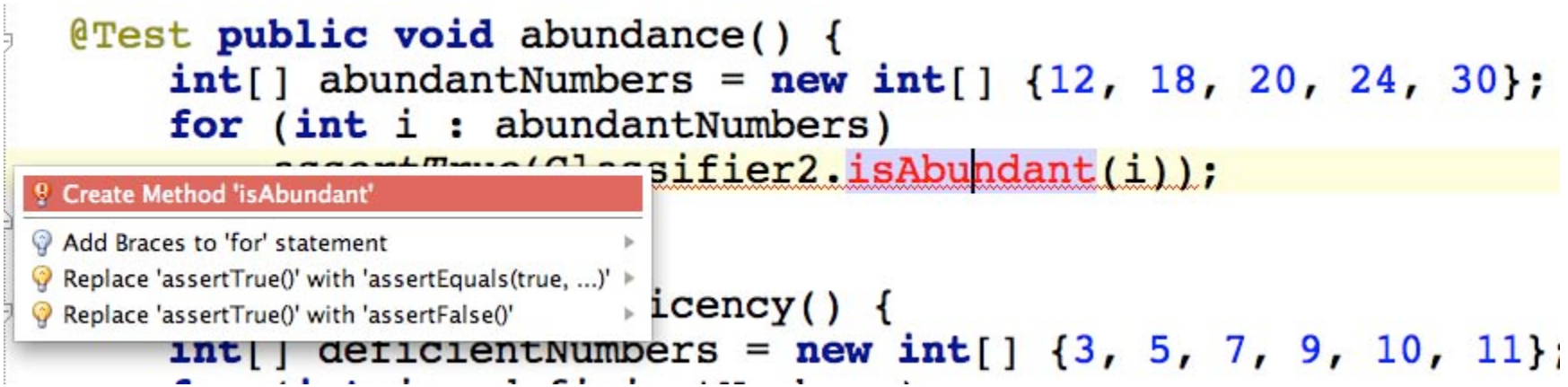

```groovy
@Test public void test_mock_with_factors_finder() {
  def expected = [1, 2, 6, 7, 23, 46]
  def facts = [factors: {expected as Set}] as Finder
  Classifier7 c = new Classifier7(46, facts)
  assertThat c.sumOfFactors(),
      is(expected.inject(0) { sum, n -> sum += n})
}
```

it is professionally irresponsible to not write tests

it is professionally irresponsible to use the most cumbersome tools

how ide's help

```
@Test public void abundance() {
    int[] abundantNumbers = new int[] {12, 18, 20, 24, 30};
    for (int i : abundantNumbers)
        assertTrue(Classifier2.isAbundant(i));
}
```



```
public static boolean isAbundant(int i) {
}
```

```
public static boolean isAbundant(int i) {
    return false;
}
```

```
public static boolean isAbundant(int i) {
    return sumOfFactorsFor(i) - i > i;
}
```

```
public static boolean isAbundant(int number) {
    return sumOfFactorsFor(number) - number > number;
}
```

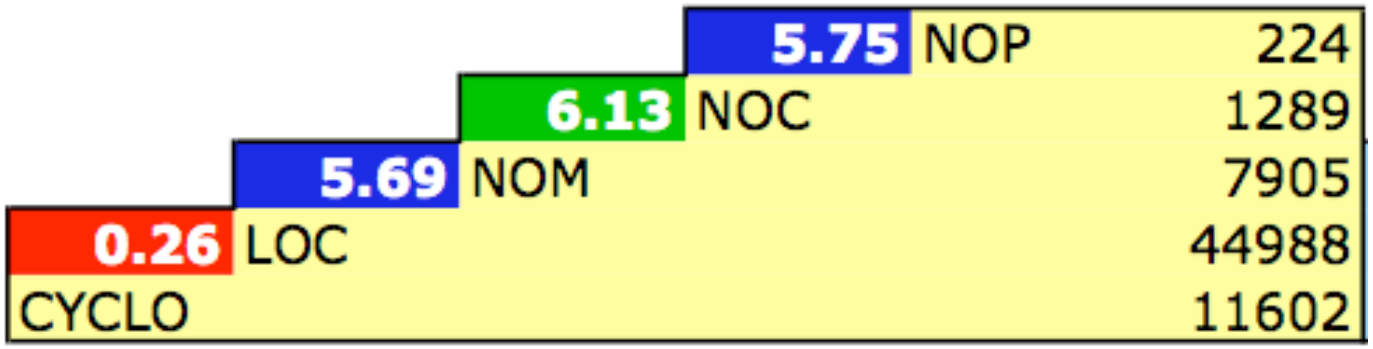
case study

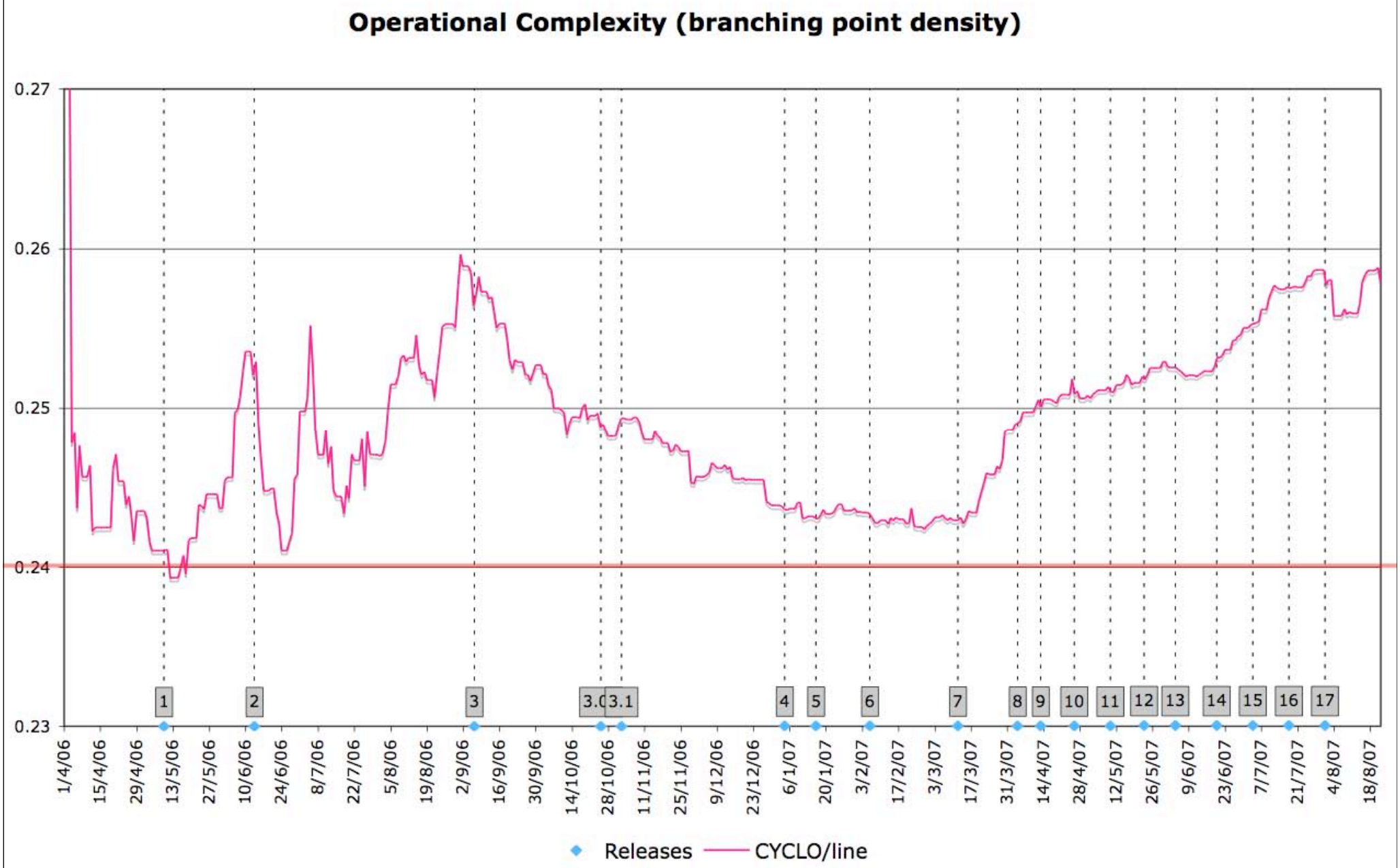
rigorously applied over the life of a real project

tech lead diligent about metrics

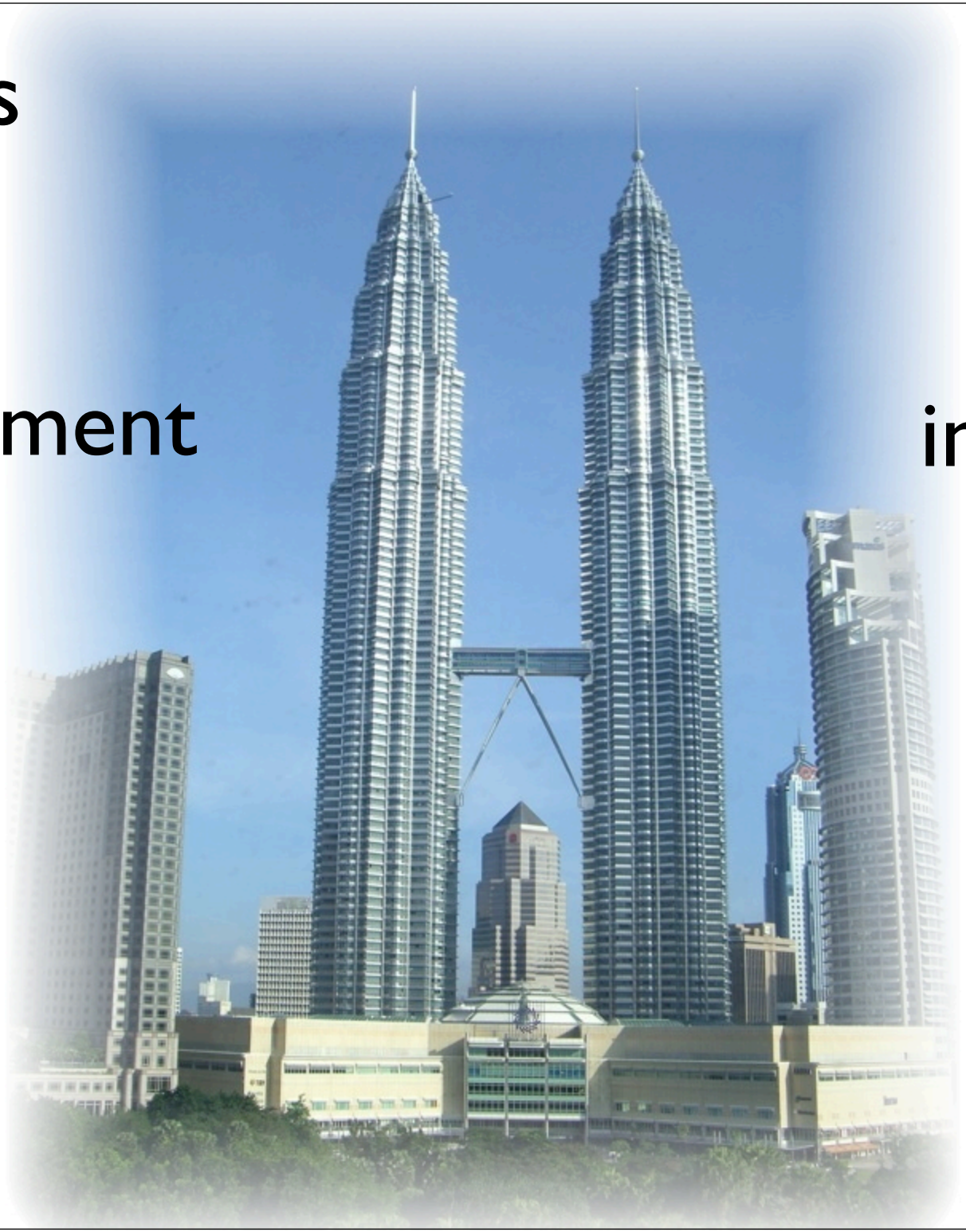
every line of code t d d

results


5.75 NOP 224
6.13 NOC 1289
5.69 NOM 7905
0.26 LOC 44988
CYCLO 11602


Operational Complexity (branching point density)
0.27
0.26
0.25
0.24
0.23
1
2
3
3.0
3.1
4
5
6
7
8
9
10
11
12
13
14
15
16
17
1/4/06
15/4/06
29/4/06
13/5/06
27/5/06
10/6/06
24/6/06
8/7/06
22/7/06
5/8/06
19/8/06
2/9/06
16/9/06
30/9/06
14/10/06
28/10/06
11/11/06
25/11/06
9/12/06
23/12/06
6/1/07
20/1/07
3/2/07
17/2/07
3/3/07
17/3/07
31/3/07
14/4/07
28/4/07
12/5/07
26/5/07
9/6/07
23/6/07
7/7/07
21/7/07
4/8/07
18/8/07
Releases
CYCLO/line

rigorous
test-
driven
development



improves
the
design
of
your
code

?'s

please fill out the session evaluations

samples at `github.com/nealford`

NEAL FORD software architect / meme wrangler

ThoughtWorks

nford@thoughtworks.com
3003 Summit Boulevard, Atlanta, GA 30319
www.nealford.com
www.thoughtworks.com
memeagora.blogspot.com