Evolutionary architecture and emergent design: Test-driven design, Part 2

More on allowing tests to drive and improve your design

Skill Level: Intermediate

Neal Ford (nford@thoughtworks.com)
Software Architect / Meme Wrangler
ThoughtWorks

07 Apr 2009

Testing is only a side effect of test-driven development (TDD); when done right, TDD improves the overall design of your code. This *Evolutionary architecture and emergent design* installment completes a walk-through of an extended example showing how design can emerge from the concerns that float up from testing.

This is the second part of a two-part article investigating how use of TDD allows better design to emerge from the process of writing tests before you write code. In Part 1, I wrote a version of a perfect-number finder using test-after development (writing the test after writing the code). Then I wrote a version using TDD (writing the tests before the code, allowing the testing to drive the code's design). At the end of Part 1, I found that I had made a fundamental flaw in thinking about the kind of data structure used to keep the list of perfect numbers: my instinct started me with an `ArrayList`, but I found that the abstraction was more suited to a `Set`. I'll pick up at that point, expanding the discussion into ways you can improve the quality of your tests and check the quality of the finished code.

Test quality

The test using the better abstraction of a `Set` appears in Listing 1:

Listing 1. Unit test with better Set abstraction

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

This code tests one of the most critical parts of my problem domain: getting factors for a number. I want to test that behavior thoroughly because it represents the most complex part of the problem, making it the most error-prone. However, it contains a clumsy construct: `new HashSet(Arrays.asList(1, 2, 3, 6));`. Even with modern IDE support, this makes for awkward coding: type `new`, type `Has`, and let code insight take over; type `<Int` and let code insight take over, *ad nauseam*. I'm going to make this easier.

About this series
This series aims to provide a fresh perspective on the often-discussed but elusive concepts of software architecture and design. Through concrete examples, Neal Ford gives you a solid grounding in the agile practices of *evolutionary architecture* and *emergent design*. By deferring important architectural and design decisions until the last responsible moment, you can prevent unnecessary complexity from undermining your software projects.

Moist tests

One of the mantras of writing good code comes from *The Pragmatic Programmer* by Andy Hunt and Dave Thomas (see Resources) — the DRY (Don't Repeat Yourself) principle. It admonishes you to keep all repetition out of your code because it frequently leads to problems. However, DRY doesn't apply to unit tests. Unit tests often need to test nuanced behavior of the code under test, leading to similar and duplicated situations. The copy-and-paste code to create the expected result from Listing 1 (`new HashSet(Arrays.asList(1, 2, 3, 6))`) is a great example of that because you're going to want lots of variants of it in different tests.

My TDD rule of thumb is that tests should be *moist* but not *drenched*. By that, I mean that some duplication in tests is acceptable (and inevitable), but you shouldn't go out of your way to create clumsy repeated constructs. To that end, I'm going to refactor my test to provide a `private` helper method to handle this common creation idiom for me; it appears in Listing 2:

Listing 2. The helper method to keep my test moist

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

The code in Listing 2 makes all my tests for factors much cleaner, as shown in the rewritten test from Listing 1, shown in Listing 3:

Listing 3. Moister test to check factors for a number

```
@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));
}
```

Just because you're writing tests doesn't mean that you should throw good design principles out the window. Tests are different kinds of code, but good (albeit different) principles apply to them as well.

Boundary conditions

TDD encourages developers to write a failing test when writing the first test for some new functionality. This prevents the test from accidentally passing in all circumstances, making for a test that doesn't actually test anything (a *tautology* test). Tests can also verify behavior that you think you have right but haven't tested enough to have confidence. These tests don't necessarily need to fail first (although a failure when you think a test should pass is pure gold, because you've found a potential bug). Thinking about testing leads you to consider what is testable.

Some of the oft-neglected test cases are *boundary conditions*: what will your code do when faced with unusual input? Writing lots of tests around the `getFactors()` method opens your thinking about what reasonable and unreasonable inputs might occur.

To that end, I'm going to add a few tests for interesting boundary conditions, shown in Listing 4:

Listing 4. Boundary conditions for factoring

```
@Test public void factors_for_100() {
    Classifier5 c = new Classifier5(100);
    c.calculateFactors();
    assertThat(c.getFactors(),
            is(expectationSetWith(1, 100, 2, 50, 4, 25, 5, 20, 10)));
}

@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)));
}
```

The number 100 seemed interesting because it has lots of factors. By way of testing for several different numbers, I realized that it makes no sense in the problem domain to have negative numbers, so I wrote a test (which did indeed fail before I fixed it) to exclude negative numbers. Thinking about negative numbers made me also think about `MAX_INT`: should my solution consider what happens if the user of the system needs `long` numbers? My original assumption limited the numbers to integers, but I need to make sure that this is a valid assumption.

Requirements gathering is lossy compression

Look around you and find a picture or piece of art. Let's say arbitrarily that the picture contains 2 million pixels. What would happen if you compressed that image to only 2,000 pixels? Would it still look the same? (Perhaps if it's a Rothko painting, but that's a rare case.) Compressing by removing information is a *lossy* compression algorithm. If you take the compressed version and try to restore it to 2 million pixels, you'll need to make some stuff up. You might be able to guess correctly sometimes, but not in every case.

Traditional "big design up front" requirements sessions are lossy compression for what an application needs to do. Business analysts can't anticipate every question that will arise, so developers are left to create information to fill in the details. Developers are notoriously bad at this, leading to much of the grief between the people who define the requirements and those who implement them.

Agile processes attempt to mitigate this lossiness by delaying the decompression algorithm as late as possible and always having someone around who can answer questions about what it should really do. Design without details is impossible, so no matter what your methodology, you must come up with a workable way to fill in the details inevitably removed by the gathering-and-definition process.

Testing boundary conditions forces you to question your assumptions. It's easy to make invalid assumptions when coding a solution. In fact, that is one of the weaknesses of traditional requirements gathering — it can never gather enough detail to eliminate implementation questions that inevitably arise. Requirements gathering is a lossy form of compression.

Because so much is omitted by the process of defining what a piece of software must do, you need mechanisms in place to help you recreate the questions you must ask to understand it fully. Making guesses about what the business people really want is hazardous because you'll mostly get them wrong. Using tests to investigate boundary conditions helps you find the questions to ask, which is most of the battle for comprehension. Finding the right questions to ask matters a lot in achieving a good design.

Positive and negative tests

When starting this exploration of the problem, I decomposed it into several subtasks. As I wrote tests, I discovered another important decomposed task. Here is the entire list:

1. I need factors for the number in question.
2. I need to determine if a number is a factor.
3. **I need to determine how to add factors to the list of factors.**
4. I need to sum the factors.
5. I need to determine if a number is perfect.

The two remaining tasks are summing the factors and testing for perfection. No surprises occur for these two tasks; the last couple of tests appear in Listing 5:

Listing 5. The last two tests for perfect numbers

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

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

After checking with Wikipedia to find the first few perfect numbers, I can write a test that verifies that I can in fact find perfect numbers. But I'm not finished. Having a positive test is only half of the job. I also need a test to verify that I don't accidentally categorize a nonperfect number. To that end, I write a negative test, which appears in Listing 6:

Listing 6. Negative test to make sure perfect-number categorization works correctly

```
@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());
    }
}
```

This code reports that my perfect-number algorithm works correctly, but it's very slow. I can guess why by looking at my `calculateFactors()` method, shown in Listing 7:

Listing 7. The naive getFactors() method.

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

The problem manifested in Listing 7 is the same in the test-after version of the code from Part 1: the factor-harvesting code goes all the way up to the number itself. I can improve this code by harvesting the factors in pairs, allowing me to analyze only up to the square root of the number, as shown in the refactored version in Listing 8:

Listing 8. The better-performing, refactored version of the calculateFactors() method

```
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);
}
```

This is a similar refactoring to the one I did in the test-after version of the code (in Part 1), but the change happens in two different methods this time. The change is simpler here because I already abstracted the `addFactors()` functionality into its own method, and this version uses the `Set` abstraction, eliminating the clumsy test to make sure I don't get duplicates that showed up in the test-after version.

The guiding principle of optimization should always be *get it right, then make it fast*. A comprehensive set of unit tests makes it easy to verify the behavior, letting you freely play "what if" games with optimization without worrying that you've broken something.

I'm done with the test-driven version of the perfect-number finder; the entire class is shown in Listing 9:

Listing 9. The complete TDD version of the number categorizer

```
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;
    }
}
```

Composable methods

One of the benefits surrounding test-driven-developed code mentioned in Part 1 is *composability*, based on the composed method pattern by Kent Beck (see Resources). Composed method encourages building software with lots of cohesive methods. TDD facilitates this because you must have small chunks of functionality for testability. Composed method helps design because it generates reusable building blocks.

You can see this in the number and names of methods in the solution driven by TDD. Here are the methods in the final version of the TDD perfect-number categorizer:

- `isFactor()`
- `getFactors()`
- `calculateFactors()`
- `addFactor()`
- `sumOfFactors()`
- `isPerfect()`

Here's an example of the benefits of composed method. Let's say that you've written your perfect-number finder TDD, and some other group within your company has written a test-after version of a perfect-number finder (an example appears in Part 1). Now, your users come running into the room in a blind panic: "We must determine abundance and deficiency too!" In an *abundant* number, the sum of the factors is greater than the number, and in a *deficient* number, the sum of the factors is less than the number.

For the test-after version, where all the logic resides in one long method, they must rewrite the entire solution, refactoring out the code that abundance, deficiency, and perfection have in common. In the TDD version, I only need to write two new methods, shown in Listing 10:

Listing 10. Support for abundant and deficient numbers

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

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

The only task left in these two methods is to refactor the `calculateFactors()` method into the constructor of the class. (It was innocuous in the `isPerfect()` method, but now it's duplicated across all three methods and therefore should refactored.)

Writing code as small building blocks makes the code more reusable, so this should be one of your main design criteria. Using tests to help evolve your design encourages writing composable methods, improving your design.

Measuring code quality

Early on in Part 1, I made the claim that the TDD version of the code would be

objectively better than the test-after version. I've shown a fair amount of anecdotal evidence, but what about the proof? Of course, no purely objective measures of code quality exist, but several metrics can show certain dimensions of it; one of those is *cyclomatic complexity* (see Resources), created by Thomas McCabe to measure the complexity of code. The formula is quite simple: the number of edges minus the number of nodes plus 2, where the edges represent the execution path and nodes represent lines of code. For example, consider the code in Listing 11:

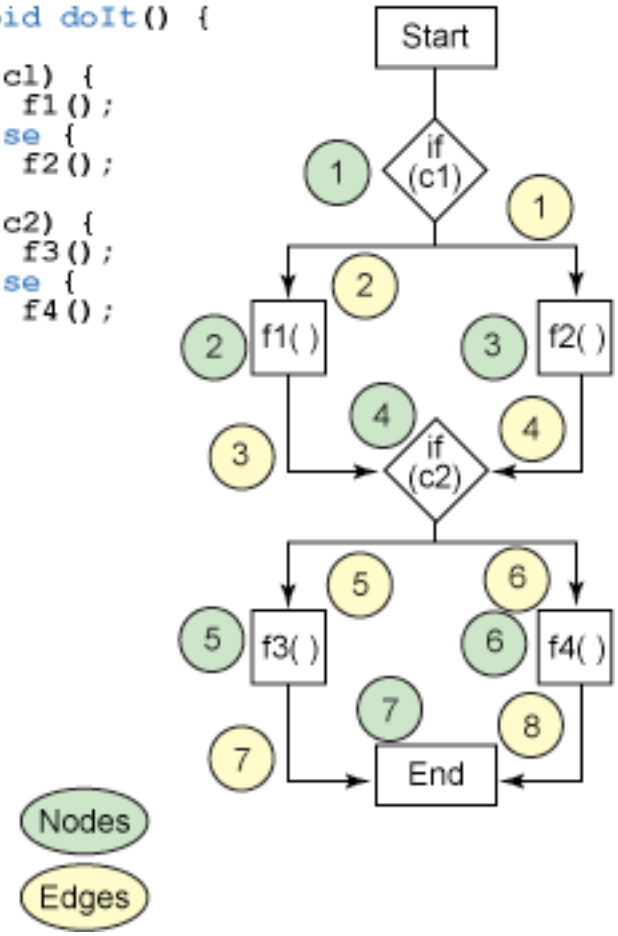
Listing 11. Simple Java method for determining cyclomatic complexity

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

If you diagram the method shown in Listing 11 into a flow chart, you can easily count the number of edges and nodes and calculate the cyclomatic complexity, as shown in Figure 1. This method has a cyclomatic complexity of 3 (8 - 7 + 2).

Figure 1. The nodes and edges of the doit() method

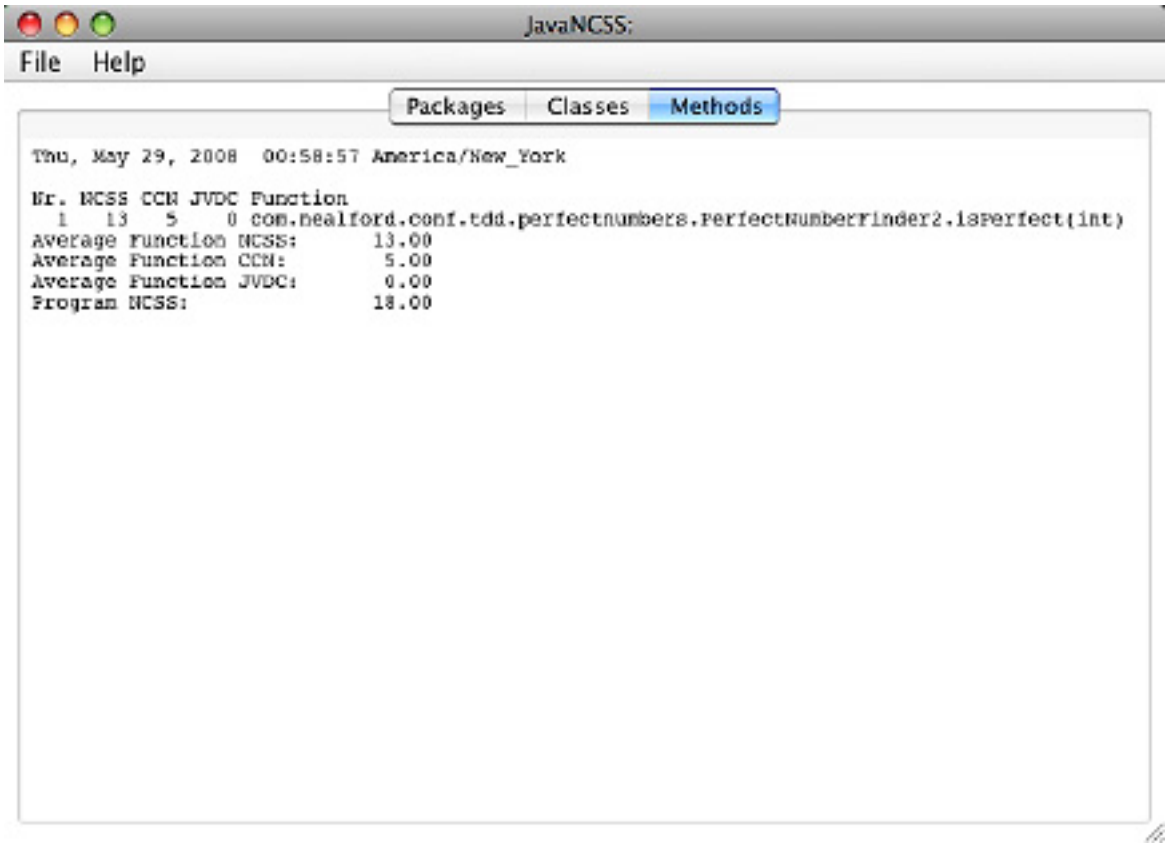
```
public void doIt() {

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



To measure the two versions of perfect-number code, I'll use an open source tool for Java cyclomatic complexity called JavaNCSS (the "NCSS" stands for "non-commenting source statements," which this tool also measures). See Resources for download information.

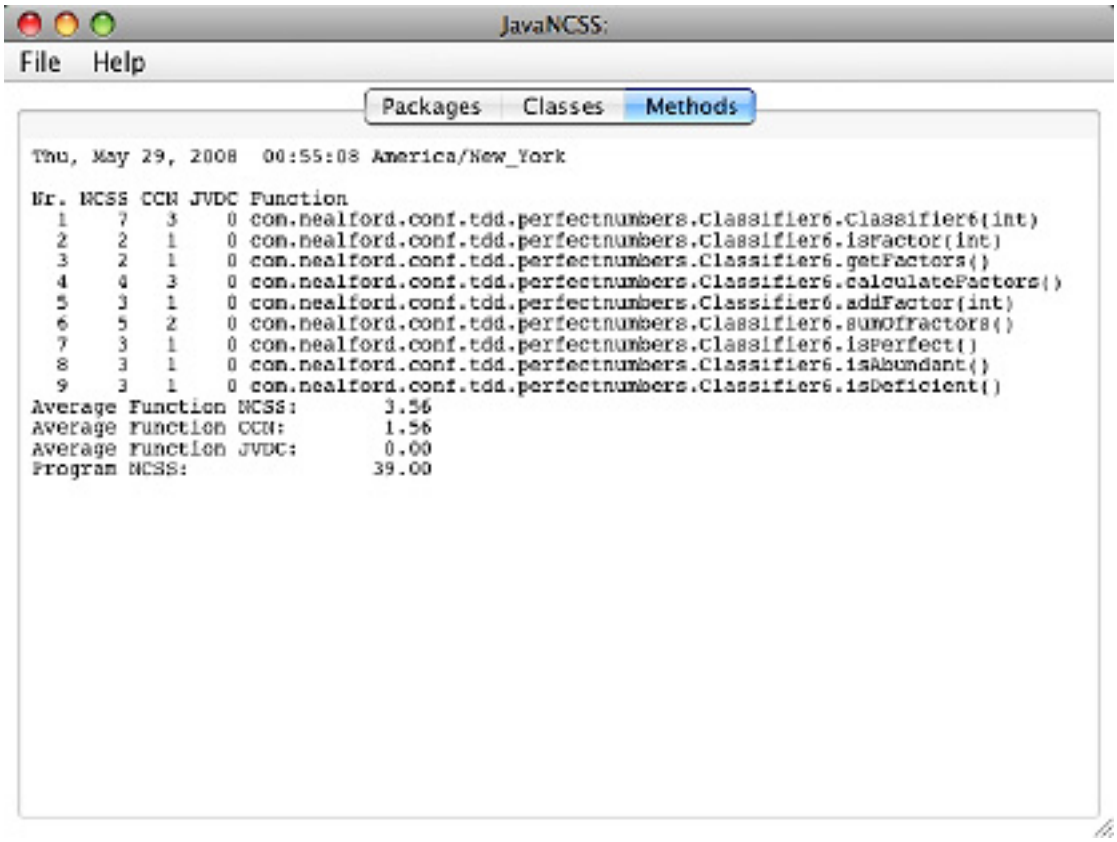
Running JavaNCSS on the test-after code yields the results shown in Figure 2:

Figure 2. Cyclomatic complexity for the test-after perfect-number finder



Only one method exists in this version, and JavaNCSS reports that the class's methods average 13 lines of code, with a cyclomatic complexity of 5.00. Compare this to the TDD version, shown in Figure 3:

Figure 3. Cyclomatic complexity for the TDD version of the perfect-number finder



The TDD version of the code clearly includes many more methods, averaging 3.56 lines of code per method, with an average cyclomatic complexity of only 1.56. Gauging by this measure, the TDD version is more than three times simpler than the test-after code. Even for this small problem, that's a significant difference.

Summary

Over the last two installments of the *Evolutionary architecture and emergent design* series, I've gone into great depth about the benefits of testing *before* you write your code. You end up with simpler methods, with better abstractions, that are more reusable as building blocks. And you get tests for free!

Testing can lead you down the path of better design if you get out of its way. One of the more insidious detriments to good design are the designers and their preconceived notions. Disconnecting the part of your brain that accidentally makes bad decisions is hard. TDD provides a habitual way to let solutions bubble up from the problem rather than rain down in the form of mistaken notions.

In the next installment, I leave testing for a while and talk about two important patterns borrowed from the Smalltalk world: composed method and the *single level of abstraction* principle.

Resources

Learn

- "Test-driven design, Part 1" (Neal Ford, developerWorks, February 2009): Read the first part of this article.
- *The Pragmatic Programmer* (Andy Hunt and Dave Thomas, Pragmatic Programmers, 2001): The DRY principle made famous by this book doesn't necessarily apply to TDD.
- Perfect number: Wikipedia explains the mathematical concept of perfect numbers.
- *Test-Driven Development By Example* (Kent Beck, Addison-Wesley, 2003): Beck, the creator of Extreme Programming, uses examples based on money to explain TDD.
- *Smalltalk Best Practice Patterns* (Kent Beck, Prentice Hall, 1996): Learn more about the *composed method* pattern.
- "*In pursuit of code quality*: Monitoring cyclomatic complexity" (Andrew Glover, developerWorks, March 2006): Read about using simple code metrics and Java-based tools to monitor cyclomatic complexity.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- JavaNCSS: Download the JavaNCSS source-measurement suite.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Neal Ford
Neal Ford is Software Architect and Meme Wrangler at **Thought**Works, a global IT consultancy. He is also the designer and developer of applications, instructional materials, magazine articles, courseware, and video/DVD presentations, and he is the author or editor of books spanning a variety of technologies, including the most recent *The Productive Programmer*. He focuses on designing and building large-scale enterprise applications. He is also an internationally acclaimed speaker at

developer conferences worldwide. Check out his Web site.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.