

Evolutionary architecture and emergent design: Test-driven design, Part 1

Allowing tests to drive and improve your design

Skill Level: Intermediate

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

24 Feb 2009

Most developers think that the most beneficial part of using test-driven development (TDD) are the tests. But, when done right, TDD improves the overall design of your code. This installment in the [Evolutionary architecture and emergent design](#) series walks through an extended example showing how design can emerge from the concerns that float up from testing. Testing is only a side effect of TDD; the important part is how it changes your code for the better.

One of the common agile development practices is TDD. TDD is a style of writing software that uses tests to help you understand the last step of the requirements phase. You write tests before you write the code, solidifying your understanding of what the code needs to do.

Most developers think that the primary benefit derived from TDD is the comprehensive set of unit tests you end up with. However, when done correctly, TDD can change the overall design of your code for the better because it defers decisions until the last responsible moment. Because you don't make design decisions up front, it keeps you open to better design options or refactoring to better designs. This article walks through an example that illustrates the power of allowing design to emerge from the decisions around unit tests.

About this series

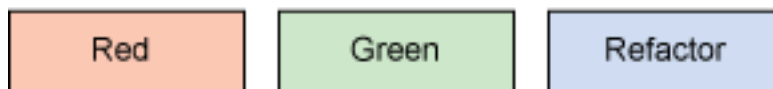
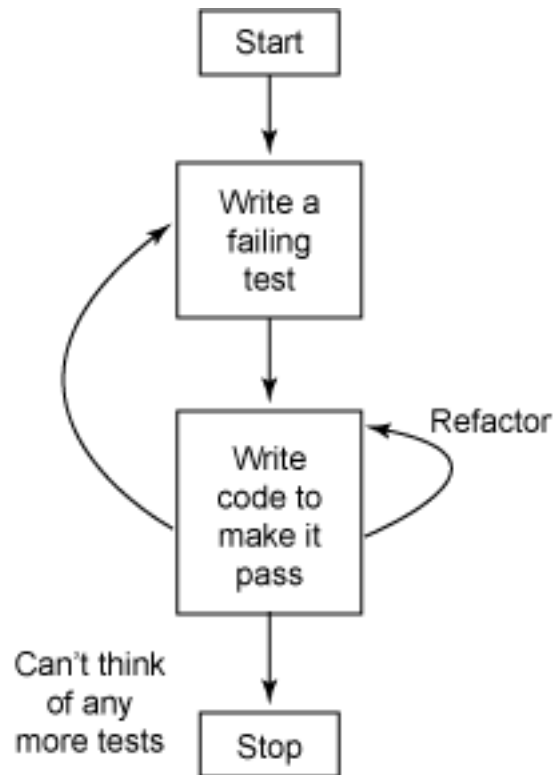
This [series](#) aims to provide a fresh perspective on the often-discussed but elusive concepts of software architecture and design. Through concrete examples, Neal Ford gives you a solid

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.

TDD workflow

The important word in the term *test-driven development* is *driven*, indicating that testing drives the development process. Figure 1 shows the TDD workflow:

Figure 1. The TDD workflow



The workflow in Figure 1 is:

1. Write a failing test.
2. Write code to make it pass.
3. Repeat steps 1 and 2.

4. Along the way, refactor aggressively.
5. When you can't think of any more tests, you must be done.

TDD vs. test after

Test-driven development insists that the tests appear first. Only after you have your tests written (and failing) do you write the code under test. Many developers use a variant of testing called test-after development (TAD), whereby you write the code and then write the unit tests. In this case, you still get tests, but you don't get the emergent design aspects of TDD. Nothing prevents you from writing some perfectly hideous code and then scratching your head over how you are going to test it. By writing the code first, you embed your preconceptions of how the code will work, then test it. TDD requires that you do the opposite: write the test first and allow it to inform how you write the code that makes the test pass. To illustrate this important distinction, I'll embark on an extended example.

Perfect numbers

To show the design benefits of TDD, I need a problem to solve. In his book *Test Driven Development* (see [Resources](#)), Kent Beck uses currency as an example — a pretty good illustration of TDD but a little simplistic. The real challenge is finding an example that isn't so complex that you get lost in the problem domain but complex enough to show real value.

To that end, I've chosen *perfect numbers*. For those of you not up on your math trivia, the concept goes back to before Euclid (who did one of the early proofs deriving perfect numbers). A perfect number is a number whose factors add up to the number. For example, 6 is a perfect number because the factors of 6 (excluding 6 itself) are 1, 2, and 3, and $1 + 2 + 3 = 6$. A more algorithmic definition for a perfect number is *a number where the sum of the factors (excluding the number itself) equals the number*. In my example, the calculation is $1 + 2 + 3 + 6 - 6 = 6$.

That's the problem domain to tackle: create a perfect-number finder. I'm going to implement this solution in two different ways. First, I'll turn off the part of my brain that wants to do TDD and just write the solution, then write the tests for it. Then, I'll evolve a TDD version of the solution so that I can compare and contrast the two approaches.

For this example, I implement a perfect-number finder in the Java language (version 5 or later because I'm using annotations in my tests), JUnit 4.x (the latest version), and the Hamcrest matchers from Google code (see [Resources](#)). The Hamcrest matchers provide a humane interface syntactic sugar on top of the standard JUnit matchers. For example, instead of `assertEquals(expected, actual)`, you can

write `assertEquals(actual, is(expected))`, which reads more like a real sentence. Hamcrest matchers come with JUnit 4.x (they are just a static import away); if you are still using JUnit 3.x, you can download a compatible version.

Test after

Listing 1 shows the first version of the `PerfectNumberFinder`:

Listing 1. The test-after `PerfectNumberFinder`

```
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;
    }
}
```

This isn't particularly spectacular code, but it gets the job done. I start by creating a list of all the factors as a dynamic list (an `ArrayList`). I add 1 and the target number to the list. (I'm adhering to the formula I gave above, and all lists of factors include 1 and the number itself.) Then, I iterate over the possible factors up to the number itself, checking each in turn to see if it is a factor. If so, I add it to the list. Next, I sum all the factors and finally write the Java version of the formula shown above to determine perfection.

Now, I need a test-after unit test to determine if it works or not. I need at least two tests: one to see if the perfect numbers report correctly and another to check that I don't get false positives. The unit tests are in Listing 2:

Listing 2. Unit tests for `PerfectNumberFinder`

```
public class PerfectNumberFinderTest {
    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));
    }
}

@Test public void test_perfection_for_2nd_version() {
    for (int i : PERFECT_NUMS)
        assertTrue(PerfectNumberFinder2.isPerfect(i));
}

@Test public void test_non_perfection_for_2nd_version() {
    List<Integer> expected = new ArrayList<Integer>(Arrays.asList(PERFECT_NUMS));
    for (int i = 2; i < 100000; i++) {
        if (expected.contains(i))
            assertTrue(PerfectNumberFinder2.isPerfect(i));
        else
            assertFalse(PerfectNumberFinder2.isPerfect(i));
    }
    assertTrue(PerfectNumberFinder2.isPerfect(PERFECT_NUMS[4]));
}
}

```

What's with the "_" in the test names?

Placing underscores in the method names when writing unit tests is one of my coding quirks. Of course, the Java standard states that camel case is the proper way to write method names. But I maintain that test method names are *different* from regular method names. Test method names should indicate what the method is testing, and therefore they become long, descriptive names, which is exactly what you want when one breaks. However, reading long camel case names is hard, especially in a unit test runner where dozens or hundreds of tests appear, because lots of test names start out with similar values and only diverge near the end. On all the projects I work on, I strongly advocate underscores (only in test names) to improve their readability.

This code correctly reports perfect numbers, but it runs very slowly for the negative test because I'm checking so many numbers. Performance issues can emerge from unit tests, which leads me back to the code to see if I can make some improvements. Currently, I'm looping all the way up to the number itself to harvest factors. But do I need to go that far? Not if I can harvest the factors in pairs. All factors are in pairs (for example, if the target number is 28, when I find the 2 factor, I can also grab 14). I only need to go up to the square root of the number if I can harvest the factors in pairs. To that end, I improve the algorithm and refactor the code to Listing 3:

Listing 3. Improved version of the algorithm

```

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;
}
}
```

This code runs in a respectable time, but a couple of the test assertions fail. It turns out that when you harvest the numbers in pairs, you accidentally grab numbers twice when you reach a whole-number square root. For example, for the number 16, the square root is 4, which inadvertently gets added to the list twice. This is easy to fix by creating a guard condition for this case, as shown in Listing 4:

Listing 4. Fixed improved algorithm

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

Now I have a test-after version of the perfect-number finder. It works, but some design problems rear their ugly head as well. First, I used comments to delineate sections of the code. This is always a code smell: it's a cry for help for refactoring into their own methods. The new stuff I just added probably needs a comment explaining what the little guard condition does, but I'll leave that alone for now. The biggest problem lies with its length. My rule of thumb on Java projects says that no methods should ever be longer than 10 lines of code. If a method exceeds that number, it is almost certainly doing more than one thing, which it shouldn't do. This method clearly violates this heuristic, so I'm going to take another stab at it, this time using TDD.

Emergent design through TDD

The mantra for coding TDD is: "What is the simplest thing for which I can write a test?" In this case, is it "is a number perfect or not?"? No — that answer is too broad. I must decompose the problem and think about what "perfect number" means. I can easily come up with several steps required to discover a perfect number:

- I need factors for the number in question.
- I need to determine if a number is a factor.
- I need to sum the factors.

Toward the idea of the simplest thing, which of the items in this list seems the simplest? I think it's the determination if a number is a factor of another number, so that's my first test, which appears in Listing 5:

Listing 5. Test for "Is a number a factor?"

```
public class Classifier1Test {  
    @Test public void is_1_a_factor_of_10() {  
        assertTrue(Classifier1.isFactor(1, 10));  
    }  
}
```

This simple test is trivial to the point of stupidity, which is what I want. To get this test to compile, you must have a class named `Classifier1`, with an `isFactor()` method. So I must create a skeleton structure of my class before I can even get a red bar. Writing insanely trivial unit tests allows you to get the structure in place before you need to start thinking about the problem domain in any significant way. I want to think about only one thing at a time, and this allows me to work on the skeletal structure without worrying about nuances of the problem I'm solving. Once I get this to compile and present a red bar, I'm ready to write the code, shown in Listing 6:

Listing 6. First pass at a factor method

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

Okay, that's nice and simple, and it does the job. Now I can move on to the next-simplest task: getting a list of factors of numbers. The test appears in Listing 7:

Listing 7. Next test: Factors for a number

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

Listing 7 has the simplest test I could muster for getting factors, so now I can write the simplest code that makes this test pass (and refactor it later to make it more

sophisticated). The next method appears in Listing 8:

Listing 8. Simple factorsFor() method

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

Although this method works, it stops me dead in my tracks. It seemed like a good idea to make the `isFactor()` method static because it merely returns something based on its input. However, now I've also made the `factorsFor()` method static, meaning that I have to pass a parameter called `number` to both methods. This code is becoming very procedural, which is a side effect of too much staticness. To fix it, I'm going to refactor the two methods I already have, which is easy because I have so little code thus far. The refactored `Classifier` class appears in Listing 9:

Listing 9. Improved Classifier class

```
public class Classifier2 {  
    private int _number;  
  
    public Classifier2(int number) {  
        _number = number;  
    }  
  
    public boolean isFactor(int factor) {  
        return _number % factor == 0;  
    }  
}
```

I've made the number a member variable within the `Classifier2` class, which allows me to avoid passing it around as a parameter to a bunch of static methods.

The next thing on my decomposition list says that I need to find the factors for a number. Thus, my next test should check that (shown in Listing 10):

Listing 10. Next test: Factors for a number

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

Now, I'll take a shot at implementing the method that returns an array of factors for a given parameter, shown in Listing 11:

Listing 11. First pass at a getFactors() method


```
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;
}
```

This code allows the test to pass, but upon reflection, it's awful! This sometimes happens when you investigate ways to implement code using tests. What's so terrible about this code? First, it's very long and complex, and it suffers from the "more than one thing" problem. My instinct led me to return an `int[]`, but it adds a lot of complexity to the code at the bottom and doesn't buy me anything. It's a slippery slope to start thinking too much about making things more convenient for future methods that might call this one. You need a compelling reason to add something this complex at this juncture, and I don't have that justification yet. Looking at this code suggests that perhaps `factors` should also exist as internal state to the class, allowing me to break up this method's functionality.

One of the beneficial characteristics that tests surface is really cohesive method. Kent Beck wrote about this in an influential book called *Smalltalk Best Practice Patterns* (see [Resources](#)). In that book, Kent defined a pattern called *composed method*. The composed method pattern defines three key statements:

- Divide your programs into methods that perform one identifiable task.
- Keep all 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.

Composed method is one of the beneficial design characteristics that TDD promotes, and I've clearly violated this pattern in [Listing 11](#)'s `getFactors()` method. I can repair it by taking these steps:

1. Promote `factors` to internal state.
2. Move the initialization code for `factors` to the constructor.
3. Get rid of the gold-plated conversion to `int[]` code and deal with it later if it becomes beneficial.
4. Add another test for `addFactors()`.

The fourth step is quite subtle but important. Writing this flawed version of the code revealed that my first pass at decomposition wasn't complete. The `addFactors()` line of code buried in the middle of this long method is testable behavior. It was so trivial that I didn't notice it when first looking at the problem, but now I see it. This is a frequent occurrence. One test can lead you to further decompose the problem into smaller and smaller chunks, each testable.

I'll put the larger problem of `getFactors()` on hold for a moment and tackle my new smallest problem. Thus, my next test is `addFactors()`, shown in Listing 12:

Listing 12. Test for `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)));
}
```

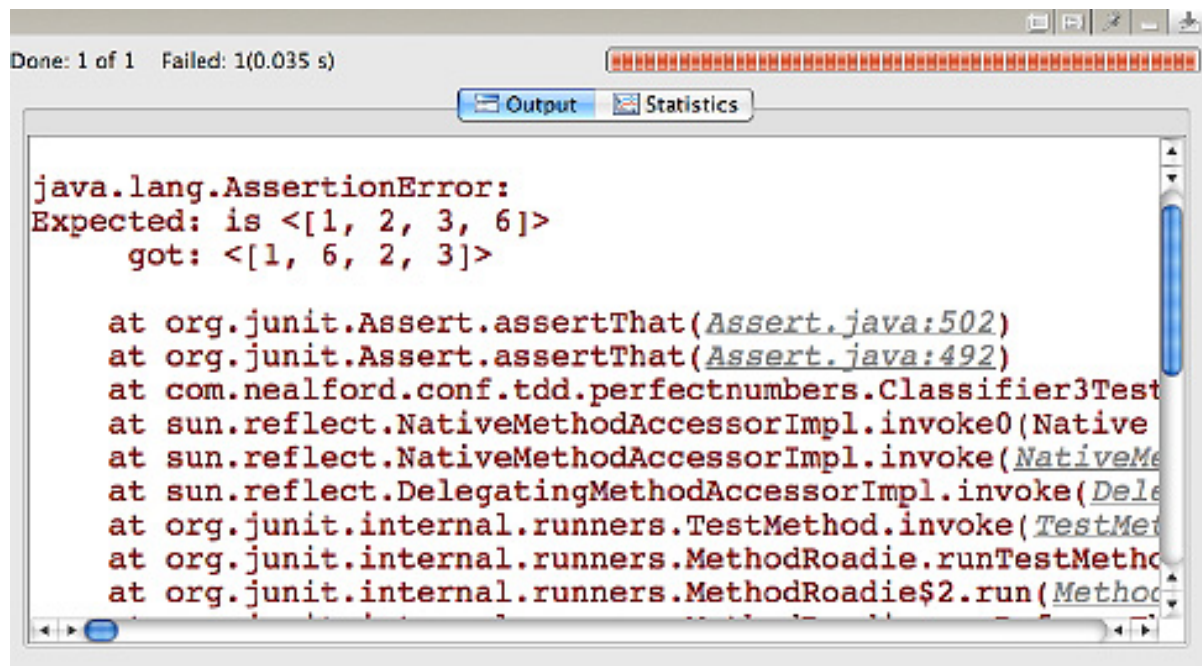
The code under test, shown in Listing 13, is simplicity itself:

Listing 13. The simple code to add factors

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

I run my unit test, full of confidence that I'll see a green bar, and it fails! How can such a simple test fail? The root cause appears in Figure 2:

Figure 2. The failed test root cause



```
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
```

My expected list has the values 1, 2, 3, 6, and the actual return is 1, 6, 2, 3. Ah, that's because I changed my code to add 1 and the number in the constructor. One solution to this problem would be always to write my expectations with the assumption that 1 and the number should always go first. But is that the *correct* solution? No. The problem is much more fundamental. Are factors a *list* of numbers? No, they are a *set* of numbers. My first (incorrect) assumption led me to use a list of integers for my factors, but that's a poor abstraction. By refactoring my code now to use sets instead of lists, I not only fix this problem but make the overall solution better because I'm now using a more accurate abstraction.

This is precisely the kind of flawed thinking that tests can expose if you write the tests before you have any code to cloud your judgement. Now, because of this simple test, the overall design of my code is better because I've discovered a more appropriate abstraction.

Conclusion

Thus far, I've discussed emergent design in the context of the perfect-numbers problem. In particular, notice that the first version of the solution (the test-after version) made the same flawed assumption about data types. "Test after" tests your code's coarse-grained functionality, not the individual parts. TDD tests the building blocks that make up the coarse-grained functionality, exposing more information in the process.

In the next installment, I'll continue with the perfect numbers problem, illustrating more examples of the kinds of design that can emerge if you get out of the way of

your tests. When I have the TDD version complete, I'll compare some metrics between the two code bases. I'll also address some other sticky design questions about TDD, such as if and when to test private methods.

Resources

Learn

- [Hamcrest matchers](#): A library of matcher objects allowing you to define "match" declaratively for use in other frameworks.
- [Test-Driven Development](#) (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.
- [The Productive Programmer](#) (Neal Ford, O'Reilly Media, 2008): A longer version of this article's example appears in the "Test Driven Development" chapter in Neal Ford's most recent book.
- ["Emergent Optimization in Test Driven Design"](#) (Michael Feathers): How testing helps avoid premature optimization.
- 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

- [JUnit](#): Download JUnit.

Discuss

- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

About the author

Neal Ford

Neal Ford is a software architect and Meme Wrangler at **ThoughtWorks**, a global IT consultancy. He also designs and develops 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.