TRANSFORM YOUR SOFTWARE

REFACTORING & GENERALIZATION IN TDD

by William Wake

ISTOCKPHOTO

Two types of code transformations are common in test-driven development (TDD). The first is *refactoring*, which improves the code while preserving its behavior. The second is *generalization*, which broadens the capabilities of the software. Each has its place, and together they help make TDD effective.

Refactoring

Refactoring, as defined in Martin Fowler's landmark book *Refactoring*, is "the process of changing software systems in such a way that it does not alter the external behavior of the code yet improves its internal structure" (see the StickyNotes for a reference). Thus, refactoring is an enabling tool for incremental development—it gives us a mechanism to design concurrently with development. There is a cycle to refactoring:

1. **Identify** a shortcoming in the code (often referred to as a *code smell*).
2. **Select** a refactoring.
3. **Apply** the refactoring.
4. **Repeat**.

Let's look at some code to see how refactoring can improve it.

A STACK-BASED CALCULATOR

Early HP calculators were built around a stack that was exposed to the user. This not only allowed users to enter complex calculations but also let the calculator forgo the complexities of parsing.

To calculate 3 * (7 + 2) you would enter the equivalent of:
3 Push 7 Push 2 + *

Had you wanted 3 * 7 + 2 you'd use:
3 Push 7 * Push 2 +

Each binary operator pops two values from the stack, combines them, and pushes the result on the stack.

We'll create a calculator along these lines. Listing 1 shows the code for a key object, `Calculator`, early in its development. Only "PUSH", "+", "*", and digits are implemented. This code was developed with TDD, so there are tests for each of these aspects (not shown). It's critical to have tests in place—especially when refactoring manually—as they help ensure behavior doesn't change unexpectedly.

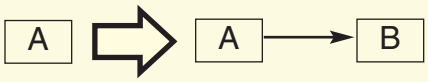
IMPROVING THE CODE

To identify code smells, we can use the catalog in any of the refactoring books, such as *Refactoring*, *Refactoring to Patterns*, or *Refactoring Workbook* (see the StickyNotes for references). Here are a few examples:

- **Duplicated Code:** `add()` and `multiply()` are very similar.
- **Uncommunicative Name:** `Calculator` is an overly broad name.
- **Divergent Change:** Several decisions are folded together in one class—what type object is being operated on, the stack nature, and the application of operators.

```
import java.math.BigInteger;
import java.util.Stack;

public class Calculator {
     final BigInteger emptyValue = new BigInteger("0");

     Stack<BigInteger> values = new Stack<BigInteger>();
     boolean topWasReplaced = false;

     public Calculator() {
          push();
     }

     public void push() {
          push(emptyValue);
     }

     public void push(BigInteger value) {
          values.push(value);
          topWasReplaced = false;
     }

     public BigInteger pop() {
          if (values.isEmpty()) return emptyValue;
          return values.pop();
     }

     public BigInteger top() {
          if (values.isEmpty()) return emptyValue;
          return values.lastElement();
     }

     public void replace(BigInteger value) {
          pop();
          values.push(value);
          topWasReplaced = true;
     }

     public void press(String value) {
          BigInteger top = top();
          StringBuffer buffer = new StringBuffer(
              topWasReplaced ? top.toString() : "");
          buffer.append(value);
          replace(new BigInteger(buffer.toString()));
     }

     public void add() {
          BigInteger value2 = top();
          pop();
          BigInteger value1 = top();
          pop();
          push(value1.add(value2));
     }

     public void multiply() {
          BigInteger value2 = top();
          pop();
          BigInteger value1 = top();
          pop();
          push(value1.multiply(value2));
     }
}
```

Listing 1



EC-1. Create a new class for B.

EC-2. Give A a link to B; give B a link to A if needed.

EC-3. Move Field on any fields that should belong to B.

EC-4. Compile and test after each move.

EC-5. Move Method on any method that should belong to B.

EC-6. Compile and test after each move.

EC-7. In A, expose the reference to B if needed.

Figure 1: Extract Class

- **Complexity:** The `press()` operation keeps track of whether the stack was computed or entered. It looks a little complicated (though this may be "just" domain complexity).
- **Primitive Obsession:** `BigInteger` isn't exactly a primitive, but it's close. The arithmetic operations may need to know the type they're dealing with, but the stack operations do not. Note that `Calculator`'s `press()` method depends on its ability to construct new instances of `BigInteger`.

To demonstrate refactoring, let's get rid of some code duplication. Figures 1 and 2 show the simplified mechanics (based on Fowler) for two refactorings we'll apply: *Extract Class* and *Move Method*. I'll select the Extract Class refactoring to pull each operation into its own class. This gives us two benefits: It isolates the duplication, and it separates the "operator" aspects from the "stack" aspects, reducing the pressure for divergent change.

Now to apply the refactoring:

EC-1, EC-2 Create a new class for `Add`; give `Add` a link to `Calculator`.

```
public class Add {
    private Calculator calc;
    public Add(Calculator calc) {
        this.calc = calc;
    }
}
```

EC-3 *Move Field:* There's nothing to move in this case.

EC-5, MM-1 *Move Method*: Copy `add()` into `Add`:

```
public class Add {
    . . .
    public void add() {
        BigInteger value2 = top();
        pop();
        BigInteger value1 = top();
        pop();
        push(value1.add(value2));
    }
}
```

MM-2 Adjust the code for its new context:

```
public class Add {
    . . .
    public void add() {
        BigInteger value2 = calc.top();
        calc.pop();
        BigInteger value1 = calc.top();
        calc.pop();
        calc.push(value1.add(value2));
    }
}
```

MM-3 Compile `Add`; it's fine.

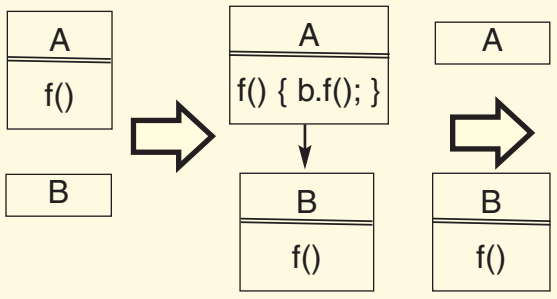
MM-4 Delegate from `Calculator`'s `add()` to `Add`'s `add()`: but note that `Calculator` needs access to an instance of `Add`. I think it will be most convenient if it's a parameter to `Calculator.add()`. A caller of `Calculator.add()` can create a new instance of the `Add` class and pass it in.

```
public void Calculator {
    . . .
    public void add(Add adder) {
        adder.add();
    }
}
```

MM-5 Compile and test; everything still works.

MM-6 Remove calls to the delegating method (on `makeAddButton()`, as shown here, and a number of test cases that are not shown):

```
private static JButton makeAddButton(
    final String buttonName) {
    . . .
```



MM-1. Copy the declaration and code for `f()` into B.

MM-2. Adjust the code to work in the new context.

MM-3. Compile B.

MM-4. Delegate from A's `f()` to B's `f()`.

MM-5. Compile and test everything.

MM-6. Replace each call to A's `f()` with a call to B's `f()`. Test after each replacement.

MM-7. Remove A's `f()` method.

MM-8. Compile and test.

Figure 2: Move Method

```
        new Add(calc).add();
        . . .
    }
```

MM-7 Remove the delegating method. There's no longer a method `add()` in `Calculator`.

MM-8 Compile and test; everything still works.

MM-1 through MM-8 Repeat all steps to create a corresponding class `Multiply`.

The resulting code is shown in listing 2.

THINGS TO NOTICE

There are several things to notice about that brief refactoring session.

The steps are fail-safe. At each point where something could go wrong, the refactoring mechanics make it easy to recover. For example, when moving the method from `Calculator` to `Add`, we copy the method and adjust it to its new context before we get rid of the original. Similarly, rather than call `Add`'s `add()` directly, we take the detour of setting up a delegating method. This lets us confirm that the new method works before we tangle that up with eliminating the delegate.

The steps are surprisingly small. Each step is smaller and easier than it needs to be. By making the steps "too easy," we're less likely to make a mistake. And if we do mess up, a few Control-Zs can take us back.

The steps are boring, almost mechanical. This is part of keeping the refactorings easy. But it's also good news for automation: If you're programming in a language such as Smalltalk, Java, or C#, your environment probably has support for many automated refactorings. But not every useful refactoring will be automated, so it's worth internalizing the approach so you can refactor well without tool support.

MOVING AHEAD

Let's continue, removing the duplication in the new `Add` and `Multiply` classes. Instead of listing the individual steps as I did above, I'll mention the refactorings as they happen. We'll work "at speed," using automated refactoring support where possible.

Note that `add()` and `multiply()` have the same shape. We'll install a common parent class, unify the interfaces, and create a template method.

Move: Common parent class

```
public class BinaryOperator {}

public void Add extends BinaryOperator { … }
public void Multiply extends BinaryOperator { … }
```

```
public class Add {
    private Calculator calc;

    public Add(Calculator calc) { this.calc = calc; }

    public void add() {
        BigInteger value2 = calc.top();
        calc.pop();
        BigInteger value1 = calc.top();
        calc.pop();
        calc.push(value1.add(value2));
    }
}

public class Multiply {
    private Calculator calc;

    public Multiply(Calculator calc) { this.calc = calc; }

    public void multiply() {
        BigInteger value2 = calc.top();
        calc.pop();
        BigInteger value1 = calc.top();
        calc.pop();
        calc.push(value1.multiply(value2));
    }
}
```

Listing 2

```
public void Add {
    . . .
    public void add() {
        BigInteger value2 = calc.top();
        calc.pop();
        BigInteger value1 = calc.top();
        calc.pop();
        calc.push(combine(value1, value2);
    }

    public BigInteger combine(BigInteger value1, value2) {
        return value1.add(value2);
    }
}
```

Listing 3

This has no effect since the parent class is empty.

Move: Pull Up Field

Move field `calc` up to `BinaryOperator`, passed in to the constructor.

Move: Form Template Method

This involves a number of small moves, but it will let us pull the common code into the new parent class.

Move: Extract Method (as part of Form Template Method)

As in listing 3, extract a method `combine()` in each `Add` and

`Multiply`. This highlights the part that varies between the classes: Once you have two values, how do you combine them into a new one?

```
public abstract class BinaryOperator {
    protected Calculator calc;

    public BinaryOperator(Calculator calc) { this.calc = calc; }

    public void execute() {
        BigInteger value2 = calc.top();
        calc.pop();
        BigInteger value1 = calc.top();
        calc.pop();
        calc.push(combine(value1, value2);
    }

    public abstract BigInteger combine(
        BigInteger value1, BigInteger value2);
}

public void Add extends BinaryOperator {
    public Add(Calculator calc) { super(calc); }

    public BigInteger combine(BigInteger value1, BigInteger value2) {
        return value1.add(value2);
    }
}

public void Multiply extends BinaryOperator {
    public Multiply(Calculator calc) { super(calc); }

    public BigInteger combine(BigInteger value1, value2) {
        return value1.multiply(value2);
    }
}
```

Listing 4

Do the same for `Multiply`.

Move: Rename Method (as part of Form Template Method)

Rename `add()` to `execute()` in `Add`.

Rename `multiply()` to `execute()` in `Multiply`.

Move: Pull Up Method (as part of Form Template Method)

Create an abstract `combine()` method on `BinaryOperator`.

Move `execute()` from `Add` to `BinaryOperator`. Everything tests OK.

Move: Remove Extraneous Method (as the final step of Form Template Method)

Remove `execute()` from `Multiply` since this method is identical to the one in the superclass.

Listing 4 shows our result.

WHAT COUNTS AS A REFACTORING?

Refactorings don't "alter the external behavior of the code," and that raises the question "What counts as an alteration?"

The most conservative stance would be to require refactorings to have no visible effect, taking everything into account. For example, moving from using a single `BigInteger` to using a stack changed the `Calculator`'s memory usage pattern significantly. Before refactoring, only one value is remembered; afterward, each value is available in the stack. A program that calls `push()` a few million times might reveal the difference by running out of memory. In some cases, changing the text of the code affects its behavior. In "Reflections on Trusting Trust," Ken Thompson describes a self-reproducing program. Refactoring that would be tricky! Any everyday refactoring would change its text and break it.

In refactoring, the usual, if implicit, answer to "What counts as an alteration?" is "different behavior with respect to an abstract model of how a program works"—a model that ignores memory limitations, performance, timing, threads, and other real-world considerations.

Generalization

Refactorings are "behavior-preserving transformations." But there are also interesting transformations that do not preserve behavior. These resemble refactorings in their mechanics but not in terms of effects.

For generalization in TDD, one type of transformation is particularly important—transformations that preserve the behavior of the tests we have but don't preserve the behavior with respect to all possible tests or inputs. "Equivalent with respect to existing tests" means we can transform our code in anticipation of a number of alternative future tests.

There's some risk in this. When we lift up some possible (not yet existing) tests and say, "We can't break these," and lower others and say, "But these can be broken," we open ourselves up to accidentally breaking some tests we had intended to keep working. Generalization is worth the risk, as it leads to new behaviors that pure refactoring can't reach.

In *Test-Driven Development: By Example*, Kent Beck identifies three approaches to adding code:

1. **Obvious Implementation**—Just implement it.
2. **Fake It 'Til You Make It**—Start with a simple (even trivial) implementation that passes the test at hand, then repeatedly generalize the code until it passes more tests.
3. **Triangulation**—Generalize code to support two or more tests.

These generalize from zero, one, or two cases, respectively.

I visualize the "Fake It" process as a sort of geometric projection. Imagine a test as a 2-D figure. We're trying to create a 3-D object with that figure as its shadow. When we fake it, the object we create is like a cardboard cutout—the shadow is right but the figure is a total cheat. But we can gradually replace parts of the cardboard with the real thing. The unchanging shadow tells us we haven't changed anything incorrectly, at least from that test's angle.

```
public class Calculator {
    BigInteger currentValue = new BigInteger("0");

    public BigInteger top() {
        return currentValue;

    }

    public void press(String value) {
        StringBuffer buffer = new StringBuffer(
            top().toString());
        buffer.append(value);
        currentValue = new BigInteger(buffer.toString());
    }
}
```

Listing 5

To see generalization in action, let's look back at one of the earliest steps in the evolution of the `Calculator` object, earlier than that shown in listing 1. Listing 5 shows the `Calculator` at that point.

Listing 6 shows a new test.

```
@Test
public void testPushedNumberGoesIntoStack() {
    calc.press("1");
    calc.push();
    assertEquals(new BigInteger("0"), calc.top());
}
```

Listing 6

Add a stub method for `push()`:

```
public void push() {
}
```

Make the test pass ("Fake It") by setting `currentValue`:

```
public void push() {
    currentValue = new BigInteger("0");
}
```

This keeps the old tests running. If I had made `top()` return the 0 value, that would have broken other tests, and it would push the system in a direction I didn't want to go. To generalize:

Extract a new version of `push()`:

```
public void push() {
    push(new BigInteger("0"));
}
public void push(BigInteger newValue) {
    currentValue = newValue;
}
```

Turn `currentValue` into a collection—a stack. Kent Beck suggests an even safer approach: "At this point I would generally keep duplicate copies of the data until I could prove that the top of the stack was always the same as `currentValue` for the test cases in question." The result is shown in listing 7.

Though the method names imply the presence of a stack, the tests we have so far don't require one. The generalization represents a conscious move toward the design we want.

WAYS TO GENERALIZE

There are several ways objects get information:

- They're given it (e.g., via a parameter to a method).
- They remember it (e.g., in a collection).
- They generate it (e.g., through calculation).
- They ask somebody else (e.g., call a method on another object).

Generalization uses the "trick" of shifting between these. For example, first fake an answer by returning a constant, and then move toward computing the same value from input parameters. Following are several ways to generalize:

Introduce Variables

If you've returned a constant (a quick "Fake It" answer), this constant was chosen as a function of other variables or values. If you can decompose this constant into its constituent parts, you'll often find that it's computed from values the object is told or remembers from being told before.

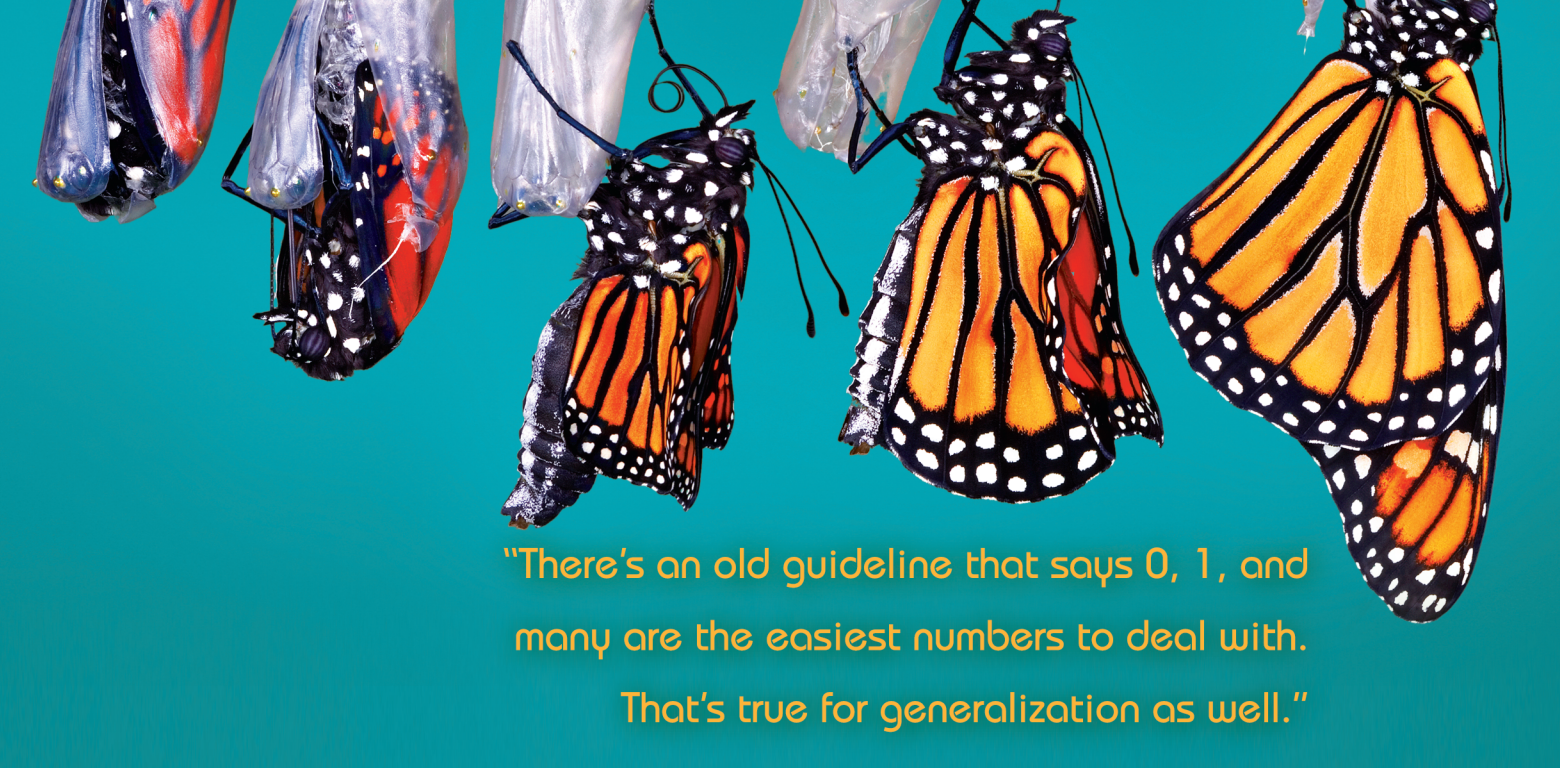
In *Test-Driven Development*, Kent Beck points out that if you're explicit about the way the constant is computed, you'll often see duplication between the expected value and the code that you need—the code can be pushed into the same shape as the answer. For example, 6 becomes 2*3 becomes `width*3` becomes `width*height`.

Substitute an Algorithm

Another way to generalize is to implement a substitute algorithm and then cut over to it. For example, you might replace a simple string matcher with one that can match regular expressions.

Introduce a Hidden/Helper Class

An object may not be sophisticated enough to do what you want, but you may be able to find or create another object that can help. There may be an internal transformation your object can do. For example, consider a buffer-gap implementation that an editor might use; it presents a normal coordinate system to its users (insertion points 0, 1, 2, up to the length of the string), but inside it maintains a second coordinate system (from 0 to the insertion point, followed by a gap, then from some position later to the end of the buffer). A buffer gap object can hide the mapping from the first coordinate system to the second one.




```
Stack<BigInteger> currentValue = new Stack<BigInteger>();

public Calculator() {
    currentValue.push(new BigInteger("0"));
}

public void press(String value) {
    StringBuffer buffer = new StringBuffer(top().toString());
    buffer.append(value);
    currentValue.pop();
    currentValue.push(new BigInteger(buffer.toString()));
}

public BigInteger top() {
    return currentValue.lastElement();
}

public void push(BigInteger newValue) {
    currentValue.push(newValue);
}
```

Listing 7

0, 1, Many

There's an old guideline that says 0, 1, and many are the easiest numbers to deal with. That's true for generalization as well. For example, if we're implementing an operation on a composite, we might generalize based on depth—moving from handling a depth of one to handling arbitrary depths. As another example, putting the stack in the `Calculator` let us move to remembering many values. It's common to move from handling one object to handling many.

Unify Special Cases

Sometimes, you've got special treatment in two different cases, but there's a unification that lets two branches of computation use the same code if conditions are right. *A null object* (a real object that has no effect) is a common way to do this. Or you may be able to generalize code to cover more cases.

For example, one project I worked on had an algorithm that looked for matches at particular positions in a sequence. There was a subclass algorithm that did a similar search but broadened the list of candidates to include neighbors of the original positions. We were able to reduce this to one case—search within a specified distance. The original "no-neighbor" case became a general search with a "neighbor distance" of 0.

Conclusion

Systematic code transformations are an important tool for TDD and elsewhere. Refactoring provides us a systematic way to improve code, supporting incremental development. Generalization allows us to add new capabilities to our software. The techniques are related, and both belong in your repertoire. **{end}**

Bill Wake is a software coach and author. He manages software development at Gene Codes Forensics, Inc., a bioinformatics software company located in Ann Arbor, Michigan. Bill's Web site is www.xp123.com, and his email address is William.Wake@acm.org. *Bill thanks Tom Kubit, Kent Beck, and an anonymous reviewer for their feedback on this article.*

Sticky Notes

For more on the following topic go to www.StickyMinds.com/bettersoftware.

- References