

# BETTER SOFTWARE

The Print Companion to  StickyMinds.com

## WE'RE ALL IN THIS TOGETHER

Create Agile subteams that work for the greater good

PAGE 34

## THE NOT-SO-GREAT DIVIDE

Factors to help you span the gap between disciplines

PAGE 12



# The Cream of the Crop

PAGE 28

*Milking Reuse for all It's Worth*

# Reduce Stress, Write A Test

by Mike Clark

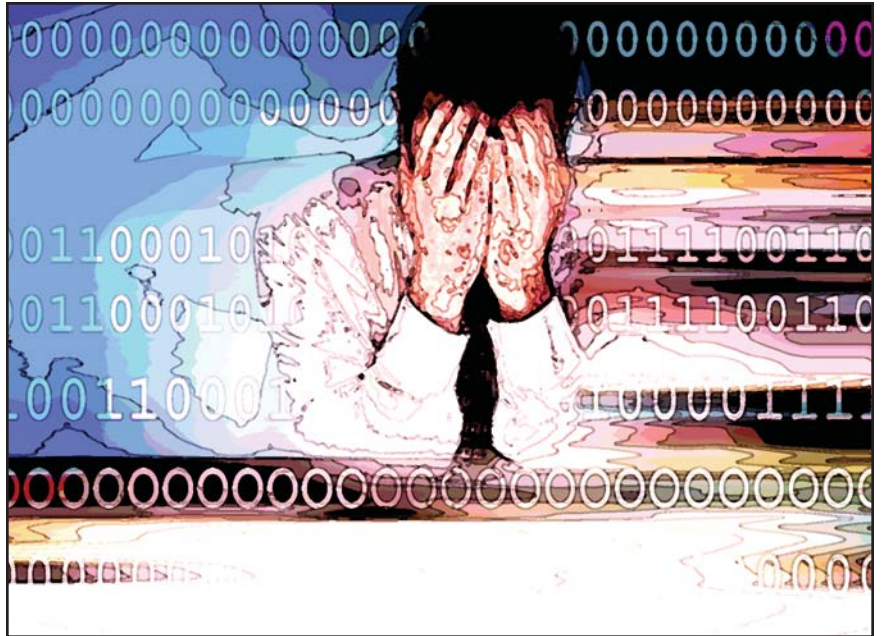
Let's take stock of our progress. So far we've been practicing craft techniques that help keep code responsive to change: choosing good names and making the code expressive. If only it were that easy. The next person faced with modifying the code—to fix a bug, add a feature, etc.—may appreciate our efforts, but clean, well-documented code simply isn't enough.

Here's the rub: Pristine code is still bad code if we don't have the confidence to modify it predictably. Every time you touch the code a cold chill runs up your spine and your typing fingers tremble with fear. You've been bitten before—an innocent change here unknowingly breaks something over there, and what you thought would take mere minutes has turned into several painstaking hours. And so the police tape is strung permanently around the code, workarounds are used to avoid losing one's sanity, and before long what once was the cleanest code on the project has decayed into the nastiest code imaginable.

It doesn't have to be this way. Automated tests drive out fear and give you confidence to change code with impunity. While it may seem like writing tests will slow you down, you actually end up delivering higher quality software—and faster. So before we go any further, let's start writing some tests.

## Got a Test for That?

Say you're working on a digital music player. Among other things, it lets you add your favorite songs to a playlist. It can tell you how long you could listen to the playlist without a song repeating. (Hey, it's a full-featured player.) Here's the Java code that makes all that magic possible:



```
public class Playlist {

    private Collection songs;

    public Playlist() {
        songs = new ArrayList();
    }

    public void addSong(Song song) {
        songs.add(song);
    }

    public String playTime() {

        int totalSeconds = 0;

        // Loop through all the songs accumulating their durations
        for (Iterator i = songs.iterator(); i.hasNext();) {
            Song song = (Song)i.next();
            totalSeconds += song.getDuration();
        }

        // Calculate the units of time
        int hours = totalSeconds / 3600;
        int mins = totalSeconds / 60 % 60;
        int secs = totalSeconds % 60;

        // Return a formatted time string
        return hours + " hrs " + mins + " mins " + secs + " secs";
    }
}
```

The `Playlist` class is simply a container of `Song` objects. The `playTime()` method accumulates the song durations and formats the resulting time in hours, minutes, and seconds. (Ideally, the user interface would handle the formatting, but assume this is a published interface we can't change right now.)

The code is relatively clear and easy to understand. Unfortunately, for all of its virtues, this code is a liability because it isn't covered by a test. We can fix that.

### Trust, but Verify

Now that we've invested time into understanding what the `Playlist` can do, we need to write a test to codify our understanding. The test increases the value of the code and helps the next person who comes along. So let's create a checked example for how to use the `Playlist` class by writing a JUnit test.

```
public class PlaylistTest extends TestCase {
    public void testPlayTime() {
        Playlist playlist = new Playlist();

        playlist.addSong(new Song("Caribbean Amphibian", 210, 20));
        playlist.addSong(new Song("Bein' Green", 100, 10));

        assertEquals("0 hrs 5 mins 10 secs", playlist.playTime());
    }
}
```

This test creates a `Playlist` and adds two songs sung by a famous amphibian vocalist. (No Muppets were harmed during the writing of this column.) The second parameter of the `Song` constructor is the duration of the song in seconds, and the third parameter is the number of times the song has been played. The test then asserts that calling the `playTime()` method returns the total duration of the two songs in a properly formatted string.

Seeing is believing, so let's run the test. Depending on which JUnit test runner you use, you'll either get a comforting green bar or a humble "OK" message.

```
.
Time: 0.011
OK (1 test)
```

Hmm, that was almost too easy. Did it really test anything? It's always good to

make sure, so let's comment out adding the second song and re-run the test.

```
.F
Time: 0.021
There was 1 failure:
1) testPlayTime(PlaylistTest):
   expected:<...5 mins 1...> but was:<...3 mins 3...>
FAILURES!!!
Tests run: 1, Failures: 1, Errors: 0
```

OK, that's good! We expected a time representing both songs, but the `playlist` returned the time for one song. Now we know that the test is actually, er, testing something. Indeed, we've just created an automated change detector—if we unknowingly break the code, the test will scream. We reset the test back to adding two songs and re-run the test to make sure it passes.

### Write a Test, Make It Pass

It's a good thing we understand the code

This test adds the same songs as before, but then asserts that the

`listenTime()` method returns a time equal to the sum of all song durations multiplied by their play count. Of course, the test won't pass until we implement the `listenTime()` method. Barring returning a hard-coded answer, what's the easiest way to get the test to pass? The solution is similar to the code in the `playTime()` method. The only difference is in the `for` loop: Each song duration also needs to be multiplied by the song's play count. So we copy the `playTime()` method, change its name to `listenTime()`, and tweak the line that calculates the time.

```
totalSeconds +=
    song.getDuration() *
    song.getPlayCount();
```

Then we run the test.

```
..
Time: 0.013
OK (2 tests)
```

It passes. So we're done, right? Not so fast. The code works, but along the way we introduced two pieces of duplication. This is bad because duplication drives up the cost of change, so we must remove duplication at all costs.

### Remove Duplication

First, in the `PlaylistTest`, each test method creates a `Playlist` containing the same two `Song` objects. We can remove this duplication by extracting the common code into a `setUp()`

```
public void testListenTime() {
    Playlist playlist = new Playlist();

    playlist.addSong(new Song("Caribbean Amphibian", 210, 20));
    playlist.addSong(new Song("Bein' Green", 100, 10));

    assertEquals("1 hrs 26 mins 40 secs", playlist.listenTime());
}
```

method and creating a playlist instance variable for use by all the methods.

Our second piece of duplication is in the Playlist where both the playTime()

the duplication in the glow of the green bar. But we'll tackle that next time . . . **{end}**

```
public class PlaylistTest extends TestCase {

    private Playlist playlist;

    public void setUp() {
        playlist = new Playlist();
        playlist.addSong(new Song("Caribbean Amphibian", 210, 20));
        playlist.addSong(new Song("Bein' Green", 100, 10));
    }

    public void testPlayTime() {
        assertEquals("0 hrs 5 mins 10 secs", playlist.playTime());
    }

    public void testListenTime() {
        assertEquals("1 hrs 26 mins 40 secs", playlist.listenTime());
    }
}
```

The setUp() method is automatically called by JUnit before each test method. This keeps each test isolated from changes made to the test fixture by other tests. We've changed code to get here, so we re-run the test to make sure we haven't broken anything.

and listenTime() methods contain much of the same code. Our copy/paste reuse was a quick way to get to a passing test, but the thrill is over. Thankfully, with tests in place to detect unwanted changes in those methods, we can safely and accurately eliminate

Mike Clark is a consultant, speaker, programmer, and the author of Pragmatic Project Automation (The Pragmatic Bookshelf, 2004). He is a frequent speaker at software development conferences and the creator of several popular open source tools. Mike helps teams build better software faster through his company, Clarkware Consulting, Inc. (<http://clarkware.com>). Contact him at [mike@clarkware.com](mailto:mike@clarkware.com).

### Don't Stop Now!

Log on to **StickyMinds.com** and join Mike Clark and your peers in a conversation about this issue's topic. At the end of the digital column, add your views or just read what others have to say.

## Our testing processes ferret out the most elusive bugs in your software products...



...So that you don't have to fret about quality.

As a strategic outsourced test partner, Disha Technologies is helping companies such as Microsoft, HP, Google, Yahoo!, Amazon amongst others to optimize their use of in-house resources, intelligently manage headcount, overcome budget and time constraints and meet the diverse requirements of building a high quality product.

A pioneer in independent testing and quality assurance, Disha offers world-class engineering and consulting services to help customers in their test strategy, planning and execution.

At Disha, we do testing **only**, not testing also. We focus on developing processes and methodologies that optimize the testing function, including strategy, planning, automation and execution. What's more, you gain through the insight and experience of our career test engineers.

We are proud of our core values of quality, integrity, objectivity and technical excellence, and an unwavering commitment to help customers build robust and reliable software products on schedule. Visit [www.disha.com](http://www.disha.com) to learn more.

• Seattle • Santa Clara • St. Louis • London • Pune • Hyderabad • Bangalore

**Disha Technologies**  
Telephone: +1 (425) 867-3900  
[marketing@dishatech.com](mailto:marketing@dishatech.com)

**Disha**  
An **aztec** Company