

# Relocate for Reuse

by Mike Clark

When we last left our code, it was trying to tell us a secret. This isn't that uncommon. Code wants to be responsive to change, and when it doesn't get its way it tries to warn us. But often we aren't listening because it's too easy to plow ahead to the next feature. Unfortunately, in our haste we miss a golden opportunity to improve the code, which is actually the design. And although the consequence of ignoring the code isn't immediately noticeable, it quickly starts to drive up the cost of change.

## Small Methods, Loosely Joined

Last month we went after duplication with a vengeance. The `playTime()` and `listenTime()` methods had gross amounts of duplicated code. So we decomposed `playTime()` into smaller methods through a series of simple refactorings. Whenever we saw a block of code following a comment, we extracted the code into a new, well-named method.

The upshot of extracting duplicated code into distinct methods was that `playTime()` and `listenTime()` ended up sharing the extracted methods to get their work done. Consequently, we whittled down `playTime()` and `listenTime()` to the point that they don't do any work of their own. (See Figure 1.)

```
public String playTime() {
    return formattedTime(playTimeInSeconds());
}

public String listenTime() {
    return formattedTime(listenTimeInSeconds());
}
```

Figure 1

Reading code like this can be frustrating if you're expecting to look inside a method and find a linear, soup-to-nuts sequence of statements. Instead, in this case, you find a one liner that goes two separate directions. Which way do you go?

Well, the path you take depends on what you need to understand. In our



GETTY IMAGES

measly program the `playTime()` method represents a fairly high-level concept—it returns the time it takes to play a playlist. Setting aside the issue of duplication, consider what happens if in the `playTime()` method we inline all the code necessary to accumulate and format the playtime. Readers of the code start at the top of the method with a relatively high-level abstraction in mind. But they're quickly greeted by all the gory details of time calculations and formatting.

By delegating work to two methods—`formattedTime()` and `listenTimeInSeconds()`—we've introduced an additional layer of

be one liners. In this particular case, `playTime()` and `listenTime()` naturally decomposed into two methods, with the result of one method feeding into the other method. Chaining the methods together gave us a one-line method.

## Finding a New Home

As a result of reducing duplication, we ended up adding several methods to the `Playlist` class. (See Figure 2.)

When small methods such as these are created, they appear at first glance to be noisy. We may not have wanted clients of the `Playlist` class to know how it handles time. And by creating these time-related methods, we've broken the encapsulation of the `Playlist` class for the sake of removing duplication. Now, as was done in Figure 2, we can restrict access to the methods using the `private` access modifier. Unfortunately, the methods still increase the overall surface area of the class as viewed from a client's perspective.

Here's where the code is whispering its secret. What do all of the methods above have in common? Or, more importantly, what don't they have in common with the `Playlist` class? Surprisingly, although they all deal with the concept of time, none of them has anything to do with playlists. This is a good sign that they don't belong in the `Playlist` class. Simply by

Now, of course, not all methods will

```
private String formattedTime(int totalSeconds) {
    return toHours(totalSeconds) + " hrs " +
        toMinutes(totalSeconds) + " mins " +
        toSeconds(totalSeconds) + " secs";
}

private int toHours(int totalSeconds) {
    return totalSeconds / 3600;
}

private int toMinutes(int totalSeconds) {
    return totalSeconds / 60 % 60;
}

private int toSeconds(int totalSeconds) {
    return totalSeconds % 60;
}

private int toHours(int totalSeconds) {
    return totalSeconds / 3600;
}
```

Figure 2

## Knowing When to Stop

You can certainly take decomposition too far. Three heuristics can help keep you from doing that:

- Extract methods to remove duplication. If two methods have similar code, make a well-named method that encapsulates the code in one place; then delegate to the new method.
- Extract methods to improve readability. For example, if a block of code within a method needs a comment to make the code more understandable, extract a well-named method instead.
- Only extract methods for the preceding two reasons.

So all we've done this month is move code from one class to another. We now have a `Time` class that groups all the

```
public class Time {

    private int totalSeconds;

    public Time(int totalSeconds) {
        this.totalSeconds = totalSeconds;
    }

    public int toHours() {
        return totalSeconds / 3600;
    }

    public int toMinutes() {
        return totalSeconds / 60 % 60;
    }

    public int toSeconds() {
        return totalSeconds % 60;
    }

    public String toString() {
        return toHours() + " hrs " +
            toMinutes() + " mins " +
            toSeconds() + " secs";
    }
}
```

Figure 3

extracting these small methods out of larger methods, their secret was revealed. So let's just move the time-related methods to a new class that represents time. (See Figure 3.)

```
public String playTime() {
    Time time = new Time(playTimeInSeconds());
    return time.toString();
}

public String listenTime() {
    Time time = new Time(listenTimeInSeconds());
    return time.toString();
}
```

Figure 4

While we're at it, let's rename `formattedTime()` to `toString()`.

Notice that we've promoted all the methods from private to public visibility. When the methods were in the `Playlist` class, we didn't want other classes calling the time-related methods. In fact, we didn't really want other classes to know about the time-related methods in the `Playlist` class. That way, we could change how `Playlist` handles time without potentially breaking the clients of `Playlist`.

But now that we've moved all the time-related methods into the `Time` class, there's no harm in making the methods publicly accessible. In fact, `Time` turns out to be a handy class in our program. Over in the `Playlist` class, we delegate time formatting to the new `Time` class. (See Figure 4.)

time-related code in one location. Any class that needs to calculate or format time can simply reuse the `Time` class. Then, if we have to change how time is handled in our program, we can make the change in the `Time` class without affecting its clients. And that means making the change will take less time. **{end}**

*Mike Clark is a consultant, author, speaker, and programmer. 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 topic. At the end of the digital column, add your views or just read what others have to say.