Evolutionary architecture and emergent design:
Refactoring toward design

Find and harvest hidden design in your code

Neal Ford

May 26, 2009

Earlier installments in this series discuss how unit testing leads you to a better design. But if you already have lots of code, how do you discover design elements lurking within it? The preceding installment talked about building structural targets for your code. In this article, series author Neal Ford expands on those ideas and covers techniques that use refactoring to allow emergent design.

Earlier installments in this series discuss how unit testing leads you to a better design. But if you already have lots of code, how do you discover design elements lurking within it? The preceding installment talked about building structural targets for your code. In this article, series author Neal Ford expands on those ideas and covers techniques that use refactoring to allow emergent design.

View more content in this series

In "Test-driven design, Part 1" and "Test-driven design, Part 2," I covered how testing can lead to better design for new projects. In "Composed method and SLAP," I talked about two critical patterns — composed method and the single level of abstraction principle — that give you an overall target for your code's structure. Keep those patterns in mind. Once you have an existing software project, the path to discovering and harvesting design elements lies in refactoring. In his classic book, Refactoring, Martin Fowler defines refactoring as "a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior" (see Related topics). Refactoring is a structural transformation with a goal in mind. A laudable goal for any project is a code base that is easy to refactor. In this article, I talk about how to use refactoring to find underutilized design lurking hidden in your code.

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.

Unit tests represent the primary safety net that allows you to refactor your code base at will. If you have 100 percent code coverage on your project, you can refactor your code with impunity. If you haven't pursued that level of testing, aggressive refactoring is more dangerous. Localized changes are easy to apply and you can see their immediate effect, but far-away side-effect breakages bedevil you. Software leads to unexpected coupling points, and a small change to one part of
the code can ripple through the code base, causing an error to occur many hundreds of lines of
code away from the change. The confidence to change code and find those far-flung errors is the
hallmark of pervasive unit testing. One 2-year ThoughtWorks project saw the technical lead make
53 different refactorings to the code the day before the project went live. He did that with serene
confidence because the project had comprehensive code coverage.

How do you get your code base in a place where major refactorings are possible? One option is to
refuse to write any more code until you've taken the time to add tests to the entire project. As soon
as you propose that, you'll be fired and you can go work for a company that values unit testing.
This approach may not be optimal. Your next best bet is to make others within your team realize
the value of testing and start slowly adding tests around the most critical parts of your code. Draw
a line in the sand and declare a date in the near future: "Starting next Thursday, our code coverage
will always rise." Every time you write new code, add a test, and every time you fix a bug, write a
test. By gradually adding tests around the most sensitive parts (new features and buggy regions),
you add tests where they do the most good.

Unit tests verify atomic behavior. But what if your code base doesn't adhere to the ideal of
composed method? In other words, what if all your methods have dozens or hundreds of lines
of code, and each method performs numerous tasks? You can use unit-testing frameworks to
write coarse-grained functional tests around those methods, concerning yourself primarily with the
transformation of the method's input and output state. These aren't as good as unit tests because
they don't verify each little piece of behavior, but they are better than nothing. For really critical
sections of your code, you might consider adding some functional tests as a safety net before you
start refactoring.

The mechanics of refactoring are simple, and all the major IDEs now have excellent refactoring
support. The difficulty lies with finding what to refactor. That's what the rest of this article is about.

### Coupling to infrastructure

Everyone in the Java world uses frameworks to kick-start development and supply critical
infrastructure of the best kind (infrastructure you didn't need to write). But a danger lurks within
frameworks, both commercial and open source: they are always trying to get you to couple
yourself too intimately with them, which can make it harder to see the design hidden in your code.

Frameworks and application servers have helper classes that entice you down a path of much
simpler development: if you just import and use some of their classes, it'll be much easier to
complete a particular task. A classic example is Struts, the incredibly popular open source Web
framework. Struts includes a set of helper classes to handle common chores for you. For example,
if you allow your domain classes to extend the Struts ActionForm class, Struts automatically
populates form fields from the request, handles validation and life-cycle events, and performs other
handy behavior. In other words, Struts is offering a trade-off: use our class and your development
work will be much easier. It encourages you to create a structure like the one shown in Figure 1:
Figure 1. Using the Struts ActionForm class

The yellow box includes your domain classes, yet the Struts framework encourages you to extend from ActionForm for its helpful behavior. However, you’ve now hopelessly coupled your code to the Struts framework. You can no longer use your domain class in anything but a Struts application. It also hurts the design of your domain classes because this utility class must now sit at the top of your object hierarchy, not allowing you to use inheritance to consolidate common behavior.

Figure 2 shows a much better approach:

Figure 2. Improved design, using composition to decouple from Struts

In this version, your domain classes have no dependencies on the Struts ActionForm. Instead, an interface defines the semantics for both your domain class and the ScheduleItemForm class, which acts as a bridge between your domain and the framework. Both ScheduleItemImpl and ScheduleItemForm implement the interface, and the ScheduleItemForm class holds a reference to your domain class via composition rather than inheritance. It is allowable for the Struts helper to maintain a dependency to your class, but the inverse is not true: you shouldn’t let your classes have a dependency on the framework. Now, you are free to use your ScheduleItem in other types of applications (a Swing application, a services layer, and so on).

Coupling to infrastructure is easy, common, and pervasive in many applications. Frameworks make it much easier to take advantage of their services when you import their goodies. You should resist the temptation. Idiomatic patterns (defined in earlier installments as the little patterns that occur in your application) are harder to discover in your code if the veneer of a framework overlays everything.
**DRY violations**

In the book *The Pragmatic Programmer*, Andy Hunt and Dave Thomas define the DRY principle: Don't Repeat Yourself (see Related topics). Two aspects of DRY violations in code — copy-and-paste coding and structural duplication — can affect design.

**Copy-and-paste code**

Duplication in code obscures the design because you can't find idiomatic patterns. The copy-and-paste code has subtle differences from one place to another, preventing you from determining the actual usage for a method or collection of methods. And, of course, everyone knows that copy-and-paste coding always hurts you in the end because you inevitably have to change the behavior, and it's hard to track down all the places you've copied and pasted code.

How do you find the duplications that have crept into your code base? IDEs either include duplication detectors (as IntelliJ does) or offer them as plug-ins (as Eclipse does). Stand-alone tools also exist, both open source (such as CPD, the Copy/Paste Detector) and commercial (such as Simian) (see Related topics).

The CPD project is part of the PMD source-analysis tool. It is a Swing-based application that analyzes a configurable number of tokens both within individual files and across multiple files. I need a nontrivial victim code base as an example, so I chose the aforementioned Struts project. Running CPD on the Struts 2 code base yields the results shown in Figure 3:

*Figure 3. CPD results on the Struts 2 code base*

CPD found numerous duplications in the Struts code base. Many of them revolve around the addition of portlet support to Struts. In fact, most of the cross-file duplications exist between PortletXXX and XXX (for example, PortletApplicationMap and ApplicationMap). This suggests that the portlet support wasn't well-designed. It is a major code smell any time you have that much duplicated code to add additional behavior to an existing framework. Either inheritance or composition offers a cleaner way to extend the framework, and it's an even bigger indictment if those are both impossible.
The other common duplication problem in this code base resides in the ApplicationMap.java and Sorter.java files. ApplicationMap.java contains a 27-line chunk of duplicated code, shown in Listing 1:

**Listing 1. Duplicated code in ApplicationMap.java**

```java
entries.add(new Map.Entry() {
    public boolean equals(Object obj) {
        Map.Entry entry = (Map.Entry) obj;
        return ((key == null) ?
            (entry.getKey() == null) :
            key.equals(entry.getKey())) && ((value == null) ?
            (entry.getValue() == null) :
            value.equals(entry.getValue()));
    }

    public int hashCode() {
        return ((key == null) ?
            0 :
            key.hashCode()) ^ ((value == null) ?
            0 :
            value.hashCode());
    }

    public Object getKey() {
        return key;
    }

    public Object getValue() {
        return value;
    }

    public Object setValue(Object obj) {
        context.setAttribute(key.toString(), obj);
        return value;
    }
});
```

Besides the multiple use of nested ternary operators (always a good indicator of coding for job security because no one else can read the code), the interesting part of this duplicated code isn't the code itself. It's the preamble that appears before this code in the two methods where the duplication occurs. The first is shown in Listing 2:

**Listing 2. Preamble for the first occurrence of the code**

```java
while (enumeration.hasMoreElements()) {
    final String key = enumeration.nextElement().toString();
    final Object value = context.getAttribute(key);
    entries.add(new Map.Entry() {
        // remaining code elided, shown in Listing 1
    });
}
```

Listing 3 shows the preamble for the second occurrence of the duplication:
Listing 3. The second preamble to the duplicated code

```java
while (enumeration.hasMoreElements()) {
    final String key = enumeration.nextElement().toString();
    final Object value = context.getInitParameter(key);
    entries.add(new Map.Entry() {
        // remaining code elided, shown in Listing 1
    }
}
```

The only difference in the entire while loop is the call to `context.getAttribute(key)` in Listing 2 versus `context.getInitParameter(key)` in Listing 3. Clearly, this could be parameterized, allowing the collapse of the duplicated code into its own method. This example from Struts is a prime illustration of gratuitous copy-and-paste code, which is not only unnecessary but trivially easy to fix.

In fact, this illustrates that the way entries are harvested and added to sets of attributes is an idiomatic pattern in the Struts code base. Allowing almost the same code to reside in multiple places hides the fact that this is something that Struts needs to do all the time, preventing rolling that code up into a more expressive place. One way to clean up the design of multiple classes in the Struts code base is to realize that this idiomatic pattern exists and consolidate that behavior.

### Structural duplications

Another form of duplication is harder to detect and therefore more insidious: structural duplication. Developers who have worked in a limited number of languages (especially languages that have anemic metaprogramming support, such as Java and C#) have an especially hard time seeing this problem. Structural duplication is best summed up by a phrase my co-worker Pat Farley uses: *Same whitespace, different values*. In other words, you've copied code that is virtually the same (that is, the whitespace is the same) but with different values for variables. This duplication doesn't appear in tools like CPD because the values in each instance of the repeated infrastructure are indeed unique, but it harms your code nonetheless.

Here is an example. Suppose I have a simple employee class with a few fields, as shown in Listing 4:

**Listing 4. A simple employee class**

```java
public class Employee {
    private String name;
    private int salary;
    private int hireYear;

    public Employee(String name, int salary, int hireYear) {
        this.name = name;
        this.salary = salary;
        this.hireYear = hireYear;
    }

    public String getName() { return name; }
    public int getSalary() { return salary; }
    public int getHireYear() { return hireYear; }
}
```
Given this simple class, I want the ability to sort on any field of the class. The Java language has a mechanism for differing sort orders via creating comparator classes that implement the Comparator interface. Comparators for name and salary appear in Listing 5:

### Listing 5. Comparators for name and salary

```java
public class EmployeeNameComparator implements Comparator<Employee> {
    public int compare(Employee emp1, Employee emp2) {
        return emp1.getName().compareTo(emp2.getName());
    }
}

public class EmployeeSalaryComparator implements Comparator<Employee> {
    public int compare(Employee emp1, Employee emp2) {
        return emp1.getSalary() - emp2.getSalary();
    }
}
```

To a Java developer, this looks perfectly natural. However, consider the view of the code in Figure 4, where I have superimposed the two comparators:

#### Figure 4. Superimposed comparators

```java
public class EmployeeSalaryComparator implements Comparator<Employee> {
    public int compare(Employee emp1, Employee emp2) {
        return emp1.getSalary() - emp2.getSalary();
    }
}
```

As you can see, the *same whitespace, different values* expression applies quite nicely. Most of the code is duplicated; the only unique part is the return value. Because I'm using the comparison infrastructure in a "natural" way (that is, the way it was intended by the language designers), it's hard to see the duplication naturally, but it is clearly there. Perhaps it isn't so bad with just three properties, but what if it grew to lots of properties? At what point do you decide to attack this duplication, and how do you go about it?

I'm going to use reflection to create a generic sorting infrastructure that doesn't have so much duplicated, boilerplate code. To that end, I create a class to handle both the sorting and generation of comparators for each of the fields automatically. **Listing 6** shows the `EmployeeSorter` class:

### Listing 6. The EmployeeSorter class

```java
public class EmployeeSorter {
    public void sort(List<DryEmployee> employees, String criteria) {
        Collections.sort(employees, getComparatorFor(criteria));
    }

    private Method getSelectionCriteriaMethod(String methodName) {
        methodName = "get" + methodName.substring(0, 1).toUpperCase() + methodName.substring(1);
        try {
            Method m = DryEmployee.class.getMethod(methodName);
        } catch (NoSuchMethodException e) {
            throw new RuntimeException(e.getMessage());
        }
        return m;
    }
}
```
The `sort()` method uses the `Collections.sort()` method, passing the list of employees and a generated comparator, calling this class's third method. The `getComparatorFor()` method acts as a factory for generating an anonymous comparator class on the fly, based on the passed-in criteria. It uses reflection via the `getSelectionCriteriaMethod()` to retrieve the correct `get` method from the employee class, invokes that method on each of the two instances under comparison, and returns the result. The unit tests in Listing 7 show this class in action for a couple of the fields:

### Listing 7. Tests for the generic comparators

```java
public class TestEmployeeSorter {
    private EmployeeSorter _sorter;
    private ArrayList<DryEmployee> _list;

    @Before public void setup() {
        _sorter = new EmployeeSorter();
        _list = new ArrayList<DryEmployee>();
        _list.add(new DryEmployee("Homer", 20000, 1975));
        _list.add(new DryEmployee("Smithers", 150000, 1980));
        _list.add(new DryEmployee("Lenny", 100000, 1982));
    }

    @Test public void name_comparisons() {
        _sorter.sort(_list, "name");
        assertThat(_list.get(0).getName(), is("Homer"));
        assertThat(_list.get(1).getName(), is("Lenny"));
        assertThat(_list.get(2).getName(), is("Smithers"));
    }

    @Test public void salary_comparisons() {
        _sorter.sort(_list, "salary");
        assertThat(_list.get(0).getSalary(), is(20000));
        assertThat(_list.get(1).getSalary(), is(100000));
        assertThat(_list.get(2).getSalary(), is(150000));
    }
}
```

Using reflection like this represents a trade-off: complexity versus conciseness. The reflection-based version is initially harder to understand, but it provides several benefits. First, it automatically handles any property for the `Employee` class, both current and future. With this code in place, you can safely add new properties to `Employee` without having to worry about creating comparators to sort them. Second, this handles a large number of properties more efficiently.
Tolerating structural duplication is easy if it isn’t egregious. But you have to ask yourself: what is the threshold number of properties that justifies using reflection to solve this problem? Is it 10, 20, or 50 properties? This number will vary among developers and teams. However, if you are looking for a more or less objective measure, why not measure how complex the reflection version is versus the individual comparators?

In "Test-driven design, Part 2," I introduced the cyclomatic complexity metric, a simple measure of the relative complexity of a single method. A nice open source tool that measures cyclomatic complexity for the Java language is the open source JavaNCSS tool (see Related topics). If I run JavaNCSS on one of the single comparator classes, it returns 1, which is not surprising: the single method in the class has only a single line and no blocks. When I run JavaNCSS on the entire EmployeeSorter class, the sum of the cyclomatic complexity of all the methods is 8. That suggests that a reasonable threshold for the number of properties to move to reflection is 9; that’s when the complexity of the structural outweighs the complexity of the reflection based version. If reflection makes you queasy, you might add a few more points for the nausea factor!

At any rate, each solution has both costs and benefits associated with it, and it’s up to you to make that trade-off. I am accustomed to reflection in the Java language and other languages, so I tend to go toward that solution more aggressively because I dislike repetition of all kinds in software.

Summary

In this installment, I started the discussion about using refactoring as a tool to help understand and identify emergent design. I covered coupling to infrastructure and the damage it entails for your design. Most of the article covered duplication in several different guises. The intersection of refactoring and design is a rich field; the next installment continues this conversation by discussing how metrics can help you find the parts of your code most in need of refactoring, and thus most likely to contain idiomatic patterns awaiting discovery.
Related topics

- *The Productive Programmer* (Neal Ford, O'Reilly Media, 2008): This book expands on a number of the topics in this series.
- **PMD**: Download PMD (including CPD).
- **JavaNCSS**: This command line-utility measures two standard source code metrics for the Java programming language.