O'REILLY®

Free Sampler

CSS Refactoring

ARCHITECT YOUR STYLESHEETS FOR SUCCESS

Steve Lindstrom

CSS Refactoring
by Steve Lindstrom

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Meg Foley
Production Editor: Kristen Brown
Copyeditor: Rachel Head
Proofreader: Molly Ives Brower
Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

November 2016: First Edition

Revision History for the First Edition
2016-11-08: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491906422* for release details.

978-1-491-90642-2

[LSI]

Table of Contents

CHAPTER 1

Refactoring and Architecture

This is the starting point of our CSS refactoring journey. In this chapter we'll learn what refactoring is and how it relates to software architecture. We'll also discuss the importance of refactoring and some of the reasons why your code might need it, and we'll also work through two refactoring examples to drive these concepts home.

What Is Refactoring?

Refactoring is the process of rewriting code in order to make it more simple and reusable without changing its behavior. It's a vital skill to have if you're writing code because you will have to do it at some point, whether you want to or not; you may have even already refactored code without realizing it! Since refactoring doesn't change the code's behavior it's understandable to wonder why it's even worth doing in the first place. However, before that question can be answered it's important to understand what software architecture is.

What Is Software Architecture?

Like a living creature, a software system is usually comprised of many smaller pieces that specialize in doing one particular thing. When combined, these smaller pieces work together to create the larger software system. *Software architecture* is the term used for describing how all of the pieces of a software project fit together.

Every piece of software, from a simple website to the control system in a spacecraft, has an architecture, whether it's intentional or not. However, the best architectures are usually planned out well before any coding takes place. Following are some of the most important characteristics of a good architecture.

Good Architectures Are Predictable

Being *predictable* means that accurate assumptions can be made about how the software works and is structured. Predictability is indicative of proper forward planning and will help save on development time because there will be no question as to:

- What a component's responsibilities are
- Where to find a particular piece of code
- Where to put a new piece of code

Because assumptions can be made accurately in a predictable architecture, developers that are unfamiliar with the code should be able to understand it more quickly.

Good Architectures Promote Code Reuse

Code reuse is the ability for code to be used in multiple places without being duplicated. Code reuse is beneficial because it speeds up development time, since you don't have to rewrite pieces of code that already exist. Similarly, the fewer pieces of code you have that solve a particular problem, the less time you will have to spend maintaining all of those implementations. For example, if you discover a bug in a piece of code that gets reused across a project, you know that bug will be present wherever that code is used. But by fixing it in one place, you'll fix it in all of the places that piece of code is used.

Good Architectures Are Extensible

Extensibility is a principle of good architecture because it allows for the system to have new functionality built upon it with ease. Most software isn't built from start to finish in one day, so it's very important that it can be built incrementally without requiring major structural changes. If your project frequently requires significant changes to its architecture it becomes much more difficult to release.

Good Architectures Are Maintainable

Much like extensibility, *maintainability* is very important to an architecture because it allows you to modify existing functionality with ease. Over time requirements may change, and you will be forced to modify your code. Having maintainable software means that you will be able to modify one piece of your code without necessarily having to drastically change all of the other pieces.

Software Architecture and Refactoring

In a nutshell, refactoring exists to help maintain and promote good software architecture. It is nothing more than a set of techniques that can be used to reorganize code

into a more meaningful structure with the intention of making it more predictable, reusable, extensible, and maintainable. When your software's architecture displays the aforementioned characteristics it will be much more reliable for its intended users, and it will be much more enjoyable for you to work on too.

Shortcomings that Lead to Refactoring

Why isn't code just written correctly in the first place so there's no need to refactor it later? Despite our best efforts to design and write the highest-quality code possible, over time *something* will change that requires refactoring. Let's take a look at a few of the causes.

Changing Requirements

Over time software systems evolve as the result of changing requirements. When software is written to satisfy one set of requirements, it likely doesn't take things into consideration that would satisfy another set of requirements that have not yet been written (nor should it). As such, when requirements change so must the code, and if there are time constraints present then code quality might suffer as a result of cutting corners.

Poorly Designed Architecture

Even if you're aware of what makes a good architecture, it's not always feasible to spend a significant amount of time planning everything out. And if you don't have a clear picture of how everything should work together from the beginning, you may have to do some refactoring down the road. It's also fairly common to build a new feature really quickly (which can result in cutting corners) to see if it gets traction with users and then either clean up the code later if it does or remove it if it doesn't.

Underestimating Difficulty

Estimating how long software development will take is difficult, and unfortunately these estimates are often used to create schedules. When a project's timescale is underestimated it puts pressure on developers to "just get it done," which leads to writing code quickly without putting too much thought into it. If this happens frequently enough even the best code can turn into a big plate of "spaghetti code" that's difficult to understand and unruly to manage.

Ignoring Best Practices

It can be difficult to stay up to date with every best practice, especially if your job encompasses many technologies and/or managing people. If you're working on a team and overlook a best practice, then hopefully you'll have a colleague that will

make you aware of it. If the opportunity to use a best practice is missed, then at some point in the future you may have to revisit your code and do some refactoring.

Difficulty Keeping Up with Best Practices

Technology changes at a very rapid pace, and as a result a technique that was once considered a best practice might not be anymore. For example, before 2011 if you wanted to display a container that appeared to have rounded corners on a website, you would need to have an image for each of the corners, embed the images in your HTML, and then position them using CSS to make sure everything lined up correctly. Today this technique is obsolete because modern browsers can display rounded corners with the `border-radius` CSS property. If you don't continually update your code to make use of modern best practices, over time your technical debt will build up and you'll find your code in a much worse state than it otherwise could be.

When Should Code Be Refactored?

Refactoring code is much easier when it's done with context. As such, it's usually best to refactor when you're fixing a bug or building a new feature that makes use of existing code. Refactoring code consistently while working on smaller tasks reduces the likelihood of breaking anything, and those who modify the same code after it's been refactored will also benefit from your work. Over time, consistent refactoring will lead to superior code, provided your changes align with the properties of good architecture.

However, sometimes you'll run across a piece of code that has a lot of dependencies, and you may be faced with the decision of whether or not you should refactor. Refactoring a piece of code that has a lot of dependencies can be like pulling a loose thread on a shirt: the more you pull the thread, the more it unravels. Similarly, the more you modify a piece of code that has a lot of dependencies, the more dependencies you'll end up having to update. In situations like this, if you're up against a tight deadline it might be beneficial to get your work done in time first, and then allocate some time to go back and refactor. However, if you find along the way that there are smaller things that can be refactored without adversely affecting your schedule, you might consider refactoring them now.

When Should Code NOT Be Refactored?

Knowing when *not* to refactor code is probably even more important than knowing when it should be refactored. Refactoring can have a bad reputation because often software developers seem to rewrite code just for the sake of rewriting it. Maybe someone else wrote the code and the person doing the unnecessary refactoring is suf-

fering from a case of Not Written Here Syndrome, where they feel the code is inferior because they didn't write it. Or perhaps one day someone decides that they just don't like the way they've written code previously (maybe they used underscores instead of dashes in class names and now want to do the opposite), so they embark down the rabbit hole of changing things to scratch this itch. In many cases this can be considered "fake work" that makes people feel productive even when they aren't. In Chapter 5 we'll discuss how to form a plan for how your code should be written by drafting a set of coding standards. At that point it will be much clearer that you should only refactor when doing so will improve your architecture or if it aligns with your coding standards.

Am I Allowed to Refactor My Code?

If you're working on a personal project, then the answer is a resounding "yes!"—but if you're working for an organization where you're not necessarily in charge, the answer might not be as clear. In a perfect world every organization would understand the importance of refactoring, but often that's not the reality. If colleagues in your organization lack technical knowledge about refactoring, you might try to educate them; I hear *CSS Refactoring* books make nice gifts!

Reasonable people that are responsible for ensuring software ships with high-quality code will likely get it, but those that don't may argue that:

- Spending time to rewrite code without seeing changes is a waste of time and money.
- If it's not broken, it doesn't need to be fixed.
- You should have written the code correctly the first time.

If you encounter any of these arguments and you feel confident enough to do so, my advice is to refactor your code anyway, provided you stay on schedule and are careful not to break anything. If you've heard statements like these, I'm willing to bet the person making them has never participated in a code review, so your changes probably won't be noticed anyway. However, if you're refactoring code just for the sake of refactoring, you may consider waiting until it becomes more apparent that the changes will be necessary; premature optimization can often be just as bad as technical debt.

Refactoring Examples

Now that you have a general idea of the benefits of refactoring and when it is (and isn't) a good idea to do it, we can start to talk about how you go about refactoring your code.

Although this book is about refactoring CSS, it's much easier to initially analyze the concept with code that calculates a discrete value as opposed to code that changes the appearance of HTML elements. So, our first example will demonstrate refactoring some basic JavaScript that calculates the total price of an ecommerce order. The second example will refactor some CSS.

Code Examples

Because it can be difficult to understand what's going on in long code passages that span multiple pages and files, smaller pieces of code will be used for examples in this book. All the JavaScript code from our first example can be embedded in an HTML file to make execution easier.

For more complicated examples, CSS that is used to define the general look and feel of the elements in the examples will be included using a separate CSS file.

Styles in this book that are included inline between `<style>` and `</style>` tags will be directly relevant to the example at hand and will be used to illustrate a granular concept.

All code examples are available online at the book's companion website.

Refactoring Example 1: Calculating the Total Price of an Ecommerce Order

Example 1-1 contains some JavaScript that calculates the total price of an ecommerce order if provided with:

- The price of each item purchased
- The quantity of each item purchased
- The cost to ship each item purchased
- The customer's shipping information
- An optional discount code that can reduce the price of the order

Example 1-1. Calculating an ecommerce order total

```
/**
 * Calculates the total order price after shipping costs, discounts, and
 * taxes are applied.
 *
 * @param {Object} customer - a collection of information about
 *   the person that placed the order.
```

```
 *
 * @param {Array.<Object>} lineItems - a collection of products
 *   and quantities being purchased as well as the cost to ship one unit.
 *
 * @param {string} discountCode - an optional discount code that can trigger
 *   a discount to be deducted before shipping and tax are added.
 */
var getOrderTotal = function (customer, lineItems, discountCode) {
  var discountTotal = 0;
  var lineItemTotal = 0;
  var shippingTotal = 0;
  var taxTotal = 0;

  for (var i = 0; i < lineItems.length; i++) {
      var lineItem = lineItems[i];
      lineItemTotal += lineItem.price * lineItem.quantity;
      shippingTotal += lineItem.shippingPrice * lineItem.quantity;
  }

  if (discountCode === '20PERCENT') {
      discountTotal = lineItemTotal * 0.2;
  }

  if (customer.shiptoState === 'CA') {
      taxTotal = (lineItemTotal - discountTotal) * 0.08;
  }

  var total = (
      lineItemTotal -
      discountTotal +
      shippingTotal +
      taxTotal
  );

  return total;
};
```

Calling `getOrderTotal` using the data in Example 1-2 results in `Total: $266` being printed. Example 1-3 explains why that result is printed.

Example 1-2. Running getOrderTotal with test input

```
var lineItem1 = {
  price: 50,
  quantity: 1,
  shippingPrice: 10
};

var lineItem2 = {
  price: 100,
  quantity: 2,
```

```
  shippingPrice: 20
};

var lineItems = [lineItem1, lineItem2];

var customer = {
  shiptoState: 'CA'
};

var discountCode = '20PERCENT';

var total = getOrderTotal(customer, lineItems, discountCode);

document.writeln('Total: $' + total);
```

Example 1-3. Explanation of why getOrderTotal prints "Total: $266"

```
discountTotal = 0
lineItemTotal = 0
shippingTotal = 0
taxTotal = 0

# FOR LOOP 1st iteration:
lineItemTotal = 0 + (50 * 1) = 50
shippingTotal = 0 + (10 * 1) = 10

# FOR LOOP 2nd iteration:
lineItemTotal = 50 + (100 * 2) = 250
shippingTotal = 10 + (20 * 2) = 50

# discountTotal gets calculated because discountCode equals "20 PERCENT":
discountTotal = 250 * 0.2 = 50

# taxTotal gets set because customer.shiptoState equals "CA":
taxTotal = (250 - 50) * 0.08 = 16

total = 250 - 50 + 50 + 16 = 266
```

Unit tests

After walking through the calculations, the math checks out and everything appears to be working as expected. To ensure that things continue working over time, we can now write a unit test. Put simply, a *unit test* is a piece of code that executes another piece of code to make sure everything is working as expected. Unit tests should be written to test singular pieces of functionality in order to narrow down the root cause of any issues that may surface. Further, a suite of unit tests that are written for your entire project should be run before releasing new code so bugs that have been introduced into the system can be discovered and fixed before it's too late.

The input data from Example 1-2 can be used to write a unit test, shown in Example 1-4, that asserts the function returns the expected value (266). After the test is done running, a count of how many successful and unsuccessful tests were run in addition to a list of unsuccessful tests will be printed.

Example 1-4. A unit test for getOrderTotal

```
var successfulTestCount = 0;
var unsuccessfulTestCount = 0;
var unsuccessfulTestSummaries = [];

/**
 * Asserts the calculations in `getOrdertotal()` are correct.
 */
var testGetOrderTotal = function () {

    // set up expectations

    var expectedTotal = 266;

    // set up test data

    var lineItem1 = {
        price: 50,
        quantity: 1,
        shippingPrice: 10
    };

    var lineItem2 = {
        price: 100,
        quantity: 2,
        shippingPrice: 20
    };

    var lineItems = [lineItem1, lineItem2];

    var customer = {
        shiptoState: 'CA'
    };

    var discountCode = '20PERCENT';

    var total = getOrderTotal(customer, lineItems, discountCode);

    // test the results against expectations

    if (total === expectedTotal) {
      successfulTestCount++;
    } else {
      unsuccessfulTestCount++;
      unsuccessfulTestSummaries.push(
```

```
            'testGetOrderTotal: expected ' + expectedTotal + '; actual ' + total
        );
    }
};

// run tests

testGetOrderTotal();
document.writeln(successfulTestCount + ' successful test(s)<br/>');
document.writeln(unsuccessfulTestCount + ' unsuccessful test(s)<br/>');

if (unsuccessfulTestCount) {
    document.writeln('<ul>');
    for(var i = 0; i < unsuccessfulTestSummaries.length; i++) {
        document.writeln('<li>' + unsuccessfulTestSummaries[i] + '</li>');
    }
    document.writeln('</ul>');
}
```

Executing `testGetOrderTotal` results in the test successfully passing the assertion, as can be seen in Figure 1-1.

Figure 1-1. Successful unit test results

However, if in the future for some reason a bug was introduced and the multiplier used in the calculation of `discountTotal` changed from 0.2 to –0.2, this would no longer be the case and we would instead see the result pictured in Figure 1-2.

Figure 1-2. Unsuccessful unit test results

Unit tests are a powerful way to ensure that your system continues working as expected over time. They can be especially helpful when rewriting code because an assertion will already be documented, and that assertion will provide greater confidence that the code's behavior hasn't changed.

Now that we understand the code used to calculate the total price of an ecommerce order and we have an accompanying unit test, let's see how refactoring can improve things.

Refactoring getOrderTotal

Looking closely at `getOrderTotal` reveals that there are a number of calculations being performed in that one function:

- The total discount to be subtracted from the final price
- The total price for all of the line items

- The total shipping costs
- The total tax costs
- The total order price

If a bug is accidentally introduced into one of those five calculations, the unit test (`testGetOrderTotal`) will indicate that something went wrong, but it won't be obvious what *specifically* went wrong. This is the main reason why unit tests should be written to test single pieces of functionality.

To make the code more granular, each of the aforementioned calculations should be extracted into a separate function that has a name describing what it does, like in Example 1-5.

Example 1-5. Extracting code fragments into new functions

```
/**
 * Calculates the total price of all line items ordered.
 *
 * @param {Array.<Object>} lineItems - a collection of products
 *   and quantities being purchased and the cost to ship one unit.
 *
 * @returns {number} The total price of all line items ordered.
 */
var getLineItemTotal = function (lineItems) {
    var lineItemTotal = 0;

    for (var i = 0; i < lineItems.length; i++) {
        var lineItem = lineItems[i];
        lineItemTotal += lineItem.price * lineItem.quantity;
    }

    return lineItemTotal;
};

/**
 * Calculates the total shipping cost of all line items ordered.
 *
 * @param {Array.<Object>} lineItems - a collection of products
 *   and quantities being purchased and the cost to ship one unit.
 *
 * @returns {number} The total price to ship of all line items ordered.
 */
var getShippingTotal = function (lineItems) {
    var shippingTotal = 0;

    for (var i = 0; i < lineItems.length; i++) {
        var lineItem = lineItems[i];
        shippingTotal += lineItem.shippingPrice * lineItem.quantity;
    }
```

```
    return shippingTotal;
};

/**
 * Calculates the total discount to be subtracted from an order total.
 *
 * @param {number} lineItemTotal - The total price of all line items ordered.
 *
 * @param {string} discountCode - An optional discount code that can trigger a
 *   discount to be deducted before shipping and tax are added.
 *
 * @returns {number} The total discount to be subtracted from an order total.
 */
var getDiscountTotal = function (lineItemTotal, discountCode) {
    var discountTotal = 0;

    if (discountCode === '20PERCENT') {
        discountTotal = lineItemTotal * 0.2;
    }

    return discountTotal;
};


/**
 * Calculates the total tax to apply to an order.
 *
 * @param {number} lineItemTotal - The total price of all line items ordered.
 *
 * @param {Object} customer - A collection of information about the person that
 *   placed an order.
 *
 * @returns {number} The total tax to be applied to an order.
 */
var getTaxTotal = function () {
    var taxTotal = 0;

    if (customer.shiptoState === 'CA') {
        taxTotal = lineItemTotal * 0.08;
    }

    return taxTotal;
};
```

Each new function should also have an accompanying unit test like the one in Example 1-6.

Example 1-6. Unit tests for extracted functions written in JavaScript

```
/**
 * Asserts getLineItemTotal works as expected.
 */
var testGetLineItemTotal = function () {
    var lineItem1 = {
        price: 50,
        quantity: 1
    };

    var lineItem2 = {
        price: 100,
        quantity: 2
    };

    var lineItemTotal = getLineItemTotal([lineItem1, lineItem2]);
    var expectedTotal = 250;

    if (lineItemTotal === expectedTotal) {
      successfulTestCount++;
    } else {
      unsuccessfulTestCount++;
      unsuccessfulTestSummaries.push(
          'testGetLineItemTotal: expected ' + expectedTotal + '; actual ' +
          lineItemTotal
      );
    }
};

/**
 * Asserts getShippingTotal works as expected.
 */
var testGetShippingTotal = function () {
    var lineItem1 = {
        quantity: 1,
        shippingPrice: 10
    };

    var lineItem2 = {
        quantity: 2,
        shippingPrice: 20
    };

    var shippingTotal = getShippingTotal([lineItem1, lineItem2]);
    var expectedTotal = 250;

    if (shippingTotal === expectedTotal) {
      successfulTestCount++;
    } else {
      unsuccessfulTestCount++;
      unsuccessfulTestSummaries.push(
```

```
            'testGetShippingTotal: expected ' + expectedTotal + '; actual ' +
            shippingTotal
        );
    }
};

/**
 * Ensures GetDiscountTotal works as expected when a valid discount code
 * is used.
 */
var testGetDiscountTotalWithValidDiscountCode = function () {
    var discountTotal = getDiscountTotal(100, '20PERCENT');
    var expectedTotal = 20;

    if (discountTotal === expectedTotal) {
      successfulTestCount++;
    } else {
      unsuccessfulTestCount++;
      unsuccessfulTestSummaries.push(
          'testGetDiscountTotalWithValidDiscountCode: expected ' + expectedTotal +
          '; actual ' + discountTotal
      );
    }
};

/**
 * Ensures GetDiscountTotal works as expected when an invalid discount code
 * is used.
 */
var testGetDiscountTotalWithInvalidDiscountCode = function () {
    var discountTotal = get_discount_total(100, '90PERCENT');
    var expectedTotal = 0;

    if (discountTotal === expectedTotal) {
      successfulTestCount++;
    } else {
      unsuccessfulTestCount++;
      unsuccessfulTestSummaries.push(
          'testGetDiscountTotalWithInvalidDiscountCode: expected ' + expectedTotal +
          '; actual ' + discountTotal
      );
    }
};

/**
 * Ensures GetTaxTotal works as expected when the customer lives in California.
 */
var testGetTaxTotalForCaliforniaResident = function () {
    var customer = {
      shiptoState: 'CA'
    };
```

```
    var taxTotal = getTaxTotal(100, customer);
    var expectedTotal = 8;

    if (taxTotal === expectedTotal) {
      successfulTestCount++;
    } else {
      unsuccessfulTestCount++;
      unsuccessfulTestSummaries.push(
          'testGetTaxTotalForCaliforniaResident: expected ' + expectedTotal +
          '; actual ' + taxTotal
      );
    }
};

/**
 * Ensures GetTaxTotal works as expected when the customer doesn't live
 * in California.
 */
var testGetTaxTotalForNonCaliforniaResident = function () {
    var customer = {
        shiptoState: 'MA'
    };

    var taxTotal = getTaxTotal(100, customer);
    var expectedTotal = 0;

    if (taxTotal === expectedTotal) {
      successfulTestCount++;
    } else {
      unsuccessfulTestCount++;
      unsuccessfulTestSummaries.push(
          'testGetTaxTotalForNonCaliforniaResident: expected ' + expectedTotal +
          '; actual ' + taxTotal
      );
    }
};
```

Finally, `getOrderTotal` should be modified to make use of the new functions, as seen in Example 1-7.

Example 1-7. Modifying getOrderTotal to use extracted functions

```
/**
 * Calculates the total order price after shipping costs, discounts, and
 * taxes are applied.
 *
 * @param {Object} customer - a collection of information about
 *   the person that placed the order.
 *
 * @param {Array.<Object>} lineItems - a collection of products
 *   and quantities being purchased and the cost to ship one unit.
```

```
 *
 * @param {string} discountCode - an optional discount code that can trigger
 *   a discount to be deducted before shipping and tax are added.
 */
var getOrderTotal = function (customer, lineItems, discountCode) {
    var lineItemTotal = getLineItemTotal(lineItems);
    var shippingTotal = getShippingTotal(lineItems);
    var discountTotal = getDiscountTotal(lineItemTotal, discountCode);
    var taxTotal = getTaxTotal(lineTtemTotal, customer);

    return lineItemTotal - discountTotal + shippingTotal + taxTotal;
};
```

After analyzing the preceding code, the following observations can be made:

- There are more functions than before.
- There are more unit tests than before.
- Each function does one particular thing.
- Each function has an accompanying unit test.
- Functions can be used together to perform more complex calculations.

Overall, this code is in much better shape now. The individual calculations used in `getOrderTotal` have been extracted and each has an accompanying unit test. This means that it will be much easier to pinpoint exactly which piece of functionality is broken should a bug be introduced into the code. Additionally, if the totals for tax or shipping needed to be calculated in another piece of code, the existing functionality that already has unit tests can be used.

Refactoring Example 2: A Simple Example of Refactoring CSS

Example 1-8 is some code that displays the headline of a website.

Example 1-8. HTML for a website headline

```
<!doctype html>
<html>
  <head>
    <title>Ferguson's Cat Shelter</title>
    <link rel="stylesheet" type="text/css" href="css/style.css" />
  </head>
  <body>
    <main>
      <h1 style="font-family: Helvetica, Arial, sans-serif;font-size: 36px;
        font-weight: 400;text-align: center;">
        San Francisco's Premiere Cat Shelter
      </h1>
    </main>
```

```
  </body>
</html>
```

Opening up a browser and loading *index.html* will display Figure 1-3.

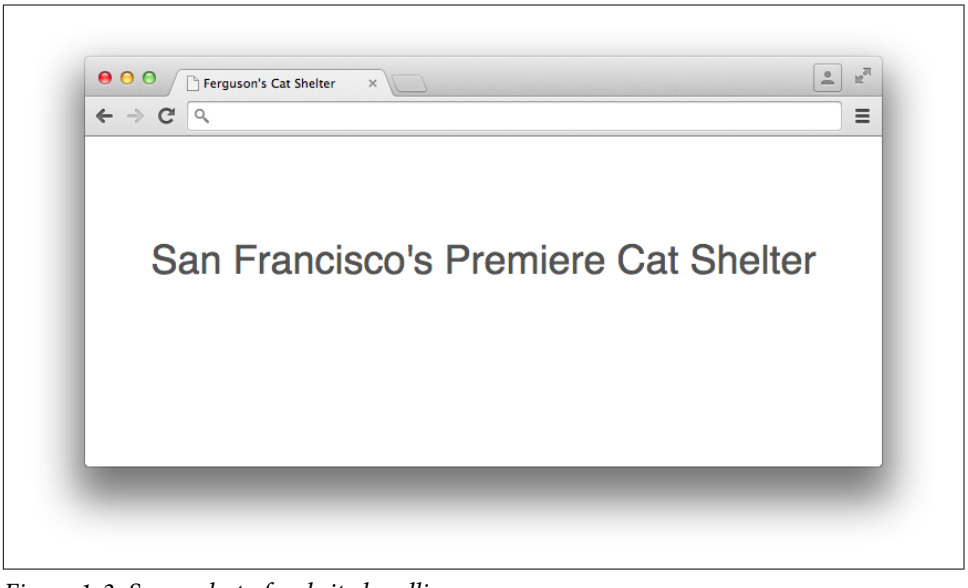


Figure 1-3. Screenshot of website headline

In our first refactoring example we wrote a unit test for the code before refactoring to ensure its behavior didn't change. When refactoring CSS it's still important to make sure that your modifications don't change anything, but unfortunately it's not as straightforward because something visual is being tested rather than something that produces discrete values. Chapter 5 discusses useful techniques for maintaining visual equality. For now, though, simply taking a screenshot to provide a visual reference before refactoring will suffice.

Refactoring the website headline

Looking at the code in Example 1-8, it's clear that there's room for improvement because the headline, denoted by an `<h1>` tag, has its styles embedded in the `style` attribute. When styles are embedded in HTML via an element's `style` attribute or between `<style></style>` tags, they are known as *inline styles*.

Much like the original function in Example 1-1 that performed multiple calculations, inline styles are not very reusable. When styles are set using the `style` attribute, they can only be applied to that particular element. When styles are embedded between `<style></style>` tags, they can only be applied to that particular page.

Because most websites have multiple pages that could each have a headline, these styles should be extracted out of the HTML into a separate CSS file (in this case *style.css*) that can be included on multiple pages and cached by the browser. The contents of *style.css* are depicted in Example 1-9, and Example 1-10 shows the HTML with the inline CSS extracted.

Example 1-9. Headline CSS extracted into style.css

```
h1 {
  font-family: Helvetica, Arial, sans-serif;
  font-size: 36px;
  font-weight: 400;
  text-align: center;
}
```

Example 1-10. HTML with inline CSS extracted

```
<!doctype html>
<html>
  <head>
    <title>Ferguson's Cat Shelter</title>
    <link rel="stylesheet" type="text/css" href="css/style.css" />
  </head>
  <body>
    <main>
      <h1>San Francisco's Premiere Cat Shelter</h1>
    </main>
  </body>
</html>
```

A quick browser refresh shows that nothing has changed, and once again some observations can be made:

- Extracting inline CSS promotes reusability.
- Separating functionality (styles and structure) makes code more readable.
- Regression testing can be performed manually in a web browser or by comparing a refactored interface against a screenshot.

Extracting styles into a separate file promotes code reuse because those styles can be used across multiple pages. When CSS is in a file separate from HTML, both the HTML and the CSS are easier to read because the HTML does not have extremely long lines of style definitions in it, and the CSS is grouped together in logical chunks. Finally, testing of changes can be performed by manually reloading the page in the browser so the changes can be compared against a screenshot that was taken before refactoring.

Although this example was very simple, lots of small changes like this can produce a sizable benefit over time.

Chapter Summary

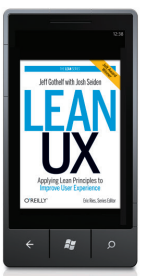
We've made it through the first chapter, and we know what refactoring is and how it relates to software architecture. We also learned why refactoring is important and when it should be performed. Finally, we walked through two refactoring examples and learned about unit tests. Next, we'll learn about the cascade, which is arguably the most important concept to understand when it comes to writing CSS.

O'Reilly ebooks.

Your bookshelf on your devices.

PDF

Mobi



ePub

DAISY

When you buy an ebook through oreilly.com you get lifetime access to the book, and whenever possible we provide it to you in four DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, and DAISY—that you can use on the devices of your choice. Our ebook files are fully searchable, and you can cut-and-paste and print them. We also alert you when we've updated the files with corrections and additions.

Learn more at ebooks.oreilly.com

You can also purchase O'Reilly ebooks through the iBookstore, the Android Marketplace, and Amazon.com.

O'REILLY®