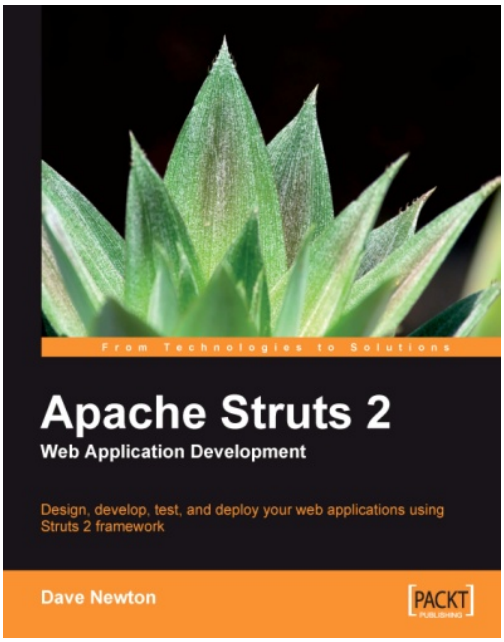
Apache Struts 2
Web Application Development:

Dave Newton



Chapter 15
"Documenting our Application"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter 15. "Documenting our Application"

A synopsis of the book's content

Information on where to buy this book

About the Author

Dave has been programming for as long as he can remember, and probably a bit longer than that. He began by playing a Star Trek game on a library computer, and quickly graduated to educating his grade school teachers about computers and the BASIC language. Receiving a TRS-80 Model 1 was pretty much the end of his social life, and he's never looked back.

A diverse background in both programming languages and problem spaces has provided him with a wide range of experience. Using a diverse set of languages, which includes Pascal, Forth, Fortran, Lisp, Smalltalk, assembly, C/C++, Java, and Ruby, brings an appreciation for many programming paradigms. Working on AI, embedded systems, device drivers, and both desktop and web application programming, has brought him an appreciation for the problems and solutions that each environment offers.

Even after thirty years of typing in his first infinite loop, he's still entertained by the process of programming. He never tires of looking for ways to do less work, and is convinced that he should never have to repeat himself to a computer. He still occasionally writes an infinite loop.

Apache Struts 2
Web Application Development

Struts 2.1 is a modern, extensible, agile web application framework, which is suitable for both small- and large-scale web applications.

The book begins with a comprehensive look at the basics of Struts 2.1, interspersed with detours into more advanced development topics. You'll learn about configuring Struts 2.1 actions, results, and interceptors via both XML and Java annotations. You'll get an introduction to most of the Struts 2.1 custom tags, and also learn how they can assist in rapid application prototyping and development.

From there, you'll make your way into Struts 2.1's strong support for form validation and type conversion, which allows you to treat your form values as domain objects without cluttering your code. A look at Struts 2.1's interceptors is the final piece of the Struts 2.1 puzzle, which allows you to leverage the standard Struts 2 interceptors, as well as implement your own custom behavior.

After covering Struts 2.1, you'll journey into the world of JavaScript (a surprisingly capable language), the Document Object Model (DOM), and CSS, and learn how to create clean and concise client-side behavior. You'll leverage that knowledge as you move on to Struts 2 themes and templates, which give you a powerful way to encapsulate site-wide user interface behavior.

The book closes with a look at some tools that make the application development life cycle easier to manage, particularly in a team environment, and more automatic.

What This Book Covers

Chapter 1 gives us a bird's-eye view of Struts 2 and examines some useful techniques of lightweight, agile development.

Chapter 2 gives an introduction to Struts 2 application configuration, using both XML and annotations. It also covers the beginning of our sample application, RecipeBox.

Chapter 3 covers some of the functionality provided by Struts 2's ActionSupport class, including I18N, and a first look at form validation. It also covers some basic RecipeBox functionality after gathering some user stories.

Chapter 4 examines several common, standard Struts 2 result types. It also covers how to write our own custom result types.

Chapter 5 gives an in-depth look at the generic Struts 2 custom tags. These include tags for iteration, list generation, conditionals, and internationalization (I18N).

Chapter 6 continues our exploration of Struts 2 custom tags, focusing especially on its form tags.

Chapter 7 examines Struts 2 form validation, including both XML and annotation-driven validation. It also teaches more about how Struts 2 converts our form values into domain objects, and shows how to create our own type converters to handle custom data types.

Chapter 8 finishes our comprehensive introduction to Struts 2, by checking out the included Struts 2 interceptors. It also discusses how to write and configure our own interceptors.

Chapter 9 looks at how to handle errors in Struts 2, as well as discusses error and exception handling in general. It also covers some general Java logging topics, focusing on using Apache Commons Logging and Log4J.

Chapter 10 explores how to best leverage JavaScript and how to keep it modular.

Chapter 11 covers the client-side functionality, which depends on more than JavaScript. By using CSS and the DOM effectively, we can accomplish a lot with a minimal amount of code.

Chapter 12 covers Struts 2 themes and templates. The themes and templates in Struts 2 allow for application-wide functionality on the client side, keeping our JSP pages lightweight and adaptable. Rather than writing boilerplate HTML on our pages, we can separate it into themes and templates.

Chapter 13 takes a look at some of Struts 2's built-in support for Ajax using the Dojo tags. It also covers the Struts 2 REST plug-in that furthers our "convention over configuration" path.

Chapter 14 covers how to apply the TDD concepts to several testing aspects, including unit, functional, and acceptance tests.

Chapter 15 looks at many aspects of documentation, including "self-documenting" code, Javadocs, generators, methodologies, and so on, with a focus on automating as much documentation as possible.

15
Documenting our Application

Every developer's favorite task is documenting their application (or so I've heard). As irritating as documentation can be, delivering a complete solution implies comprehensive, usable documentation. This goes beyond (but includes) typical Javadocs. However, more is required in order to understand how a particular application works, how its parts fit together, where dependencies lie, and so on. Even us, the developers, benefit from having a wide variety of documentation available.

In this chapter, we'll look at the ways in which we can document our applications, coding styles that can aid in understanding, tools and techniques for creating documentation from application artifacts, different types of documentation for different parties, and so on.

Documenting Java

Everybody knows the basics of documenting Java, so we won't go into much detail. We'll talk a bit about ways of writing code whose intention is clear, mention some Javadoc tricks we can use, and highlight some tools that can help keep our code clean. Clean code is one of the most important ways we can document our application. Anything we can do to increase readability will reduce confusion later (including our own).

Self-documenting code

We've all heard the myth of self-documenting code. In theory, code is always clear enough to be easily understood. In reality, this isn't always the case. However, we should try to write code that is as self-documenting as possible.

Keeping non-code artifacts in sync with the actual code is difficult. The only artifact that survives a project is the executable, which is created from code, not comments. This is one of the reasons for writing self-documenting code. (Well, annotations, XDoclet, and so on, make that somewhat less true. You know what I mean.)

There are little things we can do throughout our code to make our code read as much like our intent as possible and make extraneous comments just that: extraneous.

Document why, not what

Over-commenting wastes everybody's time. Time is wasted in writing a comment, reading it, keeping that comment in sync with the code, and, most importantly, a *lot* of time is wasted when a comment is not accurate.

Ever seen this?

```
a += 1; // increment a
```

This is the most useless comment in the world.

Firstly, it's really obvious we're incrementing something, regardless of what that something is. If the person reading our code doesn't know what `+=` is, then we have more serious problems than them not knowing that we're incrementing, say, an array index.

Secondly, if `a` is an array index, we should probably use either a more common array index or make it obvious that it's an array index. Using `i` and `j` is common for array indices, while `idx` or `index` is less common. It may make sense to be very explicit in variable naming under some circumstances. Generally, it's nice to avoid names such as `indexOfOuterArrayOfFoobars`. However, with a large loop body it might make sense to use something such as `num` or `currentIndex`, depending on the circumstances.

With Java 1.5 and its support for collection iteration, it's often possible to do away with the index altogether, but not always.

Make your code read like the problem

Buzzphrases like **Domain Specific Languages** (**DSLs**) and **Fluent Interfaces** are often heard when discussing how to make our code look like our problem. We don't necessarily hear about them as much in the Java world because other languages support their creation in more "literate" ways. The recent interest in Ruby, Groovy, Scala, and other dynamic languages have brought the concept back into the mainstream.

A DSL, in essence, is a computer language targeted at a very specific problem. Java is an example of a general-purpose language. YACC and regular expressions are examples of DSLs that are targeted at creating parsers and recognizing strings of interest respectively.

DSLs may be **external**, where the implementing language processes the DSL appropriately, as well as **internal**, where the DSL is written in the implementing language itself. An internal DSL can also be thought of as an API or library, but one that reads more like a "little language".

Fluent interfaces are slightly more difficult to define, but can be thought of as an internal DSL that "flows" when read aloud. This is a very informal definition, but will work for our purposes.

Java can actually be downright hostile to some common DSL and fluent techniques for various reasons, including the expectations of the JavaBean specification. However, it's still possible to use some of the techniques to good effect. One typical practice of fluent API techniques is simply returning the object instance in object methods. For example, following the JavaBean specification, an object will have a `setter` for the object's properties. For example, a `User` class might include the following:

```
public class User {
    private String fname;
    private String lname;
    public void setFname(String fname) { this.fname = fname; }
    public void setLname(String lname) { this.lname = lname; }
}
```

Using the class is as simple as we'd expect it to be:

```
User u = new User();
u.setFname("James");
u.setLname("Gosling");
```

Naturally, we might also supply a constructor that accepts the same parameters. However, it's easy to think of a class that has many properties making a full constructor impractical. It also seems like the code is a bit wordy, but we're used to this in Java. Another way of creating the same functionality is to include setter methods that return the current instance. If we want to maintain JavaBean compatibility, and there are reasons to do so, we would still need to include normal setters, but can still include "fluent" setters as shown here:

```
public User fname(String fname) {
    this.fname = fname;
    return this;
}
public User lname(String lname) {
    this.lname = lname;
    return this;
}
```

This creates (what some people believe is) more readable code. It's certainly shorter:

```
User u = new User().fname("James").lname("Gosling");
```

There is one potential "gotcha" with this technique. Moving initialization into methods has the potential to create an object in an invalid state. Depending on the object this may not always be a usable solution for object initialization.

Users of Hibernate will recognize the "fluent" style, where method chaining is used to create criteria. Joshua Flanagan wrote a fluent regular expression interface, turning regular expressions (already a domain-specific language) into a series of chained method calls:

```
Regex socialSecurityNumberCheck =
    new Regex(Pattern.With.AtBeginning
                .Digit.Repeat.Exactly(3)
                .Literal("-").Repeat.Optional
                .Digit.Repeat.Exactly(2)
                .Literal("-").Repeat.Optional
                .Digit.Repeat.Exactly(4)
                .AtEnd);
```

Whether or not this particular usage is an improvement is debatable, but it's certainly easier to read for the non-regex folks.

Ultimately, the use of fluent interfaces can increase readability (by quite a bit in most cases), may introduce some extra work (or completely duplicate work, like in the case of setters, but code generation and/or IDE support can help mitigate that), and may occasionally be more verbose (but with the benefit of enhanced clarity and IDE completion support).

Personally, I'm of the opinion that regular expressions are so incredibly important that it's worth learning them in their native form, as they can be used in many environments, including the IDEs so loved by Java developers. Large expressions can be broken down into components and created by concatenating strings. But the point here is more about the style of fluent programming, rather than this specific example.

Contract-oriented programming

Aspect-oriented programming (**AOP**) is a way of encapsulating cross-cutting functionality outside of the mainline code. That's a mouthful, but essentially it means is that we can remove common code that is found across our application and consolidate it in one place. The canonical examples are logging and transactions, but AOP can be used in other ways as well.

Design by Contract (**DbC**) is a software methodology that states our interfaces should define and enforce precise specifications regarding operation.

"Design by Contract" is a registered trademark of Interactive Software Engineering Inc. Other terms include **Programming by Contract** (**PbC**), or my personal favorite, **Contract Oriented Programming** (**COP**), which is how I'll refer to it from now on. I have a lot of respect for Eiffel and its creator, but this type of trademarking bothers me. Maybe I'll trademark "Singleton"?!

How does COP help create self-documenting code? Consider the following portion of a stack implementation:

```
public void push(final Object o) {
    stack.add(o);
}
```

This seems simple enough. The information available to us is that we can push an object, whatever pushing means.

What happens if we attempt to push a null? Let's assume that for this implementation, we don't want to allow pushing a null onto the stack.

```
/**
 * Pushes non-null objects on to stack.
 */
public void push(final Object o) {
    if (o == null) return;
    stack.add(o);
}
```

Once again, this is simple enough. We'll add the comment to the Javadocs stating that null objects will not be pushed (and that the call will fail/return silently). This will become the "contract" of the push method—captured in code and documented in Javadocs.

The contract is specified twice—once in the code (the ultimate arbiter) and again in the documentation. However, the user of the class does not have proof that the underlying implementation actually honors that contract. There's no guarantee that if we pass in a null, it will return silently without pushing anything.

The implied contract can change. We might decide to allow pushing nulls. We might throw an `IllegalArgumentException` or a `NullPointerException` on a null argument. We're not required to add a `throws` clause to the method declaration when throwing runtime exceptions. This means further information may be lost in both the code and the documentation.

As hinted, Eiffel has language-level support for COP with the `require/do/ensure/end` construct. It goes beyond the simple null check in the above code. It actively encourages detailed pre- and post-condition contracts. An implementation's `push()` method might check the remaining stack capacity before pushing. It might throw exceptions for specific conditions. In pseudo-Eiffel, we'd represent the `push()` method in the following way:

```
push (o: Object)
    require
        o /= null
    do
        -- push
    end
```

A stack also has an implied contract. We assume (sometimes naively) that once we call the `push` method, the stack will contain whatever we pushed. The size of the stack will have increased by one, or whatever other conditions our stack implementation requires.

One aim of COP is to formalize the nature of contracts. Languages such as Eiffel have one solution to that problem, and having it built-in at the language level provides a consistent means of expressing contracts.

Java, of course, doesn't have built-in contracts. However, it does contain a mechanism that can be used to get some of the benefits for a conceptually-simple price. The mechanism is not as complete, or as integrated, as Eiffel's version. However, it removes contract enforcement from the mainline code, and provides a way for both sides of the software to specify, accept, and document the contracts themselves.

Removing the contract information from the mainline code keeps the implementation clean and makes the implementation code easier to understand. Having programmatic access to the contract means that the contract could be documented automatically rather than having to maintain a disconnected chunk of Javadoc.

SpringContracts

SpringContracts is a beta-level Java COP implementation based on Spring's AOP facilities, using annotations to state pre- and post-contract conditions. It formalizes the nature of a contract, which can ease development.

Let's consider our `VowelDecider` that was developed through TDD. We can also use COP to express its contract (particularly the entry condition). This is a method that doesn't alter state, so post conditions don't apply here.

Our implementation of `VowelDecider` ended up looking (more or less) like this:

```
public boolean decide(final Object o) throws Exception {
    if ((o == null) || (!(o instanceof String))) {
        throw new IllegalArgumentException(
            "Argument must be a non-null String.");
    }
    String s = (String) o;
    return s.matches(".*[aeiouy]+.*");
}
```

Once we remove the original contract enforcement code, which was mixed with the mainline code, our SpringContracts `@Precondition` annotation looks like the following:

```
@Precondition(condition="arg1 != null && arg1.class.name == 'java.
lang.String'",
              message="Argument must be a non-null String")
public boolean decide(Object o) throws Exception {
    String s = (String) o;
    return s.matches(".*[aeiouy]+.*");
}
```

The pre-condition is that the argument must not be null and must be (precisely) a string. (Because of SpringContracts' Expression Language, we can't just say `instanceof String` in case we want to allow string subclasses.)

We can unit-test this class in the same way we tested the TDD version. In fact, we can copy the tests directly. Running them should trigger test failures on the null and non-string argument tests, as we originally expected an `IllegalArgumentException`. We'll now get a contract violation exception from SpringContracts.

One difference here is that we need to initialize the Spring context in our test. One way to do this is with JUnit's `@BeforeClass` annotation, along with a method that loads the Spring configuration file from the classpath and instantiates the decider as a Spring bean. Our class setup now looks like this:

```
@BeforeClass public static void setup() {
    appContext = new ClassPathXmlApplicationContext(
                  "/com/packt/s2wad/applicationContext.xml");
    decider = (VowelDecider)
                  appContext.getBean("vowelDecider");
}
```

We also need to configure SpringContracts in our Spring configuration file. Those unfamiliar with Spring's (or AspectJ's) AOP will be a bit confused. However, in the end, it's reasonably straightforward, with a potential "gotcha" regarding how Spring does proxying.

```
<aop:aspectj-autoproxy proxy-target-class="true"/>

<aop:config>
  <aop:aspect ref="contractValidationAspect">
    <aop:pointcut id="contractValidatingMethods"
          expression="execution(*
            com.packt.s2wad.example.CopVowelDecider.*(..))"/>
    <aop:around pointcut-ref="contractValidatingMethods"
                method="validateMethodCall"/>
  </aop:aspect>
</aop:config>

<bean id="contractValidationAspect"
class="org.springcontracts.dbc.interceptor.
ContractValidationInterceptor"/>

<bean id="vowelDecider"
      class="com.packt.s2wad.example.CopVowelDecider" />
```

If most of this seems like a mystery, that's fine. The SpringContracts documentation goes into it a bit more and the Spring documentation contains a wealth of information regarding how AOP works in Spring. The main difference between this and the simplest AOP setup is that our autoproxy target must be a class, which requires CGLib. This could also potentially affect operation.

The only other modification is to change the exception we're expecting to SpringContract's `ContractViolationCollectionException`, and our test starts passing. These pre- and post-condition annotations use the `@Documented` meta-annotation, so the SpringContracts COP annotations will appear in the Javadocs. It would also be possible to use various other means to extract and document contract information.

Getting into details

This mechanism, or its implementation, may not be a good fit for every situation. Runtime performance is a potential issue. As it's just some Spring magic, it can be turned off by a simple configuration change. However, if we do, we'll lose the value of the on-all-the-time contract management.

On the other hand, under certain circumstances, it may be enough to say that once the contracts are consistently honored under all of the test conditions, the system is correct enough to run without them. This view holds the contracts more as an acceptance test, rather than as run-time checking. Indeed, there is an overlap between COP and unit testing as the way to keep code honest. As unit tests aren't run all the time, it may be reasonable to use COP as a temporary runtime unit test or acceptance test.

Javadocs

We'll cover only a few things regarding Javadocs. I'm sure we're all very familiar with them, but there are a few tips that might be helpful occasionally.

Always write Javadocs!

The first bit of advice is to always write Javadocs, except when they're not really needed. Getters and setters that have no additional functionality really don't need them. However, as soon as a getter or setter does more than just get or set its value, it may deserve documentation. Even minor functionality that's trivial to understand when looking at the code may deserve Javadocs. We may not have access to the source or we may only want to look at the API documentation.

The first sentence

The first sentence of a Javadoc comment is used as the summary documentation. It isn't necessary to encapsulate every nuance of the member being documented in the first sentence, but it's important to give a very clear and concise overview of the member. By default, the first sentence is everything up to the first "." (period). Some tools will complain if the first sentence is not properly terminated.

The proper way to describe the grammar of the first sentence is something along the lines of: "use a verb phrase in the third person declarative form." What does that mean in real life?

```
/**
 * Builds and returns the current list of ingredients.
 *
 * @return List of ingredients.
 */
public List<Ingredient> buildIngredientList() { ... }
```

In the case of methods, the Javadoc summary should answer the question: "What does this member do?" One answer could be: "`buildIngredientList()` builds and returns the list of ingredients." This is opposed to saying something such as "Build and return list of ingredients", which doesn't work as an answer to the question. This is the "verb phrase" part.

The "third person declarative" part (informally) means that we answer the question as directly as possible. Sentence fragments are okay here. Additional exposition harms clarity. For example, we probably would not want to write the following:

```
/**
 * This method builds and returns the current list
 * of ingredients.
 */
```

That's not a direct answer to the question "What does `buildIngredientList()` do?". Therefore, this probably is not the best style of documentation.

This method is simple enough. Therefore, we may not need the `@return` Javadoc tag. What it returns is already specified in the first sentence. However, some tools may complain about missing Javadoc tags.

For variables, a simple descriptive sentence such as the following is usually fine:

```
/** Pre-built recipe summary. */
private String summary;
```

Is it okay to have member variables without Javadocs? The answer is yes, if the purpose is self-evident from the name. However, if our build process includes a tool that enforces Javadoc requirements, we'll either get irritating warnings or we'll need to specify what to exclude from checking.

If there aren't any Javadoc requirements, then all bets are off. However, bear in mind that Javadocs are also used by the IDE to provide information in various forms such as roll-over Javadoc pop-ups. It often boils down to whether or not we are able to come up with a good variable or method name. If we can, then the benefits of essentially repeating the same information in a Javadoc comment are very low and are probably not worth it.

Add information beyond the API name

In our `buildIngredientList()` example seen earlier, our first sentence really doesn't tell us much more than the name of the method does. This is good because it means that our method name (the API name) is probably correct and sufficient. However, let's assume that the method actually does something interesting during the construction of the ingredient list. That information should then be added to the Javadocs.

The information does not (necessarily) belong in the summary. Therefore, we can simply continue with another sentence (this is a bit contrived, since it could be merged into the first sentence quite easily).

```
/**
 * Builds and returns the current list of ingredients.
 * Initializes ingredient information if necessary.
 */
```

The summary information will consist of only the first sentence, but both sentences will be in the complete Javadocs. Note that in this case, it might make more sense to use a single sentence similar to the following:

```
/**
 * Builds and returns the current list of ingredients,
 * initializing ingredient info when necessary.
 */
```

The trick is to consistently make good decisions regarding what the most essential information is, and communicating it cleanly and concisely.

Write for multiple formats

Javadocs should be written with the thought that they might be read in several formats. Some common ways of viewing Javadocs include embedded in source code, using an IDE or an IDE popup/hover, the results of a `grep`, and so on. They may also be viewed as HTML, such as after they've been processed with the Javadoc tool. Javadoc comments may even be included in a wiki, through some sort of snippet mechanism or by including it in various forms of documentation.

In our example above, we have two sentences in a row. Let's say that we need to highlight the fact that the ingredient information will be initialized if necessary. Our first attempt just adds a bold `Note` to the second sentence.

```
/**
 * Builds and returns the current list of ingredients.
 *
 * <b>Note:</b> Initializes ingredient information if
 * necessary.
 */
```

The word `Note:` will stand out in the HTML output, but will appear connected to the opening sentence. Javadoc doesn't honor text-based line breaks. We must use HTML to format our Javadocs. Creating separate paragraphs requires the use of paragraph tags.

By formatting our Javadoc as indented HTML, we can create documentation that reads reasonably well in both text and HTML formats. Additionally, with judicious use of HTML tags, we can use doclets that create printable PDF documentation (or other printable formats).

```
/**
 * Builds and returns the current list of ingredients.
 *
 * <p>
 *   <b>Note:</b> Initializes ingredient information
 *   if necessary.
 * </p>
 */
```

As usual, this example is a bit contrived. We'd probably just want to put it all in the first sentence.

Generating targeted Javadocs

One reason people give for not writing Javadocs for a particular method is that the method isn't necessarily designed to be used by others, or that exposing even the documentation isn't a good idea. The Javadoc tool gives us a few ways to restrict what documentation is generated.

Visibility

The most obvious way to restrict what documentation is generated is based on visibility. By default, Javadoc will generate documentation for all of the public and protected classes. By using the `-public`, `-protected`, `-package`, and `-private` flags, we can control the level of visibility for which documentation will be generated. Note that we need to specify only one flag—any member with equal or greater visibility will have documentation generated for it.

For example, running Javadoc with the `-public` flag will generate documentation for only public members, creating Javadocs suitable for the users of an API. Running with the `-private` flag will generate documentation for all of the members, making the documentation suitable for the developers of the same API.

The -exclude argument

The `-exclude` argument allows us to supply package names that will be excluded from Javadoc generation. For example, if we want to create documentation that specifically excludes an "internal-use only" package (security through obscurity?), we can use the `-exclude` argument to provide a ":" (colon) separated list of packages for which no Javadocs will be generated.

```
javadoc -exclude com.packt.s2wad.internal {...}
```

No classes in the `com.packt.s2wad.internal` package will be documented.

The -use argument

The `-use` argument will generate a "Use" page for each class `C` being documented. The "Use" page will contain an entry for each package, class, method, and fields "using" class C. Uses include subclasses, methods that use the class as a parameter, and methods that return an instance of the class.

This page may not be as useful as a graphical representation of the class interdependencies, but it's an option that's available out of the box. Creating various diagrams is possible with add-on Javadoc doclets such as yDoc or UmlGraph, as well as with non-Javadoc-oriented tools.

Creating new Javadoc tags with the -tag argument

One Javadoc capability that's trivial to take advantage of is creating our own Javadoc tags. These tags are similar to the `@param` and `@return` tags that Javadoc recognizes by default. This capability may or may not fit into your organization or coding style, but it's simple enough to use that it's worth an introduction. Another potential issue is that our IDE or build process probably won't be capable of validating the information contained in the tag, unlike the default tags. For example, Eclipse can flag a warning if a method's parameters don't have corresponding `@param` tags.

We could document an action that puts a variable into session by creating a `@session` Javadoc tag. By telling the Javadoc tool to pay attention to that tag, we can create output for it just like the built-in `@param` tag. An action's Javadocs might look like this:

```
/**
 * Retrieves recipe, creating session parameters as needed.
 *
 * @session CONSTANTS.LAST_RECIPE The last recipe accessed.
 *
 * @returns SUCCESS or ERROR if retrieval fails.
 */
```

We instruct the `javadoc` tool to pay attention to our new tag by giving it `-tag` arguments. The easiest method is to just add `-tag session` to the `javadoc` command. How this is done depends on your build environment. It can be done using an Ant script, via an IDE `javadoc` frontend, and so on.

Adding the `-tag session` argument instructs `javadoc` to create output for `@session` tags similar to `@param` and `@returns` tags. The generated output will appear after the standard tags. If we want to change the order in the HTML we must supply a complete list of `-tag arguments` including the built-in tags as well. Each tag's output is generated in the order specified on the command line. If we wanted to see the `@session` documentation before the `@return` tag's documentation, then we'd specify both documentations on the command line as follows:

```
-tag session -tag return
```

We can also specify the header for a custom Javadoc tag. To set a header for the session tag we use colon-separated values (ignoring the `a` for now):

```
-tag session:a:"Session attributes accessed:"
```

Note that the header label for built-in tags can be modified in the same way, provided we have a good reason to do so. However, we will probably never have a good reason to do so.

The "a" we snuck in there determines where in the source code we can use the tag, with "a" meaning anywhere we want. A complete list of determinants is found in the `javadoc` tool documentation, but includes "t" for types (classes and interfaces), "m" for methods, "f" for fields, and so on. These may be combined, so a tag to identify injected entities for both types and fields could be specified as follows:

```
-tag injected:tf:"Injected entities:"
```

If we now try to use our new `@injected` tag in a method declaration, the Javadoc tool will signal an error, as it's been specified as being valid only for types and fields.

Note that this functionality of `javadoc` may overlap some use of annotations. For example, assume we're using an interceptor that loads and saves an annotated variable from and to the JEE session. It would make more sense to use a doclet that included this information from the annotation, rather than writing (and worse, maintaining) the Javadoc manually—the more we can do automatically, the better.

Never write Javadocs!

I know what you're thinking—but as soon as we write Javadocs, we've entered into an implicit contract to always keep them up-to-date, in perpetuity, over the life of the program. If we can't do that, it may be better not to write any. Remember, wrong documentation is worse than having no documentation.

There are many cases where it makes sense to write detailed Javadocs, describing a complicated or non-obvious algorithm being chief among them. However, it's arguable whether such documentation belongs in the application's non-code documentation or in a wiki.

Never write inline Java comments!

If we find ourselves writing chunks of comments inside methods to explain each section of a method, we might be better off refactoring the chunk into its own method. Of course, some code lends itself to this more readily than others, which might be impractical for a variety of reasons. There's always a trade-off. However, there is always a cost associated with non-code documentation, as it is ultimately maintained separately from the code itself.

Using UML

UML (**Unified Markup Language**) can handle a wide range of documentation duties, much more than will be covered here. Even UML-like diagrams can be of great assistance in many aspects of documentation. It's not necessary to follow all of the UML notation or diagrams completely, purchase an expensive enterprise UML tool, and so on. However, a basic understanding of UML is very handy when documenting our own application, or reading the documentation of other projects that use UML.

There are many ways to integrate UML into the development and documentation process. It might be used to generate source code, it can be included in both developer and end-user documentation (where appropriate), and so on. Two of the more common UML diagrams related directly to Java code are the **package** and **class** diagrams, which most of us are already familiar with.

Package diagrams

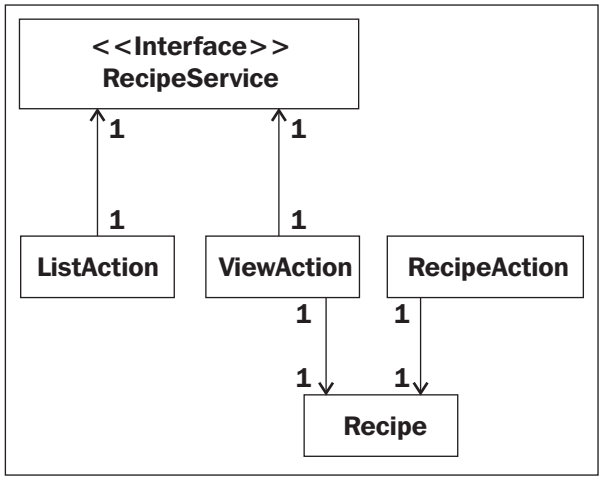
Package diagrams are similar to class diagrams, but provide a very high-level overview of an application's package and package dependencies. We saw a portion of a package diagram back in Chapter 3 when we looked at a portion of the XWork 2 packages. There, we only looked at the XWork interfaces and did not highlight package coupling.

Class diagrams

One of the most useful and commonplace UML diagrams is probably the class diagram. The class diagram is used to show the relationship between different classes in a system. Class diagrams can be created at various levels of detail to provide very high-level overviews or extremely detailed information, including all of a class's properties.

For example, a portion of our `com.packt.s2wad.recipe` package's class diagram can be represented by the following UML class diagram. Note that this is an incomplete diagram and doesn't show our implementation of `RecipeService`. It's also fairly high-level, and doesn't show any class properties or methods.

However, it's still useful because it's obvious that we have two classes that use a `RecipeService`, and two classes that have a relationship to the `Recipe` class. This type of high-level overview is particularly important when first learning an application, identifying high-level class-coupling issues, and so on.



A class diagram for an entire application can be a bit overwhelming, and isn't always useful due to the sheer amount of information. Restricting what is visible at both the class and package level can be very useful allowing usable documentation to be generated.

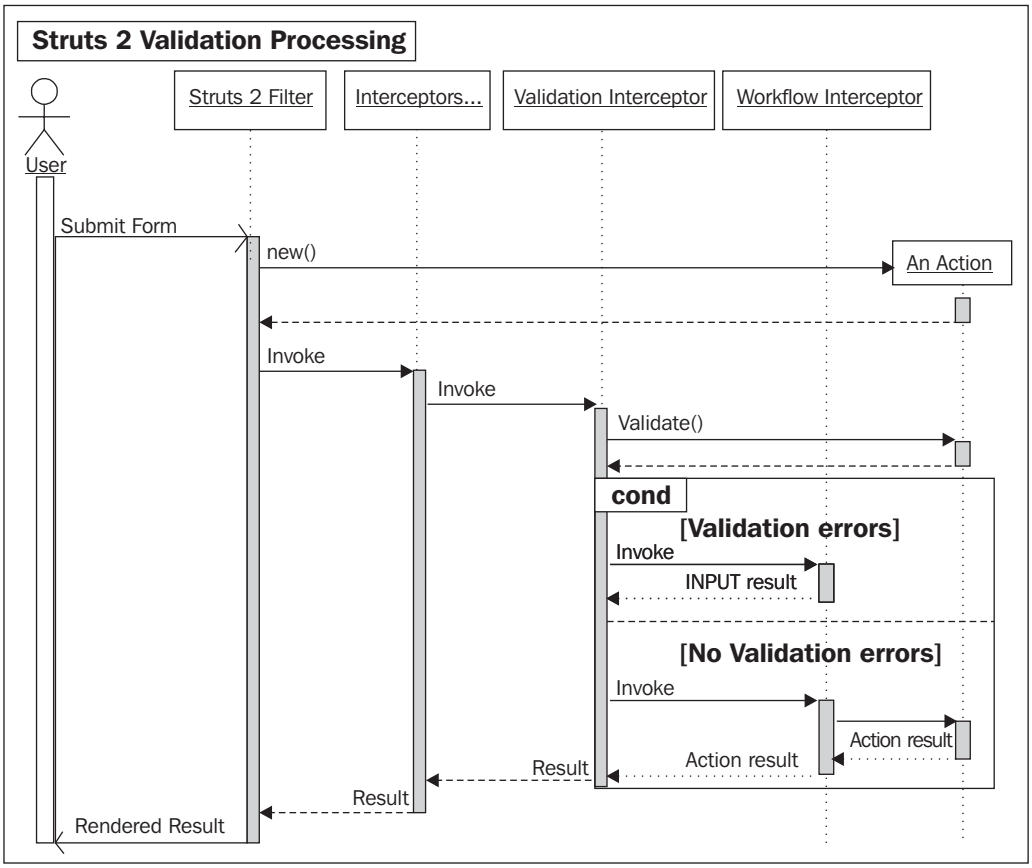
The previous image was generated with ArgoUML, an open source UML tool from application source. It was then edited by hand to improve the layout, to remove class members and operations, and so on.

Java source can also be generated from UML models. Several IDEs and modeling tools supply this functionality. The direction of generation, whether generating UML from source or source from UML, is largely a matter of preference, culture, and how well the available tools work for our style (or the style of our client).

Sequence diagrams

Another popular and useful UML diagram, which is relatively lightweight, is the sequence diagram. These diagrams are used to document interactions between entities. These entities are normally classes. However, it's not unreasonable to extend the sequence diagram metaphor beyond the official definition when necessary, such as adding user interactions, browser functionality, and so on.

As a quick example, we can take a look at the typical Struts 2 form validation processing sequence, compressing the non-related interceptors into a single entity:



Here, we're representing the user action of submitting a form in the sequence diagram. This (in simplified form) causes the Struts 2 filter to instantiate the appropriate action class, indicated by the **new()** message. The rest is (I'd imagine) largely self-explanatory.

Sequence diagrams are often easier to comprehend than plain text and can be more convenient than the code itself, as they aggregate as many classes as needed. As usual, they can suffer from decay, as the code continues to be modified and the diagrams aren't being generated automatically or being actively maintained.

This diagram was created using the Quick Sequence Diagram Editor, which creates an exportable diagram using simple text-based input (I was tempted to call it a DSL, but managed to stop myself). The input for this diagram is relatively short.

```
#![Struts 2 Validation Processing]
user:Actor "User"
s2f: "Struts 2 Filter"
etc: "Interceptors..."
val: "Validation Interceptor"
workflow: "Workflow Interceptor"
/action: "An Action"
user:s2f.Submit form
s2f:action.new()
s2f:Result=etc.Invoke
etc:Result=val.Invoke
val:action.validate()
[c:cond Validation errors]
  val:INPUT result=workflow.Invoke
--[No validation errors]
  val:Action result=workflow.Invoke
  workflow:Action result=action.Invoke
[/c]
s2f:user.Rendered Result
```

Personally, I think the send/receive messages are defined backwards. It's pretty easy to create our own DSL (for example, in Ruby) that corrects this error. It's also relatively straightforward to create a log format that could be parsed to create diagrams from actual code. Therefore, running unit tests could also be a part of the documentation process.

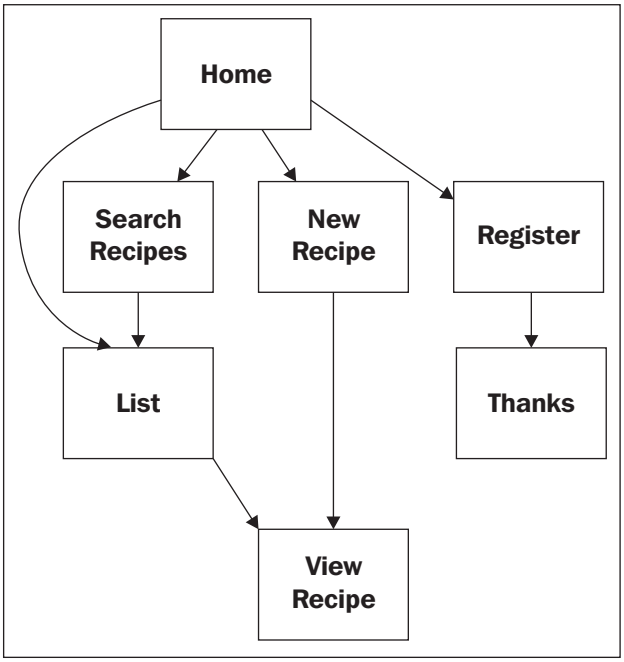
Documenting web applications

Documenting an entire web application can be surprisingly tricky because of the many different layers involved. Some web application frameworks support automatic documentation generation better than others. It's preferable to have fewer disparate parts. For example, Lisp, Smalltalk, and some Ruby frameworks are little more than internal DSLs that can be trivially redefined to produce documentation from the actual application code.

In general, Java frameworks are more difficult to limit to a single layer. Instead, we are confronted with HTML, JSP, JavaScript, Java, the framework itself, its configuration methodologies (XML, annotations, scripting languages, etc.), the service layers, business logic, persistence layers, and so on—feeling sleepy? Complete documentation generally means aggregating information from many disparate sources and presenting them in a way that is meaningful to the intended audience.

High-level overviews

The site map is obviously a reasonable overview of a web application. A site map may look like a simple hierarchy chart, showing a simple view of a site's pages without showing all of the possible links between pages, how a page is implemented, and so on.



This diagram was created by hand and shows only the basic outline of the application flow. It represents minor maintenance overhead since it would need to be updated when there are any changes to the application.

Documenting JSPs

There doesn't seem to be any general-purpose JSP documentation methodology. It's relatively trivial to create comments inside a JSP page using JSP comments or a regular Javadoc comment inside a scriptlet. Pulling these comments out is then a matter of some simple parsing. This may be done by using one of our favorite tools, regular expressions, or using more HTML-specific parsing and subsequent massaging.

Where it gets tricky is when we want to start generating documentation that includes elements such as JSP pages, which may be included using many different mechanisms—static includes, `<jsp:include.../>` tags, Tiles, SiteMesh, inserted via Ajax, and so on. Similarly, generating connections between pages is fraught with custom cases. We might use general-purpose HTML links, Struts 2 link tags, attach a link to a page element with JavaScript, ad infinitum/nauseum.

When we throw in the (perhaps perverse) ability to generate HTML using Java, we have a situation where creating a perfectly general-purpose tool is a major undertaking. However, we can fairly easily create a reasonable set of documentation that is specific to our framework by parsing configuration files (or scanning a classpath for annotations), understanding how we're linking the server-side to our presentation views, and performing (at least limited) HTML/JSP parsing to pull out presentation-side dependencies, links, and anything that we want documented.

Documenting JavaScript

If only there was a tool such as Javadoc for JavaScript. Fortunately, I was not the only one that had that desire! The JsDoc Toolkit provides Javadoc-like functionality for JavaScript, with additional features to help handle the dynamic nature of JavaScript code. Because of the dynamic nature, we (as developers) must remain diligent in both in how we write our JavaScript and how we document it.

Fortunately, the JsDoc Toolkit is good at recognizing current JavaScript programming paradigms (within reason), and when it can't, provides Javadoc-like tags we can use to give it hints.

For example, consider our JavaScript `Recipe` module where we create several private functions intended for use only by the module, and return a map of functions for use on the webpage. The returned map itself contains a map of validation functions. Ideally, we'd like to be able to document all of the different components.

Because of the dynamic nature of JavaScript, it's more difficult for tools to figure out the context things should belong to. Java is much simpler in this regard (which is both a blessing and a curse), so we need to give JsDoc hints to help it understand our code's layout and purpose.

A high-level flyby of the `Recipe` module shows a layout similar to the following:

```
var Recipe = function () {
    var ingredientLabel;
    var ingredientCount;
    // ...
    function trim(s) {
        return s.replace(/^\s+|\s+$/g, "");
```

```
        }
        function msgParams(msg, params) {
            // ...
        }
        return {
            loadMessages: function (msgMap) {
                // ...
            },
            prepare: function (label, count) {
                // ...
            },
            pageValidators: {
                validateIngredientNameRequired: function (form) {
                    // ...
                },
                // ...
            }
        };
    }();
```

We see several documentable elements: the `Recipe` module itself, private variables, private functions, and the return map which contains both functions and a map of validation functions. JsDoc accepts a number of Javadoc-like document annotations that allow us to control how it decides to document the JavaScript elements.

The JavaScript module pattern, exemplified by an immediately-executed function, is understood by JsDoc through the use of the `@namespace` annotation.

```
/**
 * @namespace
 * Recipe module.
 */
var Recipe = function () {
    // ...
}();
```

(Yes, this is one of those instances where eliminating the comment itself would be perfectly acceptable!) We'll look at the JsDoc output momentarily after covering a few more high-level JsDoc annotations.

We can mark private functions with the `@private` annotation as shown next:

```
/**
 * @private
 * Trims leading/trailing space.
 */
function trim(s) {
    return s.replace(/^\s+|\s+$/g, "");
}
```

(Again the JsDoc comment could be replaced by creating a better function name. Really—I named it poorly so that I could bring up the documentation issues.)

It gets interesting when we look at the map returned by the `Recipe` module:

```
return /** @lends Recipe */ {
    /**
     * Loads message map.
     *
     * <p>
     *   This is generally used to pass in text resources
     *   retrieved via <s:text.../> or <s:property
     *   value="getText(...)"/> tags on a JSP page in lieu
     *   of a normalized way for JS to get Java I18N resources
     * </p>
     */
    loadMessages: function (msgMap) {
        _msgMap = msgMap;
    },
    // ...
```

The `@lends` annotation indicates that the functions returned by the `Recipe` module belong to the `Recipe` module. Without the `@lends` annotation, JsDoc doesn't know how to interpret the JavaScript in the way we probably intend the JavaScript to be used, so we provide a little prodding.

The `loadMessages()` function itself is documented as we would document a Java method, including the use of embedded HTML.

The other interesting bit is the map of validation functions. Once again, we apply the `@namespace` annotation, creating a separate set of documentation for the validation functions, as they're used by our validation template hack and not directly by our page code.

```
/**
 * @namespace
 * Client-side page validators used by our template hack.
 * ...
 */
pageValidators: {
    /**
     * Insures each ingredient with a quantity
     * also has a name.
     *
     * @param {Form object} form
     * @type boolean
     */
    validateIngredientNameRequired: function (form) {
        // ...
```

Note also that we can annotate the type of our JavaScript parameters inside curly brackets. Obviously, JavaScript doesn't have typed parameters. We need to tell it what the function is expecting. The `@type` annotation is used to document what the function is expected to return. It gets a little trickier if the function returns different types based on arbitrary criteria. However, we never do that because it's hard to maintain—right?

Overloading JavaScript return values is typical because JavaScript has a wider range of truthiness and falseness than Java. This is okay, even though it gives Java programmers conniptions. That's okay too, and can be mildly entertaining to boot.

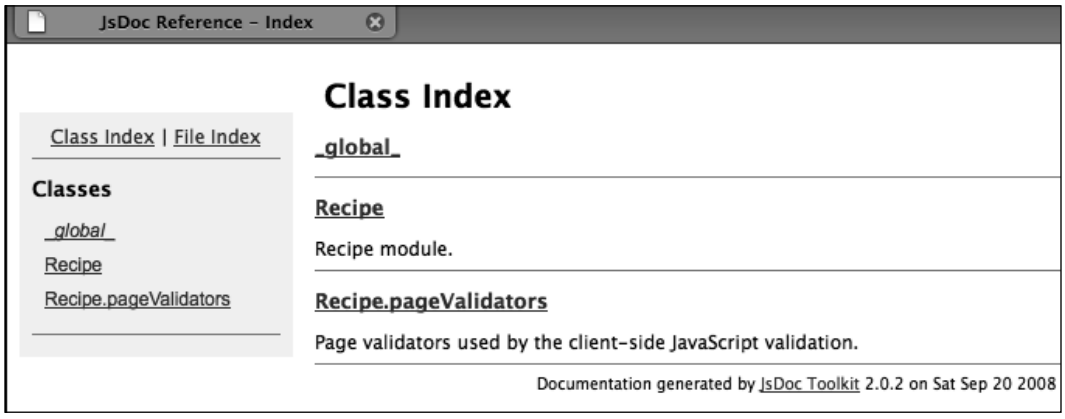
JsDoc has the typical plethora of command-line options, and requires the specification of the application itself (written in JavaScript, and run using Rhino) and the templates defining the output format. An alias to run JsDoc might look like the following, assuming the JsDoc installation is being pointed at by the `${JSDOC}` shell variable:

```
alias jsdoc='java -jar ${JSDOC}/jsrun.jar
             ${JSDOC}/app/run.js -t=${JSDOC}/templates/jsdoc'
```

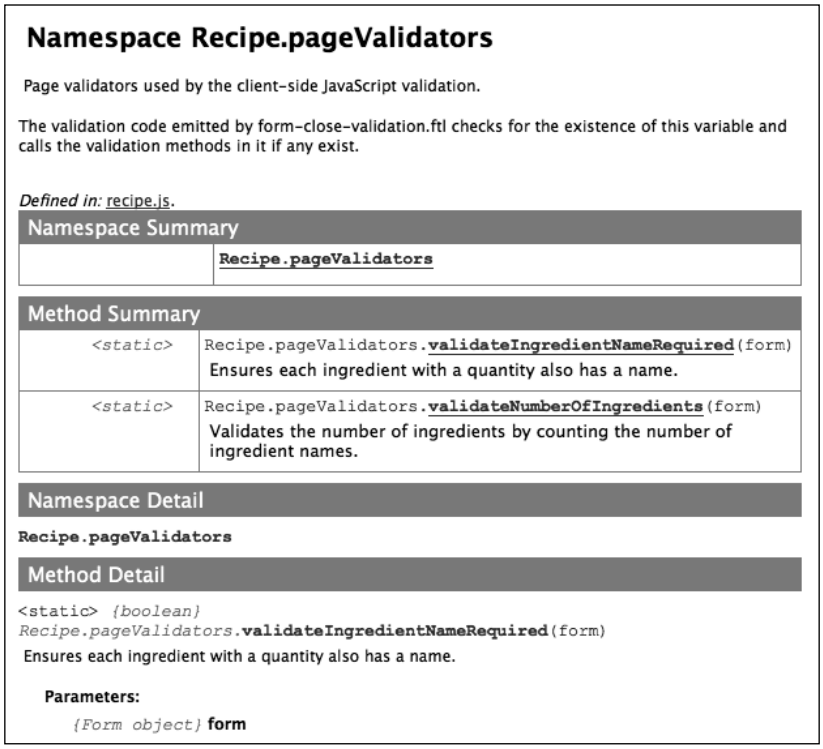
The command line to document our `Recipe` module (including private functions using the `-p` options) and to write the output to the `jsdoc-out` folder, will now look like the following:

```
jsdoc -p -d=jsdoc-out recipe.js
```

The homepage looks similar to a typical JavaDoc page, but more JavaScript-like:



A portion of the `Recipe` module's validators, marked by a `@namespace` annotation inside the `@lends` annotation of the return map, looks like the one shown in the next image (the left-side navigation has been removed):



Namespace Recipe.pageValidators

Page validators used by the client-side JavaScript validation.

The validation code emitted by form-close-validation.ftl checks for the existence of this variable and calls the validation methods in it if any exist.

Defined in: recipe.js.

Namespace Summary

	Recipe.pageValidators

Method Summary

<static>	Recipe.pageValidators.**validateIngredientNameRequired**(form) Ensures each ingredient with a quantity also has a name.
<static>	Recipe.pageValidators.**validateNumberOfIngredients**(form) Validates the number of ingredients by counting the number of ingredient names.

Namespace Detail

Recipe.pageValidators

Method Detail

<static> *{boolean}*
*Recipe.pageValidators.***validateIngredientNameRequired**(form)

Ensures each ingredient with a quantity also has a name.

Parameters:

{Form object} **form**

We can get a pretty decent and accurate JavaScript documentation using JsDoc, with only a minimal amount of prodding to help with the dynamic aspects of JavaScript, which is difficult to figure out automatically.

Documenting interaction

Documenting interaction can be surprisingly complicated, particularly in today's highly-interactive Web 2.0 applications. There are many different levels of interactivity taking place, and the implementation may live in several different layers, from the JavaScript browser to HTML generated deep within a server-side framework.

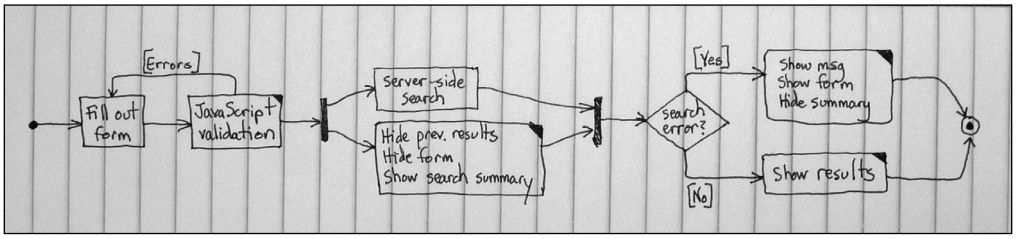
UML sequence diagrams may be able to capture much of that interactivity, but fall somewhat short when there are activities happening in parallel. AJAX, in particular, ends up being a largely concurrent activity. We might send the AJAX request, and then do various things on the browser in anticipation of the result.

More UML and the power of scribbling

The UML activity diagram is able to capture this kind of interactivity reasonably well, as it allows a single process to be split into multiple streams and then joined up again later. As we look at a simple activity diagram, we'll also take a quick look at scribbling, paper, whiteboards, and the humble digital camera.

Don't spend so much time making pretty pictures

One of the hallmarks of lightweight, agile development is that we don't spend all of our time creating the World's Most Perfect Diagram™. Instead, we create just enough documentation to get our points across. One result of this is that we might not use a $1,000 diagramming package to create all of our diagrams. Believe it or not, sometimes just taking a picture of a sketched diagram from paper or a whiteboard is more than adequate to convey our intent, and is usually much quicker than creating a perfectly-rendered software-driven diagram.



Yes, the image above is a digital camera picture of a piece of notebook paper with a rough activity diagram. The black bars here are used to indicate a small section of parallel functionality, a server-side search and some activity on the browser. I've also created an informal means of indicating browser programming, indicated by the black triangles. In this case, it might not even be worth sketching out. However, for moderately more complicated usage cases, particularly when there is a lot of both server- and client-side activity, a high-level overview is often worth the minimal effort.

The same digital camera technique is also very helpful in meetings where various documentation might be captured on a whiteboard. The resulting images can be posted to a company wiki, used in informal specifications, and so on.

User documentation

Development would be substantially easier if we didn't have to worry about those pesky users, always wanting features, asking questions, and having problems using the applications we've developed. Tragically, users also drive our paycheck. Therefore, at some point, it can be beneficial to acknowledge their presence and throw them the occasional bone, in the form of user documentation.

Developing user documentation is a subject unto itself, but deserves to be brought up here. We can generally assume that it will not include any implementation details, and will focus primarily on the user interface and the processes our applications use.

When writing user documentation, it's often sufficient to take the user stories, decorate them with screenshots and extra expository text, and leave it at that. It really depends on the client's requirements how much (if any) user-specific documentation is needed. If it's an application which will be used inside the client's business, it may be sufficient to provide one or more onsite training sessions.

One thing worth mentioning is that a screenshot can often save oodles of writing effort, communicate ideas more clearly, and remain easily deliverable through the application itself, in a training environment, and so on.

Screenshots can be a valuable documentation tool at many levels, including communications with our client when we're trying to illustrate a point difficult to communicate via text or still images alone.

Documenting development

The last form of documentation we'll look at is development documentation. This goes beyond our UML diagrams, user manual, functional specification, and so on. Development documentation includes the source control and issue tracking systems, the reasoning behind design decisions, and more. We'll take a quick look at some information we can use from each of these systems to create a path through the development itself.

Source code control systems

A **Source Code Control System** (**SCCS**) is an important part of the development process. Our SCCS is more than just a place to dump our source code—it's an opportunity to give a high-level overview of system changes.

The best ways to use our SCCS are dependent on which SCCS we use. However, there are a few quick ideas we can adopt across any SCCS and use them to extract a variety of information about our development streams.

Most clients will have their preferred SCCS already in place. If our deliverable includes source, it's nice if we can provide it in a way that preserves our work history.

Code and mental history

The history of change can be used on several levels, in several ways. There are products available that can help analyze our SCCS, or we can analyze it ourselves depending on what kind of information we're looking for.

For example, the number of non-trivial changes made to a file provides information in itself—for whatever reason, this file gets changed a lot. It's either an important document, a catchall, a candidate for parameterization, and so on. If two files are always modified together, then there's a chance of an unnecessarily tight coupling between them.

Sometimes, we just need to know what we were working on a for a particular date(s). We can retrieve all of our SCCS interaction for auditing purposes, to help determine what we were doing on a particular date, as part of a comprehensive change and time tracking system, and so on.

Commit comment commitment

We should view our commit comments as an important part of development documentation. One way to normalize commit comments is to create them as Javadoc-like comments, but different. Mostly, this just means that the first sentence is a succinct summary of the unit of work done, and the remaining sentences describe what was actually done.

What that first sentence includes is somewhat dependent on the rest of the development infrastructure. It's reasonable to put an issue tracking reference (see next section) as the most prominent part of that comment, perhaps followed by the same summary sentence as the issue item, or a summary if that's too long.

The rest of the commit comment should include any information deemed useful, and might include general change information, algorithm changes, new tests, and so on. This is the section for comments such as "Aaaaaaaaarg!", which aren't useful summaries, although it's often the most accurate.

Having a summary commit comment sentence also allows tools to get the output of history or log commands, and create a new view of existing information when necessary. For example, getting a list of files we changed between certain dates, along with a summary of why the files were changed. These can be used as a part of release notes, high-level summaries, and so on.

When (and what) do we commit

We should tend to commit more rather than less. The more recently a change was made, the easier it is to remember why and what was modified. Update the spelling in a single comment? Sure, might as well commit. When that file is changed later, and you're code-reviewing the changes, it's easier to look at only significant changes, and not at some trivial changes such as a punctuation change made the day before.

Also, while combining related commits, strive to keep them as granular as possible. For example, let's say we've updated some functionality in an action. As we were doing that, we corrected a couple of spelling errors in some other files. In an ideal world, even minor non-code changes would get their own commit, rather than being lumped in with changes to the code itself. If we see a commit message of "corrected spelling", we can probably ignore it. If it's lumped in to an issue-specific commit, we need to check the file itself to know if it's really part of the issue, and we'll be disappointed to find out it was to fix a misspelled Javadoc.

However, in the real world, we're not always so disciplined. In that case, the commit would be commented with information about the actual issue being addressed. However, in the comments, we might note that some spelling changes were included in the commit.

Note that some SCCSs make the task of breaking up our commits easier than others.

Branching

Even relatively simple changes in application functionality might warrant an experimental branch in which we could play with reckless abandon. By indicating the start of a unit of work in our SCCS, we allow all of the changes related to that unit of work to be easily reproduced.

It also creates a mini repository within which we can keep revision control of our development spike. It keeps the experimental code and its changes out of our mainline code and isolates the changes based on a distinct unit of work, which makes us feel better about life in general.

If the experimental branch lasts a long time, it should be updated with the current trunk (the head revision) as the mainline development proceeds. This will ease integration of the experimental patch when it's completed and merged back into the mainline code.

Branching discipline

Just as commits should be as granular as possible, any branches we create should be tied as closely as possible to the underlying work being done. For example, if we're working on refactoring in an experimental branch, we shouldn't begin making unrelated changes to another system in the same branch. Instead, hold off on making those changes, or make them in the parent revision and update our revision against the mainline code.

Branches of branches? Perhaps, but the management of multiple branches gets very irritating very quickly and is rarely worth the effort.

Issue and bug management

It's equally important to maintain a list of defects, enhancements, and so on. Ideally, everyone involved in a project will use the same system. This will allow developers, QA, the the client, or anybody else involved, to create help tickets, address deficiencies, and so on.

Note that the structure for doing this varies wildly across organizations. It will not always possible or appropriate to use our client's system. In cases like this, it's still a good idea to keep an internal issue management system in place for development purposes.

Using an issue tracking system can consolidate the location of our high-level to-do list, our immediate task list, our defect tracking, and so on. In a perfect world, we can enter all issues into our system and categorize them in a way meaningful to us and/or our clients. A "bug" is different from an "enhancement" and should be treated as such. An **enhancement** might require authorization to implement, it could have hidden implications, and so on. On the other hand, a **bug** is something that is not working as expected (whether it's an implementation or specification issue), and should be treated with appropriate urgency.

The categories chosen for issue tracking also depend on the application environment, client, and so on. There are a few that are safe, such as bug, enhancement, and so on. We can also have labels such as "tweak" "refactoring" and so on. These are primarily intended for internal and developmental use in order to indicate that it's a development-oriented issue and not necessarily client driven.

Issue priorities can be used to derive work lists. (And sometimes it's nice to knock off a few easy, low-priority issues to make it seem like something was accomplished.) A set of defined and maintained issue priorities can be used as part of an acceptance specification. One requirement might be that the application shouldn't contain any "bug"-level issues with a priority higher than three, meaning all priority one and priority two bugs must be resolved before the client accepts the deliverable.

This can also lead to endless, wildly entertaining discussions between us and the client, covering the ultimate meaning of "priority" and debating the relative importance of various definitions of "urgent", and so on. It's important to have an ongoing dialog with the client, in order to avoid running into these discussions late in the game. Always get discrepancies dealt with early in the process, and always document them.

Linking to the SCCS

Some environments will enjoy a tight coupling between their SCCS and issue tracking systems. This allows change sets for a specific issue to be tracked and understood more easily.

When no such integration is available, it's still relatively easy to link the SCCS to the issue tracking system. The two easiest ways to implement this are providing issue tracking information prominently in the SCCS commit comment (as discussed in an earlier section) or by including change set information in the issue tracking system (for example, when an issue is resolved, include a complete change set list).

Note that by following a convention in commit comments, it's usually possible to extract a complete list of relevant source changes by looking for a known token in the complete history output. For example, if we always referenced issue tracking items by an ID such as (say) "ID #42: Fix login validation issue", we could create a regular expression that matches this, and then get information about each commit that referenced this issue.

Wikis

We all know what a wiki is, but I'd like to advocate a moment to highlight their use as a way to create documentation and why they're well-suited to an agile environment.

Wikis lower the cost of information production and management in many ways, particularly when it's not clear upfront all that will be required or generated. By making it easy to enter, edit, and link to information, we can create an organic set of documentation covering all facets of the project. This may include processes used, design decisions, links to various reports and artifacts—anything we need and want.

The collaborative nature of wikis makes them a great way for everyone involved in a project to extend and organize everything related to the project. Developers, managers, clients, testers, deployers, anybody and everybody related to the project may be involved in the care and upkeep of the project's documentation.

Developers might keep detailed instructions on a project's build process, release notes, design documents (or at least links to them), server and data source information, and so on. Some wikis even allow the inclusion of code snippets from the repository, making it possible to create a "literate programming" environment. This can be a great way to give a high-level architectural overview of an application to a developer, who may be unfamiliar with the project.

Many wikis also provide a means of exporting their content, allowing all or part of the wiki to be saved in a PDF format suitable for printed documentation. Other export possibilities exist including various help formats, and so on.

Lowering the barrier to collaborative documentation generation enables wide-scale participation in the creation of various documentation artifacts.

RSS and IRC/chat systems

RSS allows us quick, normalized access to (generally) time-based activities. For example, developers can keep an RSS feed detailing their development activities. The feed creation can come from an internal company blog, a wiki, or other means. The RSS feed can also be captured as a part of the development process documentation.

Particularly in distributed development environments, a chat system can be invaluable for handling ad hoc meetings and conversations. Compared to email, more diligence is required in making sure that decisions are captured and recorded in an appropriate location.

Both RSS and IRC/chat can be used by our application itself to report on various issues, status updates, and so on, in addition to more traditional logging and email systems. Another advantage is that there are many RSS and chat clients we can keep on our desktops to keep us updated on the status of our application.

And let's face it, watching people log in to our system and trailing them around the website can be addictive.

Word processor documents

Personally, I'm not in favor of creating extensive word processor documents as the main documentation format. There are quite a few reasons for that: it can be more difficult to share in their creation, more difficult to extract portions of documents for inclusion in other documents, some word processors will only produce proprietary formats, and so on.

It's substantially more flexible to write in a format that allows collaborative participation such as a Wiki, or a text-based format such as DocBook that can be kept in our SCCS and exported to a wide variety of formats and allow linking in to, and out of, other sections or documents.

When proprietary formats must be used, we should take advantage of whatever functionality they offer in terms of version management, annotations, and so on. When a section changes, adding a footnote with the date and rationalization for the change can help track accountability.

Note that some wikis can be edited in and/or exported to various formats, which may help them fit in to an established corporate structure more readily. There are also a number of services popping up these days that help manage projects in a more lightweight manner than has been typically available.

Summary

We've examined the many ways of creating documentation throughout the entire development process. In particular, we've tried to highlight ways in which we can use existing tools, artifacts, and processes in order to help generate a wide variety of information.

We've also learned that documentation can be a project in itself. We have also seen that a lot of documentation, while potentially invaluable, may also represent a liability if we are forced to maintain it manually. This leads us to assume that documentation generated from the code itself (or as a by-product of running that code) is easier to sell to both ourselves and our management.

References

A reader can refer to the following:

- Fluent Interfaces:
  `http://www.martinfowler.com/bliki/FluentInterface.html`
- Aspect Oriented Programming:
  `http://en.wikipedia.org/wiki/Aspect-oriented_programming`
- Design by Contract:
  `http://en.wikipedia.org/wiki/Design_by_contract`
- SpringContracts:
  `http://springcontracts.sourceforge.net`
- yDoc:
  `http://www.yworks.com/en/products_ydoc.html`
- UmlGraph:
  `http://www.umlgraph.org`
- ArgoUML:
  `http://argouml.tigris.org`
- Quick Sequence Diagram Editor:
  `http://sdedit.sourceforge.net/`
- JsDoc Toolkit:
  `http://code.google.com/p/jsdoc-toolkit/`

Where to buy this book

You can buy **Apache Struts 2 Web Application Development** from the Packt Publishing website: `http://www.packtpub.com/apache-struts-2-web-application-development-beginners-guide/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
http://www.packtpub.com/apache-struts-2-web-application-development-beginners-guide/book