IBM

developerWorks.

Evolutionary architecture and emergent design: Emergent design through metrics

Using metrics and visualizations to find and harvest hidden design in your code

Skill Level: Intermediate

Neal Ford (nford@thoughtworks.com)
Software Architect / Meme Wrangler
ThoughtWorks

30 Jun 2009

Software metrics can help you find hidden design elements in your code, enabling them to emerge as idiomatic patterns. This installment of *Evolutionary architecture and emergent design* shows how intelligent use of metrics and visualizations lets you discover important code elements that are obscured by accidental complexity.

One of the difficulties for emergent design lies in finding idiomatic patterns and other design elements hidden in code. Metrics and visualizations help you identify important parts of your code, allowing you to extract them as first-class design elements. The two metrics I focus on in this article are *cyclomatic complexity* and *afferent coupling*. Cyclomatic complexity is a measure of the relative complexity of one method versus another. Afferent coupling represents the count of how many other classes use the current class. You'll learn about some tools for visualizing and understanding both of these metrics and how combining metrics can help you reveal design characteristics.

About this series

This series aims to provide a fresh perspective on the often-discussed but elusive concepts of software architecture and design. Through concrete examples, Neal Ford gives you a solid grounding in the agile practices of *evolutionary architecture* and *emergent design*. By deferring important architectural and design decisions until the last responsible moment, you can prevent unnecessary complexity from undermining your software projects.

© Copyright IBM Corporation 2009. All rights reserved.

I covered cyclomatic complexity in "Test-driven design, Part 2," but it has some nuances I didn't discuss there. One complicating factor of cyclomatic-complexity measurements via Java™ tools is the unit of work. Cyclomatic complexity is a method-level measurement, but the unit of work in Java programming is the class. Consequently, cyclomatic-complexity measurements generally come as either the sum or the average of the complexity of all the methods in a class. Both measures are interesting.

For example, the following scenario is possible. Let's say a class has one massively complex method (CC = 40), but also lots of very small methods (such as the get/set method pairs common in Java code). With a tool such as JavaNCSS (see Resources) that reports this metric as the sum of all the methods, the cyclomatic complexity is a high number for the entire class. If you use a tool such as Cobertura — which reports the cyclomatic complexity as an average for the class — this class no longer looks so bad, because the slew of simple methods is amortizing the highly complex one. Because of this unit-of-work mismatch, it makes sense to look at both sum and average measures of cyclomatic complexity. If you consider them independently, noise can creep into the results. Using both numbers mitigates that possibility.

Software metrics vs. physical metrics

In software, a metric refers to applying an objective measurement to development artifacts to determine coarse-grained characteristics. Unlike a physical metric (such as a meter stick), most software metrics don't reflect some characteristic in the real world. A cyclomatic-complexity number such as 5 has no units of measurement; it tells you nothing about any physical properties of the code. The number only makes sense when compared to the cyclomatic complexity of other code.

The other metrics of interest for design are the two coupling numbers: *efferent* and *afferent* coupling. Efferent coupling measures the number of classes the current class references. It is easy to determine via simple inspection: open the class in question and count the references (in fields and parameters) to other classes. Afferent coupling is harder to determine and much more valuable. It measures how many other classes use the current class. You can use command-line fu to determine this, or one of several tools that understand this metric. One such tool is ckjm, an open source tool for running the Chidamber & Kemerer object-oriented metrics suite (see Resources). Although a bit complicated to get up and running, it provides both cyclomatic complexity (reported as the sum of the cyclomatic complexity of all the methods of a class) and both efferent and afferent coupling numbers.

Once you have those numbers, though, what do they mean, especially in terms of design? The numbers generated as metrics provide a single dimension of

© Copyright IBM Corporation 2009. All rights reserved.

information about your code, and the raw numbers themselves frequently don't mean much. You can generate useful information from metrics in two ways. One is to look at how a particular value changes over time and spot trends. Or you can combine metrics to enrich the information density, which is the approach I'll show you in this article.

Metrics and design

I've been torturing the Struts code base in several articles in this series — not because I have a bias against Struts, but because it is a well-known open source project. Trust me: you can get unattractive design characteristics from most code you can find in the world! Having started with Struts, I'll continue using it to illustrate my points.

The output of ckjm is text, which is convertible to XML (and, via various transformations with XSLT, to other formats). Figure 1 shows the combination of several of the ckjm metrics, where *WMC* (Weight Methods per Class) is the sum of the cyclomatic complexity of the methods of the class and *Ca* is the afferent coupling:

Figure 1. ckjm metrics results in a table

...version="1.0"

	.classname	WMC	DIT	NOC	CBO	RFC	LCOM	Ca	NPM
1	org.apache.struts2.dispatcher.mapper.DefaultActionMapper$2$3	2	1	0	7	10	0	1	1
2	org.apache.struts2.views.velocity.components.BeanDirective	3	0	0	4	5	3	0	2
3	org.apache.struts2.util.MergeIteratorFilter	6	0	0	2	16	0	1	6
4	org.apache.struts2.components.template.VelocityTemplateEngine	5	0	0	12	32	6	0	3
5	org.apache.struts2.views.jsp.ui.AnchorTag	6	0	0	4	10	13	0	5
6	org.apache.struts2.views.freemarker.StrutsBeanWrapper$FriendlyMapModel$1	2	1	0	5	4	1	1	1
7	org.apache.struts2.portlet.PortletRequestMap	8	2	0	3	31	0	2	6
8	org.apache.struts2.views.jsp.ui.TextFieldTag	7	0	2	4	13	11	2	6
9	org.apache.struts2.portlet.servlet.PortletServletConfig	6	1	0	1	12	0	0	6
10	org.apache.struts2.views.freemarker.tags.FileModel	2	0	0	4	4	1	1	1
11	org.apache.struts2.portlet.context.PortletActionContext	17	1	0	3	22	136	7	16
12	org.apache.struts2.dispatcher.VelocityResult	10	0	0	13	40	39	0	5
13	org.apache.struts2.components.TabbedPanel	12	0	0	2	20	54	3	9
14	org.apache.struts2.interceptor.ParameterAware	1	1	0	0	1	0	1	1
15	org.apache.struts2.views.velocity.components.AbstractDirective	8	0	43	12	34	28	43	5
16	org.apache.struts2.interceptor.CookieInterceptor	7	0	0	9	30	0	0	4
17	org.apache.struts2.interceptor.ProfilingActivationInterceptor	5	0	0	4	14	0	0	5
18	org.apache.struts2.components.table.WebTable$WebTableRowIterator	5	1	0	1	11	4	1	3

Figure 2 shows the same table, sorted by WMC:

Figure 2. ckjm metrics, sorted by WMC

© Copyright IBM Corporation 2009. All rights reserved.

classname	WMC	Ca
org.apache.struts2.components.DoubleListUIBean	66	3
org.apache.struts2.views.jsp.ui.AbstractDoubleListTag	66	2
org.apache.struts2.views.xslt.AbstractAdapterNode	57	4
org.apache.struts2.portlet.util.HttpServletRequestMock	57	1
org.apache.struts2.components.UIBean	53	22
org.apache.struts2.components.Tree	51	3
org.apache.struts2.views.freemarker.tags.StrutsModels	50	1
org.apache.struts2.views.jsp.ui.TreeTag	49	0
org.apache.struts2.components.OptionTransferSelect	44	3
org.apache.struts2.views.xslt.SimpleAdapterDocument	44	2
org.apache.struts2.views.jsp.ui.OptionTransferSelectTag	43	0
org.apache.struts2.dispatcher.Dispatcher	37	19
org.apache.struts2.views.jsp.ui.AbstractUITag	34	18
org.apache.struts2.components.InputTransferSelect	34	3
org.apache.struts2.components.table.WebTable	33	12
org.apache.struts2.views.jsp.ui.InputTransferSelectTag	33	0
org.apache.struts2.components.Component	28	177
org.apache.struts2.components.Autocompleter	26	3
org.apache.struts2.views.jsp.ui.SubmitTag	25	0
org.apache.struts2.components.Form	24	10
org.apache.struts2.views.xslt.AbstractAdapterElement	24	6

Just by looking at this result, you can tell that `DoubleListUIBean` is the most complex class in the Struts code base. That suggests that it is a good candidate for refactoring to remove some of the complexity and see if you can find some abstractable, repeating patterns. However, the WMC number doesn't tell you whether investing in refactoring this class toward better design is a good use of time. Notice that the Ca for the class is 3. Only three other classes use this class, which suggests it's not worth investing lots of time improving the class's design.

Figure 3 shows the same ckjm results, sorted this time by Ca:

Figure 3. ckjm results, sorted by afferent coupling

© Copyright IBM Corporation 2009. All rights reserved.

classname	WMC	Ca
org.apache.struts2.components.Component	28	177
org.apache.struts2.views.freemarker.tags.TagModel	7	47
org.apache.struts2.views.velocity.components.AbstractDirective	8	43
org.apache.struts2.StrutsException	7	23
org.apache.struts2.components.UIBean	53	22
org.apache.struts2.dispatcher.mapper.ActionMapping	13	20
org.apache.struts2.views.jsp.ComponentTagSupport	6	19
org.apache.struts2.dispatcher.Dispatcher	37	19
org.apache.struts2.views.jsp.ui.AbstractUITag	34	18
org.apache.struts2.views.xslt.AdapterFactory	9	16
org.apache.struts2.views.xslt.AdapterNode	10	15
org.apache.struts2.ServletActionContext	11	15
org.apache.struts2.components.table.WebTable	33	12
org.apache.struts2.dispatcher.mapper.ActionMapper	2	11
org.apache.struts2.components.template.TemplateEngine	2	10
org.apache.struts2.components.template.Template	7	10
org.apache.struts2.dispatcher.StrutsResultSupport	13	10
org.apache.struts2.components.Form	24	10
org.apache.struts2.components.ListUIBean	8	9
org.apache.struts2.util.MakeIterator	3	8
org.apache.struts2.StrutsStatics	0	7

This combined view shows that the most-used class in Struts is `Component` (not surprising, given that Struts is a Web framework). Although `Component` isn't as complex, it's used by 177 other classes, which makes it a good candidate for design improvements. Making the design of `Component` better has a ripple effect on a large number of other classes.

The combination of WMC and Ca is the best way to read the perspective offered in Figure 3. This tells you both what's important and what's complex in the code base, in a single view. If you came to this code base with no prior knowledge, this view offers insights into where your effort potentially yields the best results. Although it's not infallible, you now have more information about the code base than you can derive from just looking at reams of code.

Numeric metrics provide insight into your code, but they exist at a pretty low level, providing information about specific classes but not much of a holistic view of a code base. Lots of tools are available now to take metrics to the next level via visualizations.

Metrics visualizations

Visualizations of metrics provide alternate views of specific dimensions, either of single dimensions or aggregations of several dimensions. The Smalltalk community created a huge number of metrics visualizations (and even created a platform, called Moose, to enable these kinds of visualizations; see Resources). Many of the metrics techniques developed by the Smalltalk community migrated to Java programming.

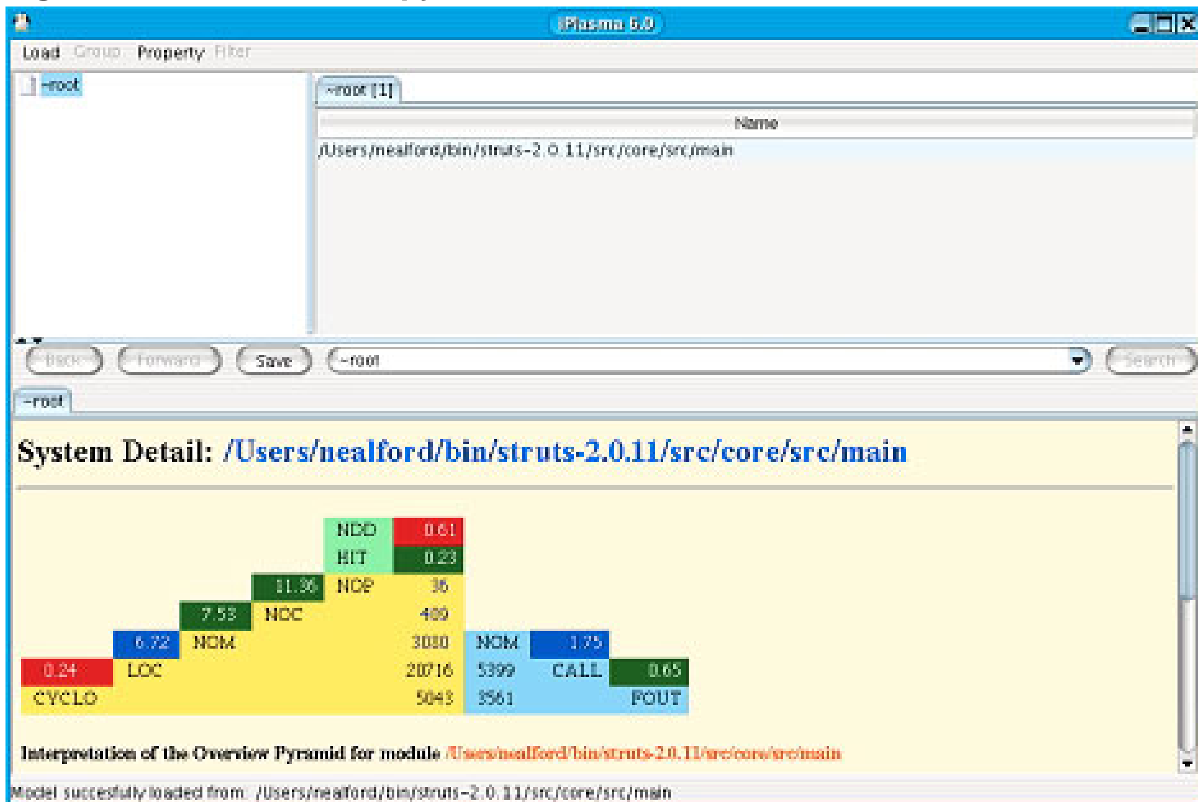
© Copyright IBM Corporation 2009. All rights reserved.

iPlasma and industry standards

Some of the common questions relating to cyclomatic complexity are "How does my code compare to others?" and "What is a good number for a particular class?" The iPlasma project answers these questions (see Resources). iPlasma is a platform, created as a university project in Romania, for quality assessment of object-oriented design. It generates a pyramid, showing key metrics for your project along with comparisons to industry-standard ranges for those numbers.

When you run iPlasma, you point to a source-code directory, and it churns away for a bit, producing a metrics pyramid like the one in Figure 4, which is based on the Struts 2.0.11 code base:

Figure 4. iPlasma metrics pyramid



This pyramid is packed with information, once you understand how to read it. Each row has a colored percentage; the percentage is derived via the ratio of the number on this row and the one under it. Table 1 shows what the numbers indicate, starting at the top:

Table 1. Understanding the iPlasma pyramid

Code	Description
NDD	Number of direct descendants
HIT	Height of inheritance tree

© Copyright IBM Corporation 2009. All rights reserved.

NOP	Number of packages
NOC	Number of classes
NOM	Number of methods
LOC	Lines of code
CYCLO	Cyclomatic complexity
CALL	Calls per method
FOUT	Fan out (number of other methods called by a given method)

The numbers indicate the ratios; the colors indicate where the ratios fit into the industry-standard ranges (derived from numerous open source projects). Each ratio is either green (within the range), blue (below the range), or red (outside the range). For the Struts code base, NDD and CYCLO are outside the industry standards for those values, and LOC and NOM are below. The ranges used appear in Table 2:

Table 2. iPlasma industry ranges for metrics

	Low	Medium	High
CYCLO / Line	0.16	0.20	0.24
LOC / method	7	10	13
NOM / class	4	7	10
NOC / package	6	17	26
CALLS / method	2.01	2.62	3.20
FANOUT / call	0.56	0.62	0.68

iPlasma also generates advice based on the pyramid, shown in the iPlasma display immediately below the pyramid. Figure 5 shows the advice for Struts:

Figure 5. iPlasma advice

Interpretation of the Overview Pyramid for module /Users/nealford/bin/struts-2.0.11/src/core/src/main

Class Hierarchies tend to be of **average height** and **wide**
(i.e. inheritance trees tend to have base-classes with many directly derived sub-classes)

Classes tend to:

- contain an **average** number of methods;
- be organized in **average-sized packages** ;

Methods tend to:

- tend to be rather **short** yet having a rather **complex logic** (i.e. many conditional branches);
- tend to call **few methods** (low coupling intensity) from a **several other classes** ;

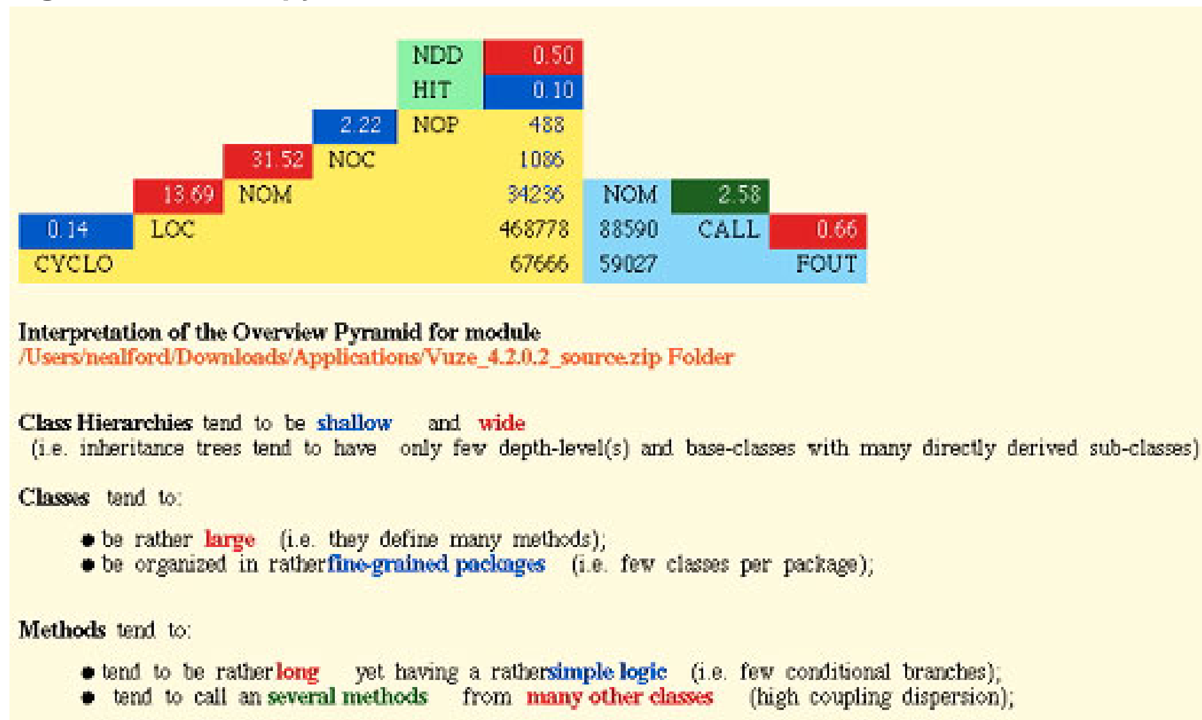
The numbers iPlasma generates serve a couple of purposes. First, they allow you to

© Copyright IBM Corporation 2009. All rights reserved.

compare your code base to others along several dimensions. Second, these numbers indicate places where you might want to expend effort to improve code hygiene and design. For example, for Struts, iPlasma indicates that the depth of the inheritance tree is pretty high and that the methods tend to be too complex. However, you must understand these numbers in context. A Web framework like Struts will tend to have a pretty elaborate hierarchy, meaning that the NDD number probably doesn't warrant concern. The CC number, though, has nothing to do with the context — it is too high and indicates a design smell at the method level.

For comparison purposes, Figure 6 shows an iPlasma pyramid for the Vuze project, an open source BitTorrent client written in the Java language (see Resources):

Figure 6. iPlasma pyramid for Vuze



Vuze is a large project (more than 500,000 lines of code), with potential design issues around the depth of inheritance tree, the number of methods in each class, the number of lines of code per method, and the number of calls per method.

Dependencies

Emergent design requires visibility into relationships and other high-level abstractions in your code. Trying to see these high-level concepts from source code calls to mind the famous image of blindfolded people trying to understand an elephant solely by touch. Each part of the elephant seems like something else, but touch is too localized to allow a holistic view.

Determining dependencies between classes and objects suffers from the same kind

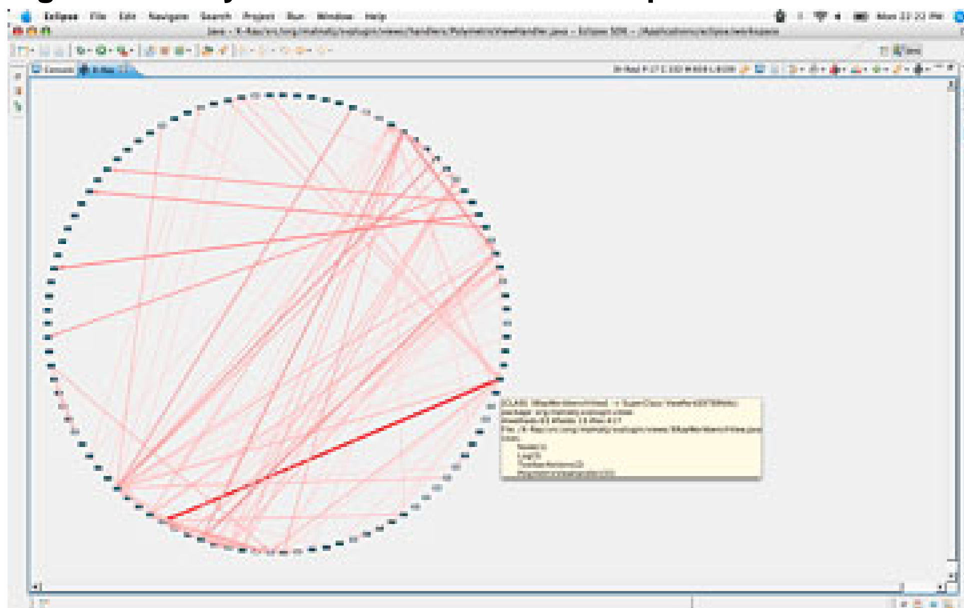
© Copyright IBM Corporation 2009. All rights reserved.

of localization problem. Tools like iPlasma allow you to see summaries of the overall characteristics of the code, but they don't tell you specific sites to investigate. Fortunately, other tools can help you see the elephant in several different lights.

The Smalltalk community created a tool called CodeCrawler (see Resources), based on the Moose platform, which shows a graphical representation of code, with dimensions for class size, method length, and some other metrics. It is possible to get CodeCrawler to work with Java code, but it's daunting. Fortunately, you don't have to fight that battle because the X-Ray project already has (see Resources).

X-Ray is an Eclipse plug-in that produces several visualizations to help you see your code's overall structure, including pie views of dependencies between classes, as shown for Struts in Figure 7:

Figure 7. X-Ray visualizations for class dependencies

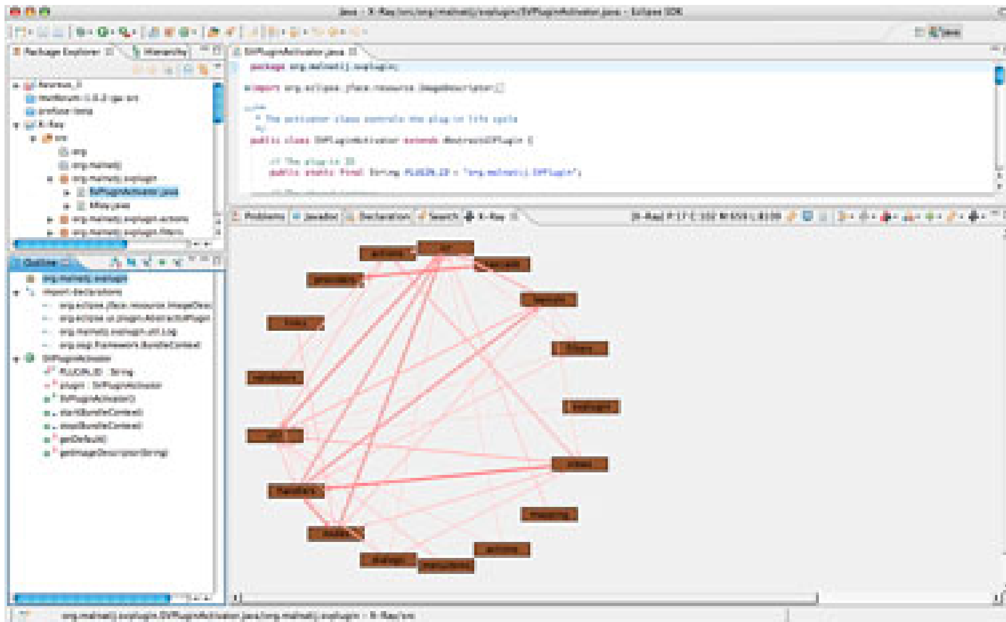


Each element along the edge of the circle is a class, and the lines indicate the dependencies between the classes. The boldness of the line indicates the strength of the dependency. Clicking on the class shows information about the class, and double clicking on it opens the class in the Eclipse editor. Of course, this view includes too much information to be useful. Fortunately, you can zoom in to see the individual lines. The bold lines indicate strong dependencies between classes (efferent coupling), which might indicate a design flaw when two classes are too intimately related to each other.

X-Ray also includes a similar view for package dependencies, shown in Figure 8:

Figure 8. X-Ray view of package dependencies

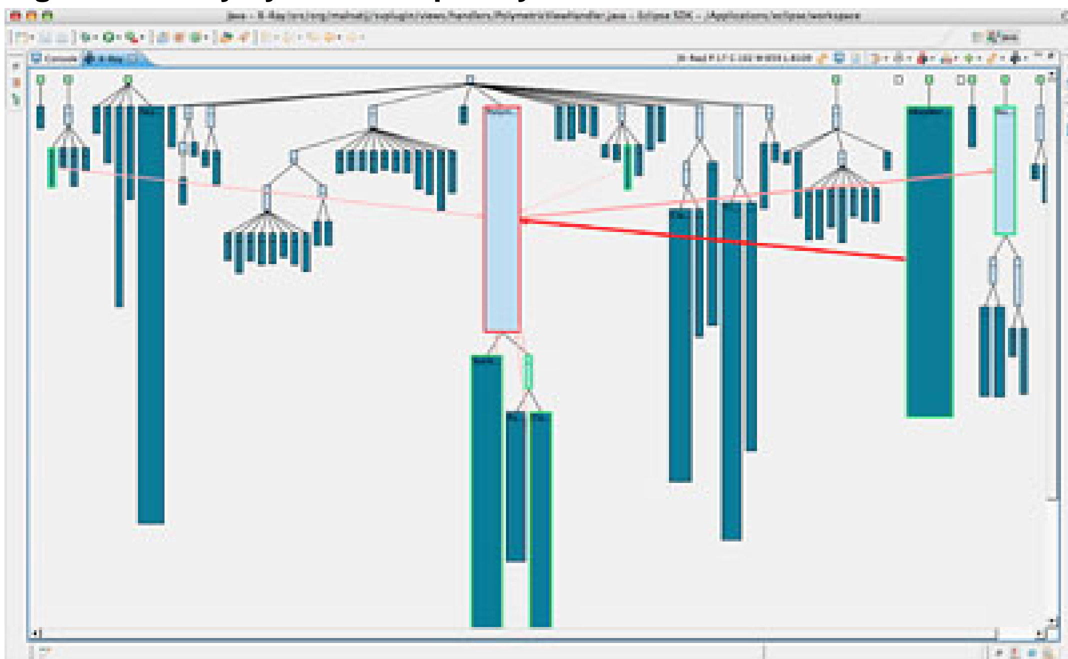
© Copyright IBM Corporation 2009. All rights reserved.



Overall structure

One other X-Ray view shows a useful code visualization, also based on CodeCrawler. The system-complexity view shows your code base as a graph in which the inheritance hierarchy appears as a top-down tree view, the size of the box indicates the number of lines of code in the class, and the box's width indicates the number of methods. Figure 9 shows the system-complexity visualization:

Figure 9. X-Ray system-complexity view



This view also shows outgoing calls (efferent coupling) as pink lines and incoming

© Copyright IBM Corporation 2009. All rights reserved.

calls (afferent coupling) as red lines. As in the previous visualizations, clicking on one of the boxes takes you to the class in Eclipse. This view of your code provides a unique perspective hard to achieve by looking at code. Finding design flaws around particular aspects becomes easier if you can quickly filter along certain dimensions, narrowing the parts of your code you should investigate further.

Summary

X-Ray and iPlasma represent just a small set of the types of visualizations available for Java code. Their judicious use allows you to narrow your focus quickly to aspects of your design that hide in the swamp of your project's code. Finding idiomatic patterns is one of the key enablers for emergent design, and tools that make it easy to see patterns (both good and bad) greatly reduce the investigatory effort, leaving more time for refactoring your code to make it better.

© Copyright IBM Corporation 2009. All rights reserved.

Resources

Learn

- *The Productive Programmer* (Neal Ford, O'Reilly Media, 2008): This book expands on a number of the topics in this series.
- Chidamber & Kemerer object-oriented metrics suite: Learn more about the metrics suite that the ckjm tool leverages.
- CodeCrawler: This project, which provides metrics visualizations for Smalltalk, is implemented for Java metrics by the X-Ray tool.
- Moose: Moose, an environment for Smalltalk metrics and visualizations, is the basis for CodeCrawler.
- *Object-Oriented Metrics in Practice* (Michele Lanza and Radu Marinescu, Springer, 2006): Explore object-oriented metrics further in this book co-authored by CodeCrawler's creator (Lanza).
- Vuze: The Vuze BitTorrent client's code base is used for an example in this article.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- JavaNCSS: This command-line utility measures source-code metrics for the Java language.
- ckjm: This open source tool generates Chidamber/Kemerer object-oriented metrics suite results for Java code.
- iPlasma, An Integrated Platform for Quality Assessment of Object-Oriented Design: Download iPlasma.
- X-Ray: Download an open source Java version of CodeCrawler.

Discuss

- Get involved in the My developerWorks community.

About the author

Neal Ford
Neal Ford is Software Architect and Meme Wrangler at **Thought**Works, a global IT

© Copyright IBM Corporation 2009. All rights reserved.

consultancy. He is also the designer and developer of applications, instructional materials, magazine articles, courseware, and video/DVD presentations, and he is the author or editor of books spanning a variety of technologies, including the most recent *The Productive Programmer*. He focuses on designing and building large-scale enterprise applications. He is also an internationally acclaimed speaker at developer conferences worldwide. Check out his Web site.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

© Copyright IBM Corporation 2009. All rights reserved.