Structural Epochs in the Complexity of Software over Time

Raghvinder S. Sangwan, Pamela Vercellone-Smith, and Phillip A. Laplante,
Pennsylvania State University

As software products evolve, complexity shifts from lower to higher structural levels, or vice-versa. Managing this complexity could require periodic major restructuring of software applications.

Software's structural complexity depends on the code base's hierarchical organization, beginning at the method level and migrating up through the class, package, and component levels. This complexity also depends on the nature of the dependencies that bind the different software elements together. In and of itself, structural complexity is not necessarily a problem. In fact, most large systems tend to be highly complex. John McGregor offers an interesting distinction between complex and complicated systems.[1] Complexity may be inherent in the problem domain and, therefore, may require

a *complex* solution. Such a solution might require time to understand because of the many interacting parts, but it has a rational structure that is understandable. A *complicated* solution, on the other hand, is difficult to understand and analyze. Clearly, designers must avoid complicated solutions and properly manage those that are complex. Otherwise, *excessive structural complexity* (XS) can become increasingly problematic for the ongoing development, testing, maintenance, and reuse of software.

Using a new complexity measurement framework called Structure 101 (www.headwaysoftware.com/products/structure101/index.php), we conducted a case study to track the structural complexity of three open source software products as they evolved through their different releases. Our analysis shows that as software products evolve, complexity shifts from lower to higher structural levels, or vice-versa. Refactoring efforts can reduce complexity at lower levels (for example, within leaf packages or methods), but they shift complexity to higher levels in the design hierarchy. Managing complexity at these

higher levels requires design restructuring or architecture reengineering. This cycle of shifting complexity can lead to points of excessive complexity in a software product, called *epochs*. If this pattern holds true for most software products, then mere refactoring at the code level might not be sufficient to effectively manage structural complexity in software. Periodic major restructuring of software applications at either the design or architectural level could be necessary.

Traditional complexity metrics

Complexity's relative impact varies depending on the abstraction level in the design hierarchy. For example, cyclic dependencies between software packages and components have a greater impact than excessively complex code at the method level, because a change in one package or component can adversely affect all dependent packages or components.

At the method level, designers assess a software system's structural complexity (also known as algorithmic complexity) by measuring its *cyclomatic*

complexity,[2] which is the number of possible execution paths through the body of code. The higher the value of cyclomatic complexity, the more complex the method code.

Designers assess structural complexity in a software system at higher abstraction levels such as a class, package, or component level by measuring coupling and cohesion metrics.[3,4] Excessively complex systems tend to have high coupling and low cohesion. High coupling implies many interacting parts forming a dependency graph with several *nodes* (representing software components, packages, or classes) and *edges* (representing relationships between software components, packages, or classes). Ideally, the coupling, measured as *afferent* (incoming dependencies) and *efferent* (outgoing dependencies), should be low, and the dependency graph should be acyclic.[5] Cycles in a dependency graph, especially large ones, form *tangles*, where the code becomes harder to develop, understand, maintain, and test. Low cohesion implies that parts within the system have many unrelated responsibilities. Designers assess cohesion by measuring relational cohesion,[5] which should be high.

These traditional metrics for cyclomatic complexity, coupling, and cohesion are not practical for quantitatively comparing complexity between different applications.[6] However, there are other techniques that are more suitable for performing such comparisons as well as for comparing complexity between different releases of the same application type.[7]

Structure 101 measurement framework

When working with large, complex code bases, traditional object-oriented metrics, such as those just described, offer only a limited snapshot of system complexity. They lack the capacity to visualize the impact of dependencies on emerging designs, particularly when these dependencies are *rolled up* through the structural hierarchy. ("Rolling up" means determining the complexity at a higher level in the structural hierarchy of a code base by recursively aggregating the complexities of its lower-level constituent parts. For instance, the complexity of a program can be determined from the complexity of its constituent packages, whose complexity in turn can be determined from their constituent subpackages, and so forth.) A simpler and definitive approach is necessary to compare two software programs' complexity, whether they are different programs or different versions of the same program.

The Structure 101 measurement framework provides a comprehensive view of structural complexity within a software system. This measurement framework is based on two aspects of XS: tangles and *fat* (see the "Measuring Excessive Complexity" sidebar). Tangles are cyclic dependencies between packages. Robert Martin's *acyclic dependency principle* dictates that "the dependency structure between packages must be a Directed Acyclic Graph (DAG). That is, there must be no cycles in the dependency structure."[5] Although cyclic dependencies between classes or methods within a package are often unavoidable, such dependencies between packages lead to highly coupled code that's difficult to maintain and extend because all the packages involved in a tangle are integrally tied to one another and affected by any changes introduced within the group. Fat measurement gives a sense of how easy it is to understand a given code base. Excessive dependencies among packages, subpackages, and classes, and excessive cyclomatic complexity within methods make the code harder to understand, test, and maintain.

The concepts used in the measurement of XS are based on Thomas McCabe's cyclomatic complexity,[2] as well as dependencies (including cyclical dependencies) between software elements. Software complexity studies have widely used these traditional metrics to characterize cohesion and coupling within a system. The only difference in the Structure 101 approach is how these complexity metrics are rolled up in the code base hierarchy to give an overall picture of how complex a particular system is. The roll-up mechanism calculates the complexity of an element higher up in the hierarchy by recursively aggregating the complexity of its constituent parts (see the sidebar).

Evolution in open source software

With the growing popularity of open source software, there is increasing interest in evaluating the development practices within open source communities. In the open source software development model, the project requirements are often loosely defined, continue to evolve over the lifetime of the application, and are driven by the interests of their user-developer communities.[8] As a consequence, high-level package design continually emerges along with the code-level design. As Lehman's laws predict, such emerging designs will become increasingly more complex over the software's evolution unless there are efforts to reduce or maintain it. Stephen Schach and his colleagues have observed this phenomenon, especially in the Linux operating system.[9] Neil Smith, Andrea Capiluppi, and Juan Ramil have observed it in 25 open source soft-

> The Structure 101 measurement framework provides a comprehensive view of structural complexity within a software system.

Measuring Excessive Complexity

The Structure 101 complexity measurement framework (www.headwaysoftware.com/products/structure101/index.php) calculates the *excessive structural complexity* (XS) of an element (such as a method, class, or package) within a software system by multiplying the degree of violation (a value normalized between 0 and 1) due to *tangles* and *fat* by its size. To calculate the degree of violation due to tangles (cyclic dependencies between packages), Structure 101 identifies the *minimal feedback set* (MFS), the minimum set of edges that must be removed from a dependency graph of a software system to eliminate cyclic dependencies. The framework divides the number of code-level references in the MFS by the total number of dependencies in the graph, and expresses this quotient as the degree of tangle violation. For example, the MFS for the graph shown in Figure A is 3. The number of code-level references in the MFS is 28(22 + 3 + 3). Therefore, the degree of tangle violation is (0.062)[28/(22 + 329 + 3 + 8 + 84 + 3)].

Structure 101 measures fat differently at each level of the code base hierarchy. The fat metric gives a sense of how easy it is to understand a given code base. At the higher-level packages (those containing subpackages), the framework calculates fat as the number of dependencies. For example, the parent of the graph in Figure A has a fat of 6 (simply count the edges and ignore the code-level references). At lower-level packages (those containing classes), the framework computes fat by counting the total number of dependencies between classes within the given package. To calculate the fat of a class, Structure 101 uses a dependency graph for the methods and attributes within the class to count the number of dependencies. Finally, the framework calculates a method's fat by using Thomas McCabe's cyclomatic complexity.[1]

The degree of fat violation depends on the extent to which fat exceeds the threshold. Structure 101 then converts the degree of violation to a normalized XS value (between 0 and 1), using the following equation:

$$\max\{0, [(value - threshold)/value]\}$$

The framework then multiplies the degree of tangle violation and the degree of fat violation by the item's size, which the framework measures according to the set of rules in Table A.

Finally, the framework calculates cumulative XS by recursively aggregating all the local XS values for a particular item's descendants. The average XS is the quotient of the cumulative XS and the size, and it reflects the percentage of the code base that is excessively complex.

Reference

1. T. McCabe, "A Complexity Measure," *IEEE Trans. Software Engineering,* Dec. 1976, pp. 308–320.
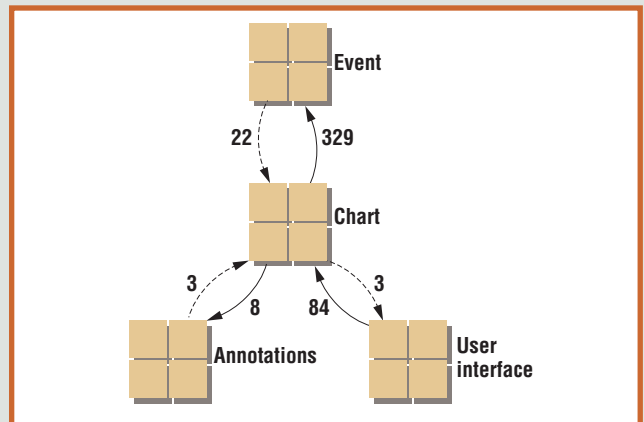
Figure A. Graph illustrating the calculation of the minimal feedback set (MFS). Edges represent dependencies. Numbers on edges represent code-level references. Dotted edges represent the edges in the MFS.

Table A

Set of rules used by Structure 101 to determine the size of each element in a software system

Element	Size
Field	1
Method	1 + NI*
Class (and inner classes)	1 + sum of sizes of all contained fields and methods
Leaf-level package	Sum of sizes of all contained classes
Design-level package	Sum of sizes of all contained packages

* NI is the number of bytecode instructions in the method body.

ware systems.[10] We applied the new complexity-monitoring technique to track structural complexity in software applications as they evolved through their different releases.

Case Study: Structural complexity in JFreeChart

To begin our study, we conjectured that a high proportion of structural complexity in the early releases would be found at the application-code level, which would then progressively migrate to higher-level design and architectural elements in subsequent releases. This pattern would then repeat itself throughout the evolution of a software product, requiring periodic maintenance efforts such as code refactoring, design restructuring, or architecture renovation (Figure 1). We presumed that this phenomenon would particularly hold for open source applications, in which a high-level design emerges with the code-level design.

We explored this phenomenon for 38 different releases of JFreeChart, an open source charting application written in Java. JFreeChart has been under active development for more than six years and has a well-documented history, with 38 versions available in the SourceForge repository (http://SourceForge.net). Because of these characteristics, several evolutionary studies have used JFreeChart.[11–13] In addition to studying the appropriate metrics, we reviewed the developer logs. Confirming our empirical evidence, developer release notes specifically indicate that package-splitting activities occurred whenever total XS decreased significantly. These package-splitting activities greatly reduced fat, but the packages remained integrally tied together, thus introducing cyclic dependencies at the design level. These observations support our hypothesis.

The distribution of each release of JFreeChart includes several jar files, but we focused on examining the complexity introduced during the evolution of the core graphics library (jfreechart.jar and jcommon.jar) and not any complexity from peripheral jar files (junit.jar, servlet.jar, and gnujaxp.jar). Figure 2 shows the average XS values per release obtained using only the core application jars.

For each of the 38 release versions of JFreeChart, Figure 3 shows the relative contribution of fat and tangles to the XS, broken down by design-level scope. Interestingly, early in JFreeChart's development, fat leaf packages were the greatest contributors to XS. With the ongoing development of JFreeChart, the XS increasingly moved to higher levels in the design hierarchy. This was particularly apparent during the first major restructuring effort, which occurred between releases 13 and 15.
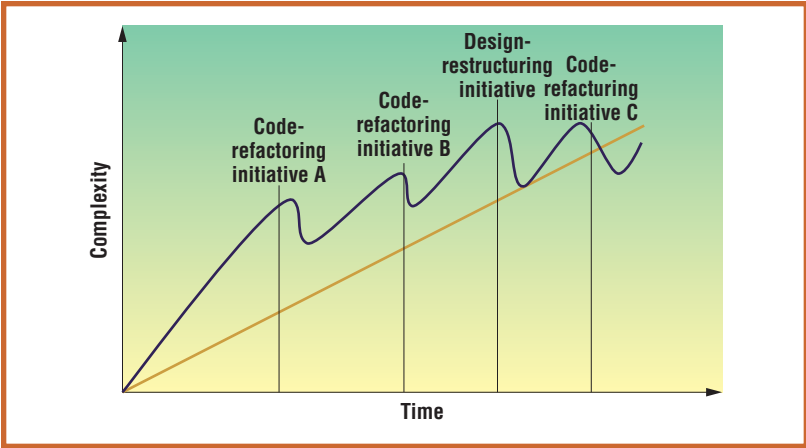

Figure 1. Idealized evolutionary pattern in which complexity grows until code refactoring, design restructuring, or architectural renovation is necessary. (The gold line represents idealized growth of source lines of code over time.)
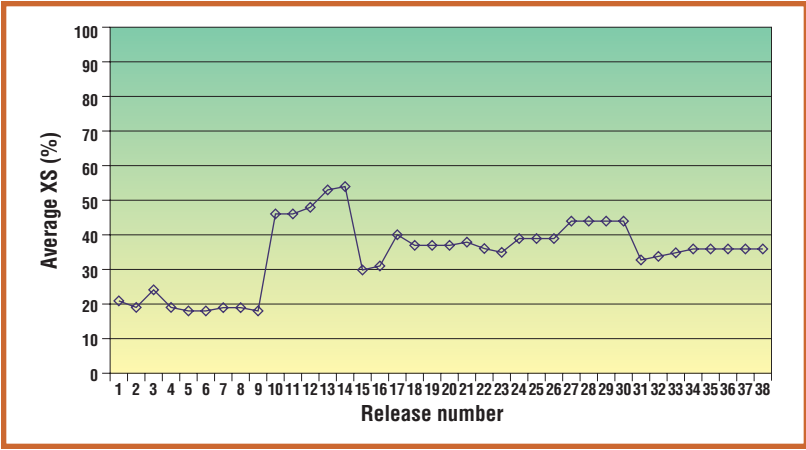

Figure 2. Average excessive structural complexity (XS) per release in JFreeChart.
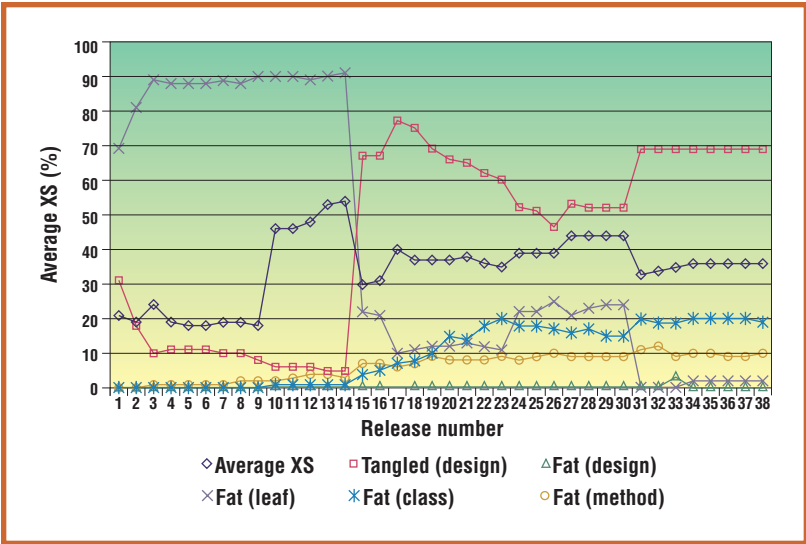

Figure 3. XS breakdown related to metric and design-level scope in JFreeChart releases.

Figure 4. Sources of XS contribution in the early XS change epoch: (a) release 9 and (b) release 10.

percent between the 9th and 10th releases. During this time, the release notes indicate the addition of new functionality with respect to new plot types and changes to the combination plot framework. These included

- adding capability for an area chart, a horizontal 3D bar chart, a Gantt chart, and a thermometer chart;
- reworking combination plots to provide a simpler framework; and
- restructuring code to facilitate combining category plots.

This epoch represents one of the few times during the evolution of JFreeChart when the code base's size decreased substantially.

In release 9 and earlier, the code for the demo application was packaged in the high-level design hierarchy. The demo code in release 9 was nearly 28 KNI (28,000 bytecode instructions). Beginning with release 10, the demo code resided in a separate jar, resulting in a relative decrease in the code base's size, from 94 KNI to 64 KNI. Although the cumulative XS grew substantially between these releases (as Figure 3 and Table 1 show), we must interpret the magnitude of this increase with caution because the code base's size concomitantly underwent a substantial decrease. The change in the relative average XS, therefore, might appear exceedingly inflated because of the code base's size reduction. Without the decrease in this size, the cumulative XS of 28,941 NI in release 9 would correspond to an average XS of approximately 31 percent, which would still be a substantial increase in XS.

As a general guideline, Structure 101 recommends keeping the complexity of packages to 60 dependencies, because having fat packages, classes, or methods would defeat the purpose of having a simple acyclic package structure. (We aren't trying to establish specific thresholds for fat, tangles, or XS that define what *complex* means. Rather, our

Because of this restructuring effort, overall XS decreased by 24 percent. This decrease was largely due to the reduction of fat leaf packages. However, as a consequence of the refactoring efforts to reduce fat packages, cyclic dependencies crept in, creating tangles in the design-level hierarchy. This permutation occurred again 16 months later, when the JFreeChart developers restructured release 30 to reduce fat leaf packages, which had grown excessively large. The complexity moved to the design level in release 31, with an increasing number of cyclic dependencies present in the high-level design hierarchy.

We selected for further study any consecutive releases of JFreeChart exhibiting more than a 10 percent increase or decrease in average XS. Among the 38 releases of JFreeChart, we identified three development periods, or epochs, that exhibited XS changes greater than 10 percent. As Table 1 shows, these development epochs occurred between releases 9 and 10, releases 14 and 15, and releases 30 and 31. We designate them as early, mid, or late, depending on when they occurred in the evolution of the product.

Early evolution epoch: Complexity begins to surface

The average XS increased sharply from 18 to 46

goal is to establish an evolutionary pattern on the basis of one complexity measure.) Although neither release 9 nor 10 contained items that exceeded the fat at the design level, fat leaf packages were a substantial problem in both releases. In fact, they were the greatest contributors to the XS (see Figure 4). In both versions, fat leaf packages contributed 90 percent of the total XS, whereas design tangles contributed 8 percent in release 9 and contributed 6 percent in release 10. During this early developmental period, fat methods also contributed to the XS. Release 9 had seven excessively complex items, four of which were fat methods. In release 10, fat methods accounted for six of the top 10 excessively complex items.

Mid-evolution epoch: Complexity begins to migrate

The release notes indicate major changes to the JFreeChart API in release 15. These changes included substantial architectural restructuring and the addition of new functionality. The structural changes included

- adding many new interfaces and modifying existing ones;
- creating separate packages for the axes (com. jrefinery.chart.axis), plots (com.jrefinery.chart. plot), and renderers (com.jrefinery.chart.renderer);
- moving responsibilities for category distribution and control of gridlines;
- adding new renderers for *xy* plots; and
- adding new classes, including `ContourPlot` and `MovingAverage`.

Additionally, the JFreeChart developers added to the functionality of the application by introducing support for features such as secondary axes, data sets, and renderers; support for reading pie data sets and category data sets; improvements to the legend; and modifications to the Gantt chart displays.

This substantial architectural design restructuring led to a decrease in average XS from 54 percent in release 14 to 30 percent in release 15. Moreover, the XS distribution profile varied greatly between the two versions. As Figure 5 shows, fat leaf packages contributed 91 percent of the total XS in release 14. In release 15, the developers substantially reorganized the fat leaf packages, which reduced the XS contribution from fat leaf packages to 22 percent of the total XS. Interestingly, the reduction of fat in release 15 correspondingly introduced additional XS into higher levels of the design hierarchy, with design-
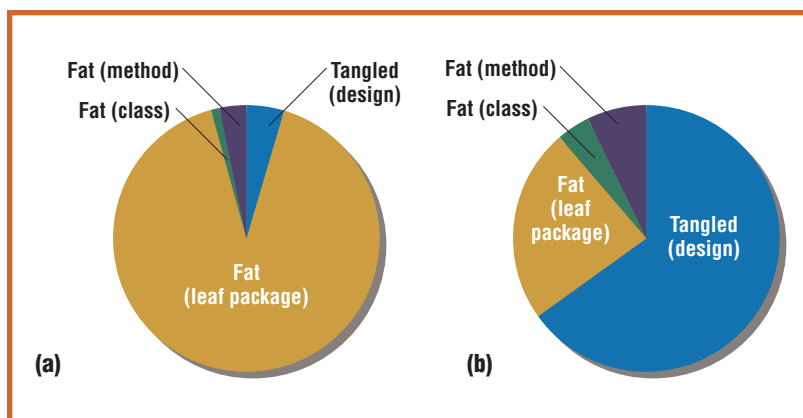

Figure 5. Breakdown of XS contribution in the mid XS change epoch: (a) release 14 and (b) release 15.
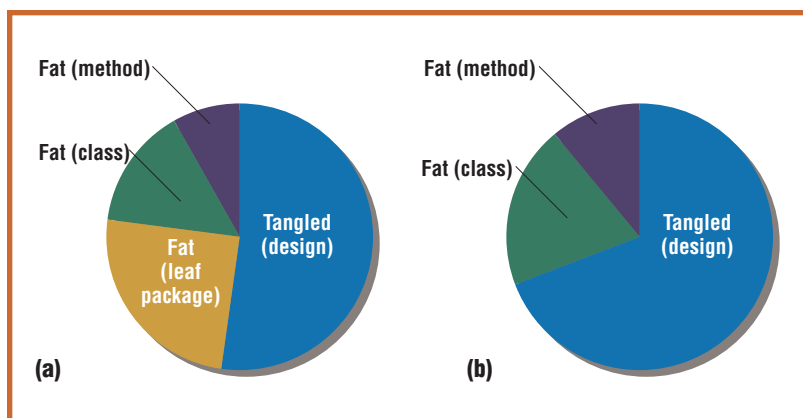

Figure 6. Breakdown of XS contribution in the late evolution phase: (a) release 30 and (b) release 31.

level tangles now comprising 67 percent of the total XS.

Late evolution epoch: Migration continues

Although the release notes accompanying release 31 indicated no major changes, the few changes made succeeded in reducing the average XS from 44 to 33 percent between releases 30 and 31, thanks to package splitting. The developers split the org.jfree. data and org.jfree.chart.renderer packages into subpackages for the category and *xy* charts. Aside from the creation of subpackages, there were only a few minor changes, but these seemingly minor changes nevertheless impacted the XS distribution profile between the two versions, as Figure 6 shows. In release 30, fat leaf packages contributed 24 percent of the total XS. Interestingly, the developers totally eliminated fat leaf packages in release 31, which largely explains the reduction in XS between these two releases. In release 30, design-level tangles contributed 52 percent of the total XS, whereas in release 31 they contributed 69 percent of the total XS. This late evolution epoch once again demonstrates that design restructuring is necessary to reduce the size of excessively
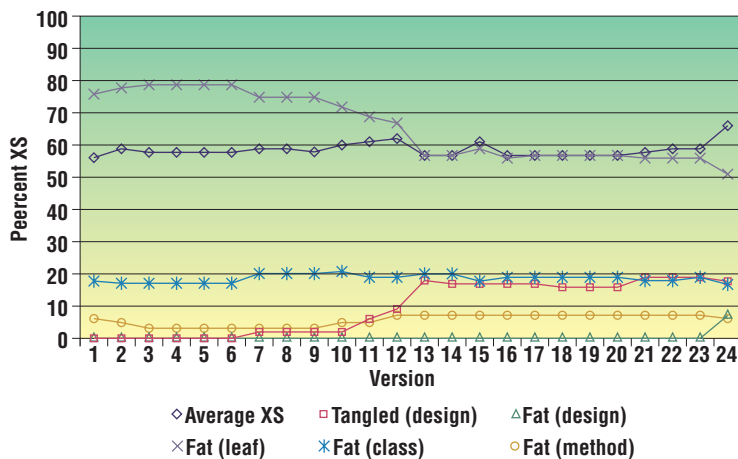
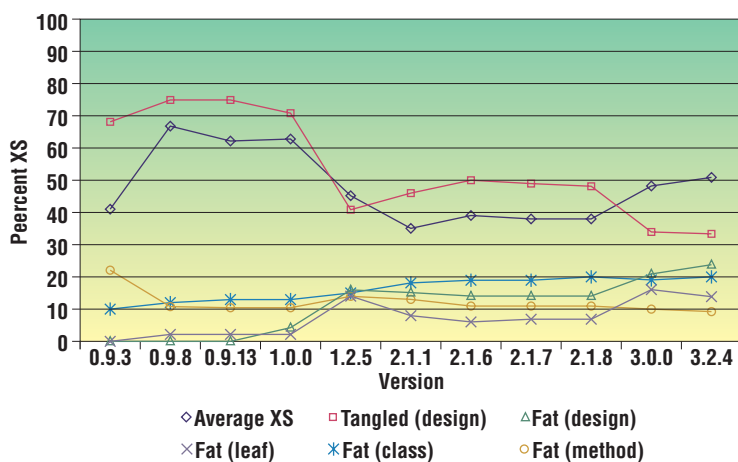**Figure 7. XS in the Findbugs open source software application.**

Figure 8. XS in Hibernate open source software application.

between releases 9 and 13. In release 9, fat leaf packages contributed 75 percent of the total XS, whereas tangles contributed only 2 percent of the total XS. There was a substantial shift in structural complexity over the next five releases: the XS contribution from fat decreased from 75 to 57 percent, and the percentage of tangles rose from 2 to 18 percent of the total XS. Unlike JFreeChart, where the average XS value decreased during major complexity shifts, in Findbugs the overall average XS metric value remained relatively constant (at about 57 to 62 percent) during this time frame.

We also observed significant shifts in the distribution of XS in Hibernate. However, in this case, the pattern of shifting complexity differed from that of JFreeChart and Findbugs. As Figure 8 shows, we observed large shifts in complexity between versions 1.0.0 and 1.2.5 of Hibernate, and also between versions 2.1.8 and 3.0.0. During both of these development periods, there was a substantial reduction in tangles, whereas excessive fat accumulated in both the design and leaf package levels.

The Findbugs and Hibernate applications provide further evidence that XS shifts during software evolution, but the exact pattern may vary from one application to another. In JFreeChart and Findbugs, first code became excessively complex, requiring refactoring efforts. Refactoring reduced complexity at the local level (for example, within leaf packages and methods) but shifted the complexity to a higher level in the design hierarchy. In Hibernate, on the other hand, XS originated at a higher level in the design hierarchy, and design restructuring efforts shifted complexity to the lower-level leaf packages. It isn't clear how much of this shifting nature of complexity is due to the lack of focus on high-level design or architecture early in the development life cycle of open source systems; the high-level design continually emerges with the code-level design. We intend to study this in our future research. 🎵

large fat leaf packages. As a consequence of code-refactoring efforts, the complexity moves to the design level, with cyclic dependencies contributing a larger proportion of the total XS.

Pattern of shifting structural complexity

To further examine the phenomenon of shifting structural complexity, we examined two additional open source applications: Findbugs and Hibernate. Findbugs is a static analysis tool for identifying bugs in Java programs. The Hibernate application supports the development of persistent classes following an object-oriented idiom.

Figures 7 and 8 show the results. With the Findbugs releases, we observed a complexity-shifting pattern similar to that of JFreeChart. Namely, as overall excess fat decreased, there was a concomitant increase in tangles introduced at the design level. As Figure 7 shows, this shift occurred

References

1. J.D. McGregor, "Complexity, Its in the Mind of the Beholder," *J. Object Technology*, Jan./Feb. 2006, pp. 31–37; www.jot.fm/issues/issue_2006_01/column3.
2. T. McCabe, "A Complexity Measure," *IEEE Trans. Software Engineering*, Dec 1976, pp. 308–320.
3. W.P. Stevens, G.J. Myers, and L.L. Constantine, "Structured Design," *IBM Systems J.*, vol. 13, no. 2, 1974, pp. 115–119.
4. S.R. Chidamber, and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Software Engineering*, June 1994, pp. 476–493.

5. R.C. Martin, *Agile Software Development*, Prentice Hall, 2003.
6. M. Larsson and P.A. Laplante, "On the Complexity of Design in Imaging Software," *Proc. 11th IEEE Int'l Conf. Engineering of Complex Computer Systems* (ICECCS 06), IEEE CS Press, 2006, pp. 37–42.
7. R. Sangwan, P. Vercellone-Smith, and P. Laplante. "Measuring the Complexity of Design in Real-Time Imaging Software," *Proc. 11th Real-Time Imaging Conf.*, SPIE, 2007, vol. 6486, pp. 1–6.
8. W. Scacchi, "Understanding Open Source Software Evolution," *Software Evolution*, N.H. Madhavji et al., eds., John Wiley & Sons, 2004.
9. S.R. Schach et al., "Maintainability of the Linux Kernel," *IEE Proc.—Software*, vol. 149, no. 1, 2002, pp. 18–23.
10. N. Smith, A. Capiluppi, and J.F. Ramil, "A Study of Open Source Software Evolution Data Using Qualitative Simulation," *Software Process Improvement and Practice*, vol. 10, no. 3, 2005, pp. 287–300.
11. Y. Crespo et al., "Language Independent Metrics Support towards Refactoring Inference," *Proc. 9th ECOOP Workshop Quantitative Approaches in Object-Oriented Software Engineering*, Springer, 2005, pp. 18–29.
12. Z. Xing and E. Stroulia, "Towards Experience-Based Mentoring of Evolutionary Development," *Proc. 21st IEEE Int'l Conf. Software Maintenance* (ICSM 05), IEEE CS Press, 2005, pp. 621–624.
13. Z. Xing and E. Stroulia, "Understanding the Evolution and Co-evolution of Classes in Object-Oriented Systems," *Int'l J. Software Engineering and Knowledge Engineering*, vol. 16, no. 1, 2006, pp. 23–52.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

About the Authors
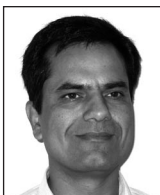
Raghvinder S. Sangwan is an assistant professor of information science in the Engineering Division of the Great Valley School of Graduate Professional Studies at the Pennsylvania State University. His research interests include analysis, design, and development of software systems; their architecture; and automatic and semiautomatic approaches to assessing their design and code quality. He received his PhD in computer and information sciences from Temple University. Contact him at rsangwan@psu.edu.

Pamela Vercellone-Smith is a research associate in the Engineering Division of the Great Valley School of Graduate Professional Studies at the Pennsylvania State University. Her research interests include software design complexity and open source software systems. She received her PhD in microbiology from the University of Delaware. She is a member of the ACM and the American Society for Microbiology. Contact her at pav115@psu.edu.

Phillip A. Laplante is a professor of software engineering in the Engineering Division of the Great Valley School of Graduate Professional Studies at the Pennsylvania State University. His research interests include software project management, the role of the CIO, and open source software systems. He received his PhD in computer science from the Stevens Institute of Technology. He is a Fellow of the IEEE and is currently a member of the IEEE Computer Society's Board of Governors. Contact him at plaplante@psu.edu.