Design Evolution of an Open Source Project Using an Improved Modularity Metric

Roberto Milev, Steven Muegge, and Michael Weiss

Department of Systems and Computer Engineering
Carleton University
Ottawa, ON K1S 5B6, Canada
rmilev@connect.carleton.ca, smuegge@sce.carleton.ca,
weiss@sce.carleton.ca

Abstract. Modularity of an open source software code base has been associated with community growth, incentives for voluntary contribution, and a reduction in free riding. As a theoretical construct, it links open source software to other domains of research, including organization theory, the economics of industry structure, and new product development; however, measuring the modularity of an open source software design has proven difficult, especially for large and complex systems. Building on previous work on Design Structure Matrices (DSMs), this paper describes two contributions towards a method for examining the evolving modularity of large-scale software systems: (1) an algorithm and new modularity metric for comparing code bases of different size; and (2) evolution analysis of Apache Tomcat to illustrate the insights gained from this approach. Over a ten-year period, the modularity of Tomcat continually increased, except in three instances: with each major change to the architecture or implementation, modularity first declined, then increased in the subsequent version to fully compensate for the decline.

1 Introduction

A growing body of theory and evidence suggests that modularity is of central relevance to open source software projects and open source communities. O'Reilly [16,17] argues that a highly modular "architecture of participation" is required to support the growth of communities around "systems that are designed for user contribution". Baldwin and Clark [2] provide a theoretical argument that a more modular open source code base will attract more voluntary contributions and have less free riding of non-contributors than one that is less modular. Other domains, including organization theory [19], and industry structure [4,14], and product design [24], also hold modularity as a central construct. It links the microstructure of these different domains, deep "in the very nature of things" [1, p. 2], in ways that enable theorizing about the fundamental nature and extent of their interconnectedness [9,21,10]. However, measuring the modularity of a software design has proven difficult in practice, especially for large and complex systems [13,15].

In this paper, we describe two contributions towards a method for examining the evolving modularity of large-scale software systems using Design Structure Matrices

C. Boldyreff et al. (Eds.): OSS 2009, IFIP AICT 299, pp. 20–33, 2009.

(DSMs) and modularity metrics. First, we extend work by MacCormack et al. [15], Fernandez [8], and Idicula [12] to develop an algorithm and new modularity metric for comparing code bases of different size. Second, we illustrate the application of our approach with a case study examining the evolution of Apache Tomcat from version 3.0 (released 1999) to version 6.016 (released 2008).

The body of this paper consists of seven remaining sections. Section 2 reviews the salient literature on modularity, DSMs, and the modularity of open source software projects. Section 3 describes a method for evolution analysis. Section 4 develops algorithms and modularity metrics. Section 5 examines the evolving modularity of Apache Tomcat. Section 6 discusses the findings. Section 7 concludes.

2 Background

This section reviews salient research on modularity, design structure matrices, and recent studies on the modularity of open source software platforms. Our conceptual framework closely follows that of Baldwin and Clark's design rule theory [1], which builds on well-established ideas in engineering design [21,22,6,7], software engineering [18], and complex adaptive systems [11]. Baldwin and Clark's groundbreaking work examined the role of modularity in the evolution of the computer industry, and has since been applied to software systems, e.g. [2,13,15].

2.1 Modularity and Software Design

The *design structure* of an artifact is a list of all design parameters and the physical and logical interdependencies between them [1, pp. 21 and 25]. A *module* is a unit whose structural elements are strongly connected among each other and relatively weakly connected to elements in other units [1, p. 63]. Just as there are degrees of connectedness, there are degrees of modularity, thus motivating interest in metrics and techniques to measure the modularity of the structures comprising an artifact.

When the complexity of one of the elements crosses a certain threshold, that complexity can be isolated by defining a separate abstraction that has a simple interface. The abstraction hides the complexity of the element; the interface indicates how the element interacts with the larger system [1,18]. Modularity decreases complexity in several ways. In particular, it allows designers to focus on individual modules rather than the whole integrated artifact. This radically changes the design process and allows for work on individual modules to be parallelized.

2.2 Design Structure Matrices

The *Design Structure Matrix* (DSM), pioneered by Steward [22] and extended by Eppinger [6,7], is an analysis tool for mapping complex systems. It provides a compact representation of a complex system that visualizes the interdependencies between system elements [3]. According to Baldwin and Clark [1, p. 43], "it is a powerful analytic device, because by using it we can see with clarity how the physical and logical structure of an artifact gets transmitted to its design process, and from there to the organization of individuals who will carry the process forward."

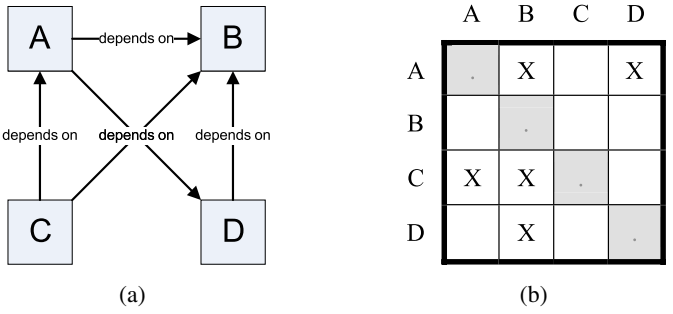

Fig. 1. (a) Example system with dependencies. (b) DSM representing the example system.

A DSM is a square matrix with off-diagonal cells indicating dependencies between the system elements. A value in d_{ij} (the cell at row i and column j) means that the element at position i depends in some way on the element at position j. Our analysis employs binary DSMs that indicate only the presence or absence of a dependency (an "X" or a blank cell, respectively). Alternatively, cells could contain numeric values with information about the strength of dependencies, and main diagonal cells could also contain information.

Figure 1(a) shows a simple system of four elements, labeled *A* to *D*. Arrows indicate a dependency relationship. Elements *A*, *C* and *D* all depend on element *B*. Element *C* also depends on element *A*, and element *A* also depends on element *D*. Figure 1(b) is the DSM that describes the dependencies between these elements. The DSM shows whether or not the row element depends on the column element. It is not symmetrical, because dependencies are not necessarily mutual (if element *A* depends on element *B*, this does not imply that element *B* also depends on element *A*).

The software systems examined here are implemented in Java. However, the approach and metrics proposed here are generally applicable, provided appropriate tools for extracting dependencies from the code base. The design elements are Java classes and our dependencies are references between classes. More formally, class *A* depends on class *B* if any of these conditions are met:

- Class *A* inherits from (extends) class *B*.
- Class *A* declares a field of class *B*.
- A method from class *A* calls a method in class *B*.

We extract this information automatically from the code base using a method described in section 4. *Clustering* reorganizes the DSM elements to more clearly visualize and analyze dependency relationships. The automated clustering algorithms employed in this study will be described in a separate paper.

2.3 Modularity and Open Source Software

Baldwin and Clark [2] argue on theoretical grounds that the architecture of a code base is a critical factor that lies at the heart of the open source development process. Drawing on design rule theory [1], they argue that designs have option-value because a new

design creates the right but not the obligation to adopt it. A modular design allows for experimentation and changes within modules without disturbing the functionality of the whole system. The authors use game theory to show that increased modularity (and thus increased option value) increases the *incentives* of developers to get involved and remain involved in the development process, and decreases the amount of *free riding* in the equilibrium. Both effects promote growth of the developer community, suggesting that modularity of design is critical to the success of an open source development project.

MacCormack et al. [15] employ DSMs to empirically compare the design structures of two software products, the Linux kernel and the Mozilla web browser. They propose a clustering algorithm to measure dependencies that is an important improvement over previous work [8,12]. By calculating marginal changes in cost rather than the total cost of the matrix, computation time is significantly reduced. However, the comparison of the Linux kernel and Mozilla critically depends on selecting versions of the systems that are comparable in terms of number of source files (elements in the DSM). One motivation of our work was to remove this restriction, and to allow the comparison of code bases of different size.

LaMantia et al. [13] build on [15] to examine the evolution over time of two software products, the open source Apache Tomcat application server and a closed source commercial server product (not identified by name). They introduce a coarse metric that represents the change ratio between the consecutive versions in the product evolution. The authors conclude that DSMs and Design Rule Theory [1] can explain how real-world modularization activities allow for different rates of evolution to occur in different modules, and create strategic advantage for a firm.

Looking across the literature on modularity, DSMs, and open source software, we find cogent arguments that a highly modular design is needed to attract and empower a community of developers around an open source project; designs that are more modular generate more opportunities for creating and exchanging work between open source developers. Additionally, we find a small but growing body of research employing DSMs, clustering algorithms, and modularity metrics to analyze the design of complex software systems.

3 Method

Our method for examining the evolving modularity of large-scale software systems implemented in Java builds on the DSM methods and algorithms of MacCormack et al. [15] and LaMantia et al. [13], but differs from past work in several aspects. As with [15,13], we automate dependency extraction from the software code base, employ design structure matrices for visualization and analysis of dependency information, and compute cost metrics as measures of modularity. We differ from [15] in our unit of analysis (Java classes rather than C source files) and from [15,13] in our use of the relative clustered cost metric (described in Section 4).

Our design elements are Java classes and our dependencies are references between classes, whether by inheritance, declared fields, or method calls. Because dependencies between Java classes can be extracted from the compiled code of a software system, we need only obtain binary distributions of the selected versions.

The steps of our method are as follows:

1. Select the versions to be analyzed and obtain their binary distributions.
2. For each version, extract the dependency information from the compiled code.
3. Create DSM instances and extract cost metrics.

4 Modularity Metrics

This section describes three algorithms and modularity metrics implemented. *Propagation cost* (Section 4.1) measures the extent to which a change in one element impacts other elements. It is a representation of the degree of coupling without consideration of the proximity between elements. *Clustered cost* (Section 4.2) is a more sophisticated metric that assigns different costs to dependencies based on the locations of elements within clusters. It has an important limitation in that it can only be used to compare DSMs of similar sizes [15]. *Relative clustered cost* (Section 4.3) extends the clustered cost metric to compare DSMs of different sizes. It, therefore, avoids the above limitation of the clustered cost metric. The propagation cost and clustered cost metrics were previously implemented by MacCormack et al. [15]. The relative clustered cost metric is a new contribution of this paper.

4.1 Propagation Cost

As noted in Section 2.2, each cell of the DSM holds a binary value that indicates the presence or absence of a dependency. Alternatively, we can think of this as a matrix, D, of direct dependencies, d_{ij}, or dependencies of path length 1:

$$D^1 = DSM = \begin{bmatrix} d_{11} & d_{12} & \dots & d_{1N} \\ d_{21} & d_{22} & \dots & d_{2N} \\ \dots & \dots & & \dots \\ d_{N1} & d_{N2} & \dots & d_{NN} \end{bmatrix} \tag{1}$$

We can identify indirect dependencies by raising D to successive powers; the results show the direct and indirect dependencies for successive path lengths:

$$D^i = D \times D^{i-1} \text{ if } i > 1 \tag{2}$$

The visibility matrix, V, is the sum of these matrices:

$$V = \sum_{i=0}^{N} D^i = \begin{bmatrix} v_{11} & v_{12} & \dots & v_{1N} \\ v_{21} & v_{22} & \dots & v_{2N} \\ \dots & \dots & & \dots \\ v_{N1} & v_{N2} & \dots & v_{NN} \end{bmatrix} \tag{3}$$

Following MacCormack [15], we are interested only in the presence or absence of dependencies. The binary visibility matrix, V', offers us computational advantages over the visibility matrix, V. Operations on binary numbers are performed more quickly

than operations on real numbers, permitting faster execution times for computing the successive powers of n for the dependency matrix, D:

$$V' = f(V) = \begin{cases} v'_{ij} = 0 \text{ if } v_{ij} = 0 \\ v'_{ij} = 1 \text{ if } v_{ij} > 0 \end{cases} \tag{4}$$

Propagation cost is the sum of all elements of the binary visibility matrix, V', divided by the square of N, the total number of design elements. It indicates the proportion of elements that may be affected on average, either directly or indirectly, when a change is made to one element in the system.

$$PropagationCost = \left(\sum_{i=0}^{N} \sum_{i=0}^{N} v'_{ij} \right) / N^2 \tag{5}$$

4.2 Clustered Cost

The propagation cost measure only considers the existence of dependencies, and does not consider whether the dependencies occur between elements located in the same or different modules of the system. The *clustered cost* measure assigns different costs to dependencies based on the location within clusters of the elements between which they occur. This requires clustering the DSM nearly into an "idealized modular form" [15], in which there are no dependencies between clusters.

Idicula [12] described a *total coordination cost* clustering algorithm that groups tasks into clusters with minimal interdependencies. The algorithm is noteworthy because it allows for clustering when the number and the size of the clusters are not known in advance, and it introduces stochastic clustering. After a random element is selected, bids to join a cluster are calculated from all existing clusters. Fernandez [8] and Thebeau [23] each propose further refinements that improve on the basic algorithm that gives special treatment to vertical buses in the clustering process.

Some elements in software systems contain functions that are commonly used by a large number of other elements. These *vertical buses* are excluded from the clustering process and dependencies to these elements are assigned significantly lower clustered cost. A *bus threshold* parameter determines the minimal proportion of elements that must depend on a particular element for it to be considered a vertical bus. Setting this parameter too low would select too many classes as vertical buses, leaving no inter-cluster dependencies for the clustering algorithm to work with. Similarly, setting this parameter too high would select too few vertical buses. Through a trial-and-error process, we saw no large variations in clusters for bus threshold values greater 10%. Thus, we set the bus threshold parameter to 10% like [15].

Cluster membership is determined by a *dependency cost* measure that assigns a smaller cost to dependencies between elements within the same cluster than to elements in different clusters. It also assigns a smaller cost to dependencies on vertical buses than to dependencies between elements belonging to the same cluster:

$$DependencyCost(i,j) = \begin{cases} n^{\lambda} d_{ij} & \text{if } i \text{ and } j \text{ are in the same cluster} \\ N^{\lambda} d_{ij} & \text{if } i \text{ and } j \text{ are not in the same cluster} \\ d_{ij} & \text{if } j \text{ is a vertical bus} \end{cases} \tag{6}$$

where n is the size of the smallest cluster containing both i and j, and λ is a user-defined parameter. There is no "correct" choice for the value of λ, however, the literature [15] suggests to set $\lambda = 2$, as the number of potential interactions among elements increases as a power law with the number of elements in a cluster.

Aggregation of the dependency costs between all elements in a DSM results in the *clustered cost* measure as defined below:

$$ClusteredCost = \sum_{i=1}^{N} \sum_{j=1}^{N} DependencyCost(i, j) \tag{7}$$

Initially, each element is placed into its own cluster. As with [12,8,23], the algorithm selects a random element and accepts bids from other clusters for that element. If the element joins a cluster, the dependency costs for the elements in both its original and new clusters change, thus changing the overall clustered cost of the DSM. The bid of each cluster represents the decrease in the clustered cost if the element was to join that cluster. If the highest bid results in a decrease in the clustered cost, the DSM is rearranged and the element is added to the winning cluster. These steps are repeated for the next randomly chosen element. The iterative process ends when no further improvements in the clustered cost can be achieved for a given threshold number of iterations. Following [15], we set this threshold equal to N. The clustered cost algorithm is described by the pseudo code in Figure 2.

4.3 Relative Clustered Cost

The clustered cost measure is useful only when comparing software code bases of the same size or of similar size. MacCormack et al. [15] use this measure to compare the modularity of Mozilla with a similarly sized version of Linux. However, the clustered cost measure cannot be used to compare DSMs of different sizes.

The dependency cost is proportional to the number of dependencies in the DSM, and its value is always greater than or equal to 1. Because the clustered cost measure is an aggregate of all the dependency costs, a larger DSM is more likely to have a higher clustered cost. To develop a relative measure of clustered cost to compare code bases of different sizes, we define a new dependency cost function. We can normalize[1] the dependency cost function by dividing it by $N^{2\lambda}$:

$$RelativeDependencyCost(i, j) = DependencyCost(i, j)/N^{2\lambda} \tag{8}$$

The relative dependency cost function is as good a distance measure as the original dependency cost function because the ratios between costs in each of the three cases are unchanged. Using the relative dependency cost function in the bidding process results in the same output. As the clustered cost measure is an aggregate of all dependency costs, the newly defined dependency cost defines a measure that is relative to the size of the DSM. We define *relative clustered cost* as follows:

[1] The choice of this factor is informed by the logic that, in the extreme case, where each element is in a cluster of its own and all elements are interdependent, clustered cost will be $N \cdot N^{\lambda} = N^{2\lambda}$.

```
Input: design structure matrix
Output: clusters
place each element in its own cluster
repeat
    select an element i
    accept bids for element i from all clusters
    determine the highest bid
    if bid improves clustered cost then
        move element i to the winning cluster
    end
until no more improvement to clustered cost
```

Fig. 2. Clustered cost clustering algorithm

$$RelativeClusteredCost = \sum_{i=1}^{N} \sum_{j=1}^{N} RelativeDependencyCost(i, j) \tag{9}$$

Relative clustered cost is not proportional to the size of the DSM, and has values in the interval between 0 and 1. Regardless of DSM size, it has a minimum value of 0 if the DSM is without any dependencies, and a maximum value of 1 if each element is in a cluster of its own and all elements are interdependent. Therefore, the relative clustered cost measure can be used to compare DSMs of different sizes.

5 Evolving Modularity of Apache Tomcat

This section reports our findings from employing the method previously described in sections 3 and 4 to examine the evolving code base of an open source system.

Apache Tomcat is an open source application server developed and maintained by the Apache Foundation (`http://apache.org`). It is implemented in Java. Figure 3 provides a simplified view of the Tomcat architecture as having two major and distinct functional modules: the Tomcat server core (Tomcat-main), and Jasper, a separate module that processes Java Server Pages (according to the JSP specification). Tomcat-main and Jasper are linked only through the J2EE API.

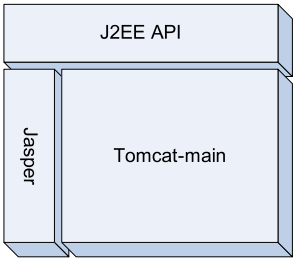

Fig. 3. Simplified view of the Tomcat architecture

Over the ten-year period between 1999 and 2008, four major versions of Apache Tomcat were released, shown here with the versions of key specifications:

- Apache Tomcat 3.x is based on the original implementations of the Servlet 2.2 and JSP 1.1 specifications donated by Sun Microsystems.
- Apache Tomcat 4.x implements the Servlet 2.3 and JSP 1.2 specifications and Catalina, a new servlet container based on a different architecture.
- Apache Tomcat 5.x implements the Servlet 2.4 and JSP 2.0 specifications.
- Apache Tomcat 6.x implements the Servlet 2.5 and JSP 2.1 specifications.

Since support for specific standards specifications is of primary importance to Tomcat users, major version numbers for Tomcat mirror the versions of the Servlet and JSP specifications that Tomcat supports. There were many minor versions within each major release. However, a change in major version numbers does not necessarily correspond to major changes in the structure of the code base.

Thus, when we selected the versions of Tomcat for our analysis, we identified significant architectural events in the evolution of the Tomcat code base, such as major changes to the architecture to improve performance, or the introduction of the Catalina servlet container. Table 1 describes the nine versions selected for analysis. Binary and source code of all Apache Tomcat releases is available for download from the project archives (`http://archive.apache.org/dist/tomcat`).

For each version, we examined the Tomcat-main and Jasper modules separately, and in combination. For each analysis, we computed the following metrics:

- Number of classes
- Number of dependencies (non-zero DSM elements)
- Propagation cost (defined in section 4.1)
- Number of vertical busses (fixed threshold of 10%)
- Number of clusters (post-clustering algorithm)
- Clustered cost (defined in section 4.2)
- Relative clustered cost (defined in section 4.3)

As noted in Section 4, the cost metrics are meaningful only in comparing two or more versions; they provide no significant information in isolation. Table 2 reports the number of classes, dependencies, and clusters. Figure 4 provides DSMs of Apache Tomcat 5.5.26 before and after clustering. The graphs in Figs. 5(a) to 5(f) plot propagation cost and relative clustered cost metrics for each version.

6 Findings

This section describes the results of our analysis. Summarizing the evolution of Tomcat, Table 2 shows that the number of classes has nearly tripled between version 3.0 (353 classes) and 6.0.16 (923 classes). This is clear evidence of the need for modularity measures that permit comparisons of code bases of different size.

To gain a better understanding of the relationship between the various modularity measures of the Apache Tomcat code base, we computed the correlation between propagation cost and relative clustered cost, and performed an F-test on the measures. The

Table 1. Tomcat versions selected for analysis

Version	Release date	Description
3.0	Dec. 1999	The initial Apache Tomcat version.
3.1.1	Dec. 2000	Final version of Tomcat 3.1.x introduced WAR support, servlet reloading and web server connectors for IIS and Netscape.
3.2.4	Nov. 2001	Final version of Tomcat 3.2.x introduced new features and major changes for improving performance and stability.
3.3.2	Apr. 2002	Latest production release of Tomcat 3.x finished the refactoring effort, and introduced a more modular design by allowing adding and removing modules that control the execution of servlet requests.
4.0.6	Oct. 2002	Final release of Tomcat 4.x introduced the Catalina servlet container.
4.1.37	Feb. 2006	Latest production release of Tomcat 4.1.x refactored Tomcat 4.x, and added new features: JMX, Coyote, and a rewritten JSP compiler.
5.0.30	Aug. 2004	Final release of Tomcat 5.0.x introduced performance improvements and a new standalone application deplorer.
5.5.26	Aug. 2007	Latest production release of Tomcat 5.5.x brought improvements in performance, stability, and total cost of ownership.
6.0.16	Jan. 2008	Latest production release of Tomcat 6.x introduced memory usage optimizations, advanced IO capabilities, and refactored clustering.

Table 2. Basic metrics for the selected Tomcat versions

	Combined			Jasper			Tomcat-main		
Version	**Classes**	**Depends.**	**Buses**	**Classes**	**Depends.**	**Buses**	**Classes**	**Depends.**	**Buses**
3.0	353	1110	4	97	453	14	256	655	2
3.1.1	412	1461	5	108	529	17	304	928	4
3.2.4	331	1461	8	115	572	17	216	861	9
3.3.2	444	1901	9	124	594	17	320	1276	9
4.0.6	464	1939	7	146	653	14	318	1286	9
4.1.37	520	2317	4	119	627	10	401	1690	8
5.0.30	651	2916	5	205	1141	11	446	1775	3
5.5.26	771	3318	3	226	1196	10	545	2122	3
6.0.16	923	4007	4	242	1241	8	681	2766	5

correlation coefficient of 0.43 suggests a medium correlation, and the F-test result of 1.03×10^{-12} suggests that the samples are similar in variance [5].

Initially, we expected the modularity of Tomcat to increase throughout the evolution of the product. The rationale for this expectation was that as a system evolves, its structure would be continually examined by developers. Specifically, we expected that architectural improvements would also lead to increased modularity. For example, when Tomcat 4.x introduced a new implementation of the servlet container based on a new architecture (Catalina), we expected the new architecture to be more modular because it was built from the ground up for flexibility and performance.

However, from Figs. 5(a) and 5(b) we observe that the propagation costs for Tomcat 3.3.2 and 4.0.6 are 9.6% and 14.6%, respectively, and the relative clustered costs are

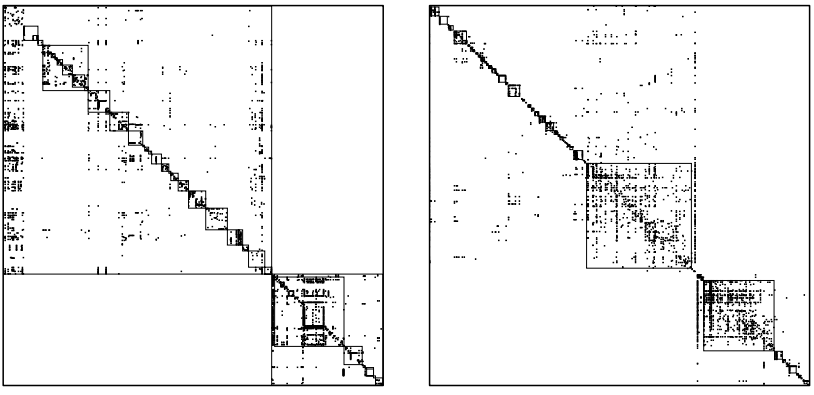

Fig. 4. DSMs for Apache Tomcat 5.5.26 before and after clustering

0.0031 and 0.0035 (as highlighted by the dashed circles). Both metrics suggest that version 4.0.6 is actually *less* modular than version 3.3.2 – the opposite of what we expected to find. Similarly, from Figs. 5(c) and 5(d), we see a spike in both propagation cost and relative clustered cost for the Jasper subsystem between versions 4.0.6 and 4.1.37. With the introduction of version 4.1.37, Jasper became *less* modular.[2]

A closer examination of the events (see also Table 1) surrounding these spikes in propagation cost and relative clustered cost suggests that each *decrease* in modularity was precipitated by a major architectural or implementation change. For all other releases, whether major versions or incremental releases, the code became *increasingly* more modular. Interestingly, each spike is immediately followed by an *increase* in modularity. In fact, in each case, the increase in modularity of the consecutive version more than compensated for the previous decrease.

Our data is not conclusive on *why* this pattern occurred, but we cautiously put forward a plausible, albeit tentative explanation. Once new functionality is initially deployed and working, focus shifts. Developers revisit the design and perform refactoring and cleanup activities (consisting of changes to the structure, but not to the behavior of the system). Increased understanding and experience gained through the original implementation permits developers to more easily restructure the existing code into a more modular design. The result is a significant increase in modularity that compensates for the original decrease in the previous version.

To capture these observations, we propose three propositions that can guide future research on the evolution of modularity of (open source) software systems:

Proposition 1. *Major architectural and implementation changes cause the modularity of a software system to decrease at first.*

Proposition 2. *Major changes are followed by periods of refactoring and cleanup activities, which cause the modularity of the software system to increase again.*

[2] A third such event occurred with the introduction of version 3.2.4, which decreased modularity from version 3.1.1. However, the interpretation the this event is similar to other two events.

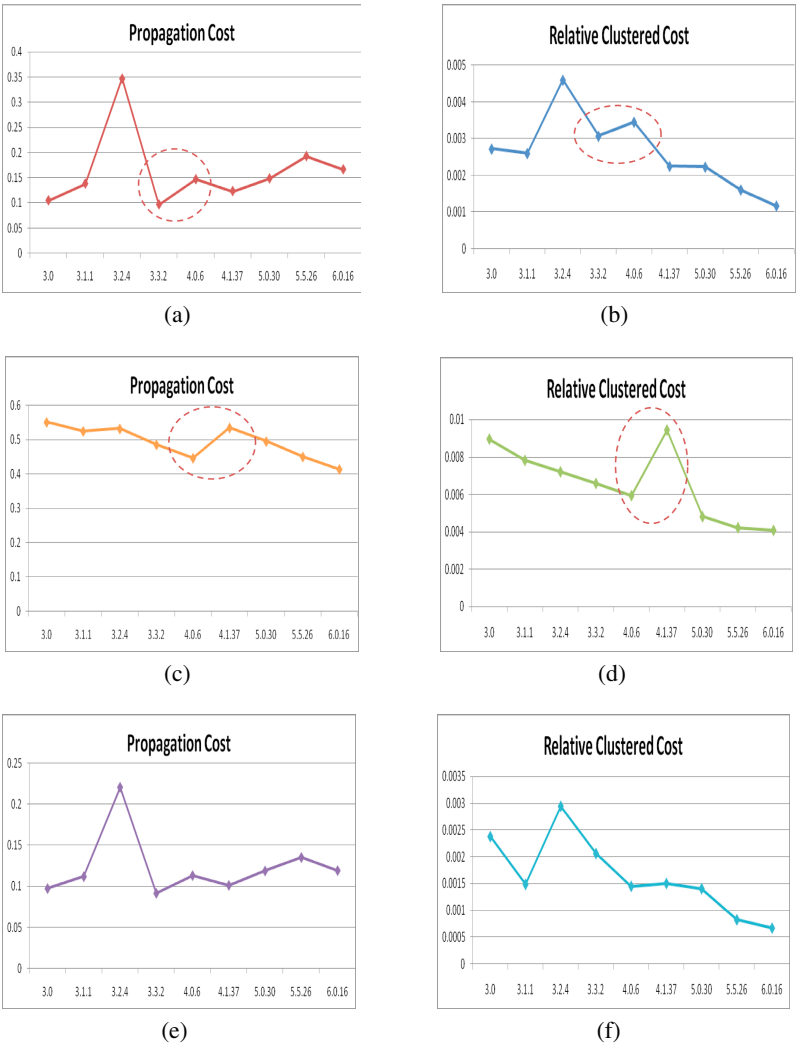

Fig. 5. (a) Propagation cost Tomcat-main. (b) Relative clustered-cost Tomcat-main. (c) Propagation cost Jasper. (d) Relative clustered-cost Jasper. (e) Propagation cost combined. (f) Relative clustered-cost combined.

Proposition 3. *The increase in modularity as a result of refactoring and cleanup activities more than offsets the decrease in modularity due to a major change.*

7 Conclusion

In this paper we examined the evolution of a large open source system, and proposed an improved modularity metric that allows the comparison of code bases of different size. The method makes improvements over existing approaches based on Design Structure

Matrices [15]. It also presents an alternative as well as a complement to other approaches to measuring evolving software complexity [20].

We provided initial evidence that as a large software system evolves, major architectural changes, at first, lead to an increase in modularity, but are followed by refactorings and cleanup activities which lead to a subsequent increase in modularity. We formulated propositions around these observations that can guide future research on the evolution of software system modularity.

References

1. Baldwin, C.Y., Clark, K.B.: Design Rules: The Power of Modularity. MIT Press, Cambridge (2000)
2. Baldwin, C.Y., Clark, K.B.: The architecture of participation: does code architecture mitigate free riding in the open source development model. Management Science 52(7), 1116–1127 (2006)
3. Browning, T.R.: Applying the design structure matrix to system decomposition and integration problems: A review and new directions. IEEE Transactions on Engineering Management 48(3), 292–306 (2001)
4. Christensen, C.M., Michael, R., Verlinden, M.: Skate to where the money will be. Harvard Business Review, 72–81 (November 2001)
5. Cohen, J.: Statistical power analysis for the behavioral sciences, 2nd edn. Lawrence Erlbaum, Hillsdale (1988)
6. Eppinger, S.D.: Model-based approaches to managing concurrent engineering. Journal of Engineering Design 2(4), 238–290 (1991)
7. Eppinger, S.D., Whitney, D.E., Smith, R.P., Gebala, D.A.: A model-based method for organizing tasks in product development. Research in Engineering Design 6(1), 1–13 (1994)
8. Fernandez, C.I.G.: Integration analysis of product architecture to support effective team colocation. Master's thesis, Sloan School of Management, Cambridge (1998)
9. Garud, R., Kumaraswamy, A., Langlois, R.N.: Managing in a Modular Age: Architectures, Networks, and Organizations. Blackwell Publishing, Malden (2002)
10. Henderson, R.M., Clark, K.B.: Architectural innovation: the reconfiguration of existing product technologies and the failure of established firms. Administrative Science Quarterly 35, 9–30 (1990)
11. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence, 2nd edn. University of Michigan Press, Ann Arbor (1992)
12. Idicula, J.: Planning for concurrent engineering. Research report, Gintic Institute, Singapore (1995)
13. LaMantia, M.J., Cai, Y., MacCormack, A., Rusnak, J.: Analyzing the evolution of large-scale software systems using design structure matrices and design rule theory: two exploratory cases. In: Working IEEE/IFIP Conference on Software Architecture (WICSA 2008), pp. 18–22. IEEE, Washington (2008)
14. Langlois, R.L., Robertson, P.L.: Networks and innovation in a modular system: lessons from the microcomputer and stereo components industries. Research Policy 21, 297–313 (1992)
15. MacCormack, A., Rusnak, J., Baldwin, C.Y.: Exploring the structure of complex software designs: an empirical study of open source and proprietary code. Management Science 52(7), 1015–1030 (2006)
16. O'Reilly, T.: The Open Source Paradigm Shift. In: DiBona, C., Stone, M., Cooper, D.: Open Sources 2.0: The Continuing Evolution, 253–272 (2005)

17. O'Reilly, T.: What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software (2005), http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html
18. Parnas, D.L.: On the criteria To be used in decomposing systems into modules. Communications of the ACM 15, 1053–1058 (1972)
19. Sanchez, R., Mahoney, J.T.: Modularity, flexibility, and knowledge management in product and organization design. Strategic Management Journal 17, 63–76 (1996)
20. Sangwan, R.: Structural epochs in the complexity of software over time. IEEE Computer, 66–73 (July 2008)
21. Simon, H.A.: The Sciences of the Artificial, 3rd edn. MIT Press, Cambridge (1996)
22. Steward, D.V.: The design structure system – a method for managing the design of complex systems. IEEE Transactions on Engineering Management 28(3), 71–74 (1981)
23. Thebeau, R.E.: Knowledge management of system interfaces and interactions for product development processes. Master's thesis. MIT, Cambridge (2001)
24. Ulrich, K.: The role of product architecture in the manufacturing firm. Research Policy 24(3), 419–440 (1995)