The Inevitable Stability of Software Change

Rajesh Vasa, Jean-Guy Schneider
Faculty of Information & Communication Technologies
Swinburne University of Technology
P.O. Box 218, Hawthorn, VIC 3122, AUSTRALIA
{rvasa,jschneider}@swin.edu.au

Oscar Nierstrasz
Institute of Computer Science
University of Bern
Bern, CH-3012, SWITZERLAND
oscar@iam.unibe.ch

In Proceedings of the 23rd International Conference on Software Maintenance (ICSM 2007), IEEE Computer Society Press, Paris, France, October 2007, pp. 413. DOI: 10.1109/ICSM.2007.4362613

Abstract

Real software systems change and become more complex over time. But which parts change and which parts remain stable? Common wisdom, for example, states that in a well-designed object-oriented system, the more popular *a class is, the less likely it is to change from one version to the next, since changes to this class are likely to impact its clients. We have studied consecutive releases of several public domain, object-oriented software systems and analyzed a number of measures indicative of size, popularity, and complexity of classes and interfaces. As it turns out, the distributions of these measures are remarkably stable as an application evolves. The distribution of class size and complexity retains its shape over time. Relatively little code is modified over time. Classes that tend to be modified, however, are also the more popular ones, that is, those with greater Fan-In. In general, the more "complex" a class or interface becomes, the* more *likely it is to change from one version to the next.*

1. Introduction

It is well-established that software systems *must* change and become more complex over time as they are used in practice [15]. However what is less well-understood is how change and complexity are distributed over time, particularly in object-oriented systems.

In order to effectively manage the evolution of complex software systems, we would like to know where we can expect growth and change to occur. To avoid triggering a series of further changes and bug fixes, for example, it seems wise to make new classes depend on stable, reliable parts of the system, rather than on those that are constantly changing. As a consequence, it seems logical that new code should depend on existing, proven parts of the system.

We therefore ask the following questions: How do size, complexity and "popularity" (*i.e.*, Fan-In) of classes evolve over time? Which classes tend to grow and become more complex over time? Which classes are most likely to change?

We have analyzed a number of open source applications that have evolved over at least 15 releases during a period of at least 24 months. For each of these applications we have collected various established size and complexity measures, and analyzed how they have evolved over time. In particular, we have studied how these measures vary for all classes, for modified classed and for newly created classes.

The key results of our studies show that:

1. We have observed that relatively little code is changed in a software project as it evolves. Code is even less likely to be removed than changed.
2. The profile of size and complexity measures for any given application rapidly stabilizes and *does not change over time*. As a consequence, barring a major structural change to an application, average class size or complexity will not change in the long term.
3. Fan-In for modified classes is significantly higher than the average for all classes. In other words, *popular classes are more likely to change*.
4. Fan-In for new classes is generally lower than the average, but tends towards the typical profile over time. This suggest that new classes start out as clients, rather than suppliers, but become more popular over time.
5. Classes with high Branch Count tend to be modified more than those with low Branch Count. That is, large and complex classes tend to be changed more than small and simple classes.

Our results suggest that, in the absence of a major architectural shift or a rewrite of the code base, nothing will

perturb the average size and complexity of classes. Furthermore, efforts to base new code on stable components will inevitably make those components less stable.

The rest of this paper is organized as follows: in Section 2 we provide an overview of our experimental method, and we justify the selection of the case studies. Section 3 presents the results of our studies. In Section 4 we suggest possible interpretations and consequences of our observations. Section 5 provides a brief overview of related work. We conclude in Section 6 with some remarks about future work.

2. Experimental Method

In this section we first present our criteria for selecting the systems to be studied. Next we briefly discuss the selected measures, followed by a review of the notion of a *type dependency graph* used to define in particular Fan-In and Fan-Out. Finally, we briefly describe the means by which measurements were performed.

2.1. Input Data Set Selection

For the purpose of our study, we have restricted our input to free open-source software developed using the Java programming language.

The rationale for using open source software is the availability of the systems, access to change logs (such as developer release notes or change logs), as well as non-restrictive licensing that gives us free access to both source and object code. The choice of the programming language was influenced by our interest in understanding software developed using Java as it is used in a variety of application domains, as well as by the availability of a suitable infrastructure to implement a metrics tool.

In order to identify suitable systems for our study, we define a number of selection criteria that ensure the systems have a sufficiently long development history to provide meaningful data. Our selection criteria are as follows:

1. At least *10 releases* of the system must be available. Only complete builds are considered to be releases. Branches and releases not derived from the main system tree are ignored.
2. The system has been in active development and in use for at least *12 months*.
3. The system comprises at least *200 classes*[1] at some point in its lifetime and consists of no less than *100 classes* when analyzed in order to eliminate trivial systems.
4. *Change logs* exist that document modifications made to the software. This data provides the invaluable information to understand major changes made to a given system.

1 To improve readability we will refer to "classes" when we mean "classes or interfaces". We will only refer to "types" in the context of the formal measures.

Using these selection criteria, we have been able to identify over 100 candidate systems. However, due to time and resource constraints, we have selected 12 representative systems (cf. Table 1) for this study, each having at least 15 releases over a time span of more than 24 months, with a total of 292 releases analyzed.

For each of systems under investigation, we use a *Release Sequence Number* (RSN) [5] as the pseudo-time measure. RSNs are universally applicable and independent of any release numbering schedule and/or scheme. An RSN is a sequential number allocated based on release dates, where the first version is 1 and then each subsequent version increases by one.

2.2. Software Measures

Software systems exhibit two broad quantitative aspects that are captured by a wide range of software measures [6]: *size* and *complexity*. These measures provide an objective view for both the process being used to create the software system and its internal structure. By collecting and analyzing these measures over time, we can distill a temporal dimension, which is capable of revealing new, valuable information such as the rate of size growth [14, 16] and evolutionary jumps in the complexity of a software system [10], respectively. Moreover, previous work shows that *evolution measures* can be used to detect architectural shifts automatically [21, 23].

We have extracted 25 different measures for each class in each system analyzed. The measures that turn out to be particularly interesting are Fan-In, Fan-Out and Branch Count (*i.e.*, the number of *branch* instructions in the Java Bytecode). We measure Branch Count instead of McCabe cyclomatic complexity [18] since we are analyzing changes at a class-level, and not at a method-level. Some of the other measures we extracted are Method Count, Field Count, Load Instruction Count (*i.e.*, the number of *load* instructions), and Store Instruction Count (*i.e.*, the number of *store* instructions). We then used a type dependency graph (see Section 2.3) to compute the Fan-In for each class.

Only if *all measures* under consideration are equal from one version of a system to the next for a given class do we consider this class as being *unchanged*. Although it is possible that some distinctly smaller subset of measures would suffice to assess whether a class has changed without significant loss of precision, we adopted the more conservative approach for our analysis and used the full set.

Name	Releases	Time Span	Initial Size	Current Size	Description
Acegi	17	32 mo.	135	368	Role-based security framework
Active MQ	26	27 mo.	205	2295	Message queue framework
Axis	23	65 mo.	166	636	Apache SOAP server
Azureus	21	41 mo.	103	2526	Bittorent Client
Castor	27	48 mo.	483	691	Data binding framework
Findbugs	15	34 mo.	223	567	Automated bug finding application
Hibernate	46	73 mo.	120	1055	Object-relational mapping framework
Saxon	15	64 mo.	459	786	XML transforming library
Spring	41	43 mo.	386	1570	Light-weight container
Velocity	17	72 mo.	230	214	Template engine
Webwork	19	36 mo.	75	473	Web application framework
Wicket	25	30 mo.	181	631	Web application framework

Table 1. Systems under analysis for this study. Size is the number of classes and interfaces.

2.3. Type Dependency Graphs

We capture Fan-In and Fan-Out by defining a *type dependency graph* [21] as an ordered pair $G^T = (V, E)$, where V is a finite, nonempty set of *types* (*i.e.*, classes and interfaces) and E is a finite, possibly empty, set of *directed links* between types (*i.e.*, $E \subseteq V \times V$). $N = |V|$ denotes the number of nodes and $L = |E|$ denotes the total number of directed links of a given type dependency graph. Note that some other studies of software graphs (*e.g.*, [20]) have treated dependencies as *undirected*.

To capture both Fan-In and Fan-Out of a given type, represented by a node $n \in V$, we use $l_{in}(n)$ to denote the *in-degree* and $l_{out}(n)$ to denote the *out-degree* of node n, where:

$$\begin{aligned} l_{in}(n) &\stackrel{\text{def}}{=} |\{\langle n_i, n\rangle \in E\}| \\ l_{out}(n) &\stackrel{\text{def}}{=} |\{\langle n, n_j\rangle \in E\}| \end{aligned}$$

The in-degree is a measure of the "popularity" of node n in the graph G^T, whereas the out-degree is node n's "usage" of other types in the graph G^T [19].

2.4. Extracting Measures

In order to perform the analysis, we have developed a *metrics extraction* tool [23], which analyzes Java Bytecode and extracts data to capture the degree of change of a system with respect to its size and complexity. This tool takes as input the *core JAR files* for each release of a system being investigated and generates the desired metric data.

Measures needed for our research efforts were extracted by processing the raw Java Bytecode. This approach allows us to avoid running a (sometimes quite complex) *build process* for each release under investigation and we only analyze "code" that has actually been compiled. The Java Bytecode generally reveals almost as much about a system as its source code (unless a Bytecode obfuscator is used), and only some subtle changes to a software system cannot be detected using this approach (*e.g.*, renaming of local variables).

Our extraction tool uses `ASM`, a Java Bytecode manipulation framework,[2] to collect static dependency information from the classes contained within the core JARs. For each class, the set of dependencies are extracted and recorded. However, the following types are ignored, as they do not add any specific value to the analysis process [23]:

1. All primitive Java types such as `int`,
2. The class `java.lang.String`,
3. The root class `java.lang.Object`, and
4. `self`-references (*i.e.*, all occurrences of `this`).

2.5. Analysis method

In this paper, we have focused our attention on the key complexity measures of Fan-In, Fan-Out and Branch Count. In order to understand if these measures tend to exhibit an inherent structural pattern, we plot them as a relative frequency distribution for each version under study and compute the similarity between successive versions using the Bhattacharya measure [1]. This similarity measure is a value between 0 and 1; the closer to 1 the value is, the more similar the distributions of successive versions are to each other. Statistical outliers in the similarity measure highlight structural changes in the frequency distributions. Similarly, Fan-Out distribution changes highlight global changes in the way classes are defined. Branch Count distribution changes highlight a change in the cyclomatic complexity of the corresponding classes.

In order to understand the way software evolves, we need to look at the various measures that new classes as well as

2 `asm.objectweb.org`.

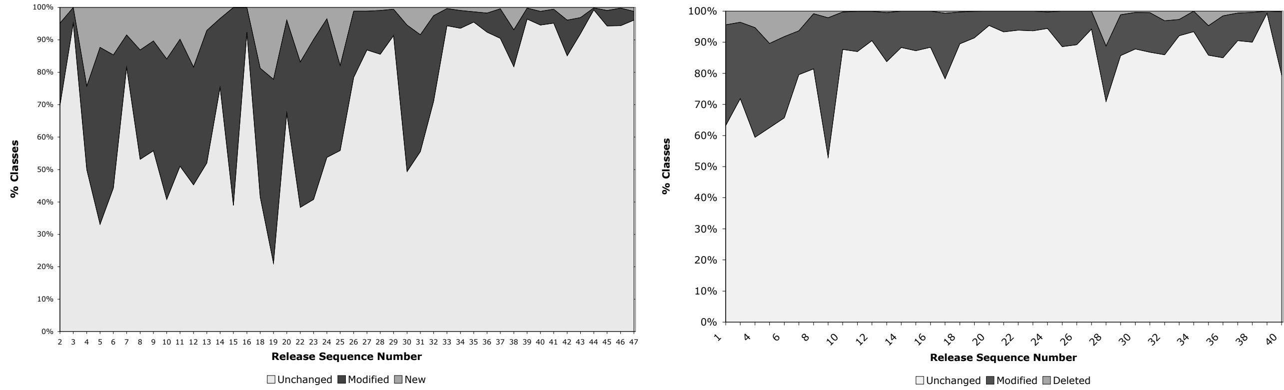

Figure 1. Evolution of types in Hibernate (left) and Spring (right)

modified classes in a system tend to posses. We have focused our attention on classes that can be considered popular and hence should remain stable in order to minimize the ripple impact of any change. We define a class to be popular if it has a *Fan-In of 5 or more*. The value 5 was selected since, on average, only 20% of the classes (applying the Pareto principle) in a system have this level of Fan-In.

3. Observations

We now summarize our observations from analyzing the 12 systems listed in Table 1. First we analyze the evolution of the rate of change in each system. We then draw some observations concerning the evolution of the *profile* of size and complexity over time. Finally we consider the evolution of Fan-In, Fan-Out, Branch Count and Method Count for changed and new classes, respectively.

3.1. Probability of Change

What is the likelihood that a class will change from a given version to the next? Does this probability change over time? Is it project-specific?

As a system evolves incrementally, software entities are added, removed, adapted and otherwise modified. To assess the likelihood of a class changing from version v to version $v+1$, we gather the following statistics:

u_v percentage of classes that are unchanged
c_v percentage of classes that are changed
d_v percentage of classes that are removed
a_v percentage of classes that are added

In our input data set where we studied over 275 unique changed versions across 12 systems, we determined that for any given version v, the following property holds in 85% of the versions:

$$u_v > c_v > d_v$$

and the following property holds for 80% of versions:

$$u_v > c_v > a_v$$

When we look ahead one version, on average across all systems that we studied, we observe that 75% of the classes are unchanged, 20% are modified and 5% are removed. When we look back one version to detect new classes, on average we note that 72% of the classes are unchanged, 20% are modified and around 8% are new classes. Figure 1 highlights our observations for two of the systems under investigation.

3.2. Evolution of Complexity Measures

How do measures of size and complexity evolve over time? Do they tend to grow with time? Are the tendencies project-specific?

Figure 2 presents the boundaries of the histograms based on the minimum and maximum values of Fan-In and Branch Count attained across all versions of the Spring[3] case study. One can clearly see that the relative frequency distributions of these measures have a distinct profile that is bounded in a small range.

This same phenomenon was observed across all projects and for all size and complexity measures that we collected. The profile of the relative frequency distribution of the complexity measures Fan-In, Fan-Out and Branch Count holds its broad shape across the evolutionary history of any given software system. So, if 20% of the classes in a system have a Fan-In of 5 or greater in Version 1, the probability that this value will change by more than a few percent is very low over the evolutionary history of the product. This holds for all of the various values of the Fan-In measure in the histogram.

3 `www.springframework.org`

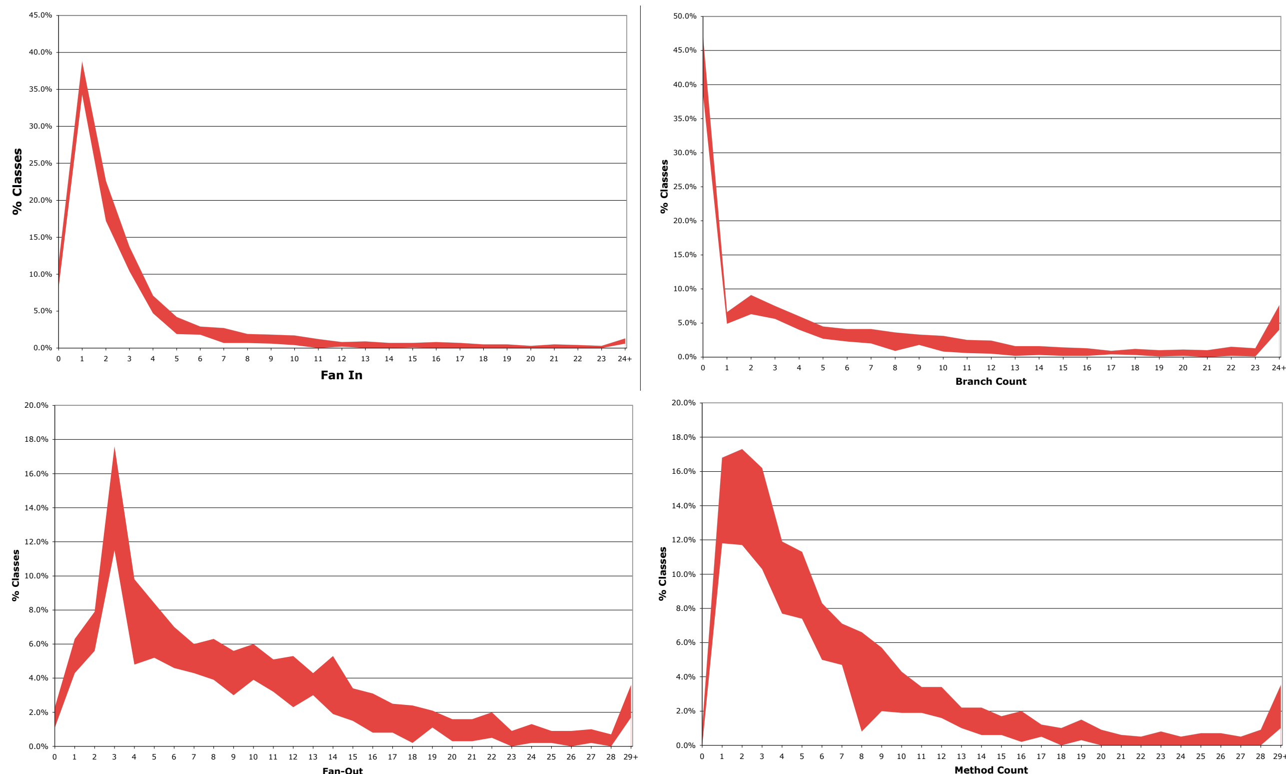

Figure 2. Spring evolution profiles showing the upper and lower boundaries on the relative frequency distributions. All metric values during the entire evolution fall within the boundaries shown

The only exceptions to this rule seem to coincide with structural shifts from one major release to another. In Hibernate[4], one of the systems in our study, we noticed the profile of the Fan-In, Fan-Out, and Branch Count distributions has shifted significantly, twice during its evolutionary history. We detect the significance by observing the Bhattacharya measure of similarity. In most cases, this measure changes by less than 0.5% between consecutive releases. However, for Hibernate, we detected changes in the Bhattacharya measure to be between 1.5% and 2.5%, corresponding to known changes between major releases.

When we computed the amount of change, we noticed that the probability of change in the distribution profile is slightly higher in earlier versions than later. So, as a software system ages, its distribution profile tends to stabilize and becomes highly predictable. Figure 3 shows profile stabilizing over time; this chart plots the Bhattacharya similarity metric that measures the similarity between consecutive frequency histograms for Fan-In, Fan-Out, Branch Count and Method Count measures for the Spring framework.

4 www.hibernate.org

3.3. Fan-In of Modified Classes

What characterizes the classes that do change? In particular, is it true that the most popular classes are indeed the most stable ones?

In Figure 3.5 we see histograms for both Spring and Azureus[5] showing the proportion of classes that have a Fan-In value greater than 4 over the entire evolutionary history. We can see that the proportion of modified classes with high Fan-In is consistently greater than the proportion of all classes. (This is in spite of the fact that the overall profile of Fan-In is stable over time.) The same observation holds across all 12 case studies.

Developers would approach a class with high Fan-In with care due to the associated risk of impacting other classes that rely on the services provided. However, our observations show that on average, classes with a higher Fan-In tend to be modified more than those with lower Fan-In.

In order to ensure that we are not observing a size-related effect, *i.e.*, larger classes tend to be modified more because they have more overall code volume, we have run a corre-

5 azureus.sourceforge.net

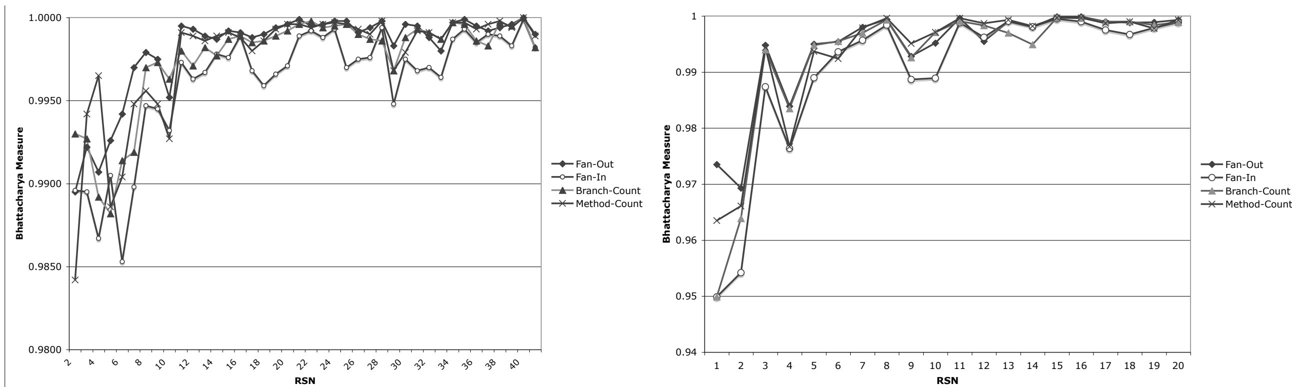

Figure 3. Evolution stabilization for Spring (left) and Azureus (right)

lation measure between Fan-In count and the overall raw size of the class. The correlation coefficient measure is between -0.2 and $+0.2$ across all of the versions in our data set, which suggests that there is little evidence that size is directly correlated with Fan-In.

3.4. Fan-In of New Classes

What characterizes new classes? How do new classes compare with existing and modified classes?

We have seen that the profile of class size is stable over time. As a consequence we know that system growth is mainly due to the addition of new classes, rather than growth within existing classes. The number of classes in all of the systems under study has increased over their evolutionary history.

We observed that the Fan-In profile of the new classes is very different from the profile of existing code. New classes tend to start with a lower Fan-In, and as they are modified over time move towards the overall trend. If we compare the proportion of new classes with Fan-In greater than 4 (see Figure 3.5), with that of all classes, we can clearly see that this proportion is consistently lower than the norm (Gaps in the Azureus Fan-In evolution are due to the absence of new classes in selected revisions).

New classes, therefore, tend to start out with low popularity. But we also know that the profile of Fan-In distribution is stable over time, so this suggests that, as they evolve, Fan-In of the new classes will tend towards the typical profile.

3.5. Branch Count of modified classes

How does the size or complexity of a class impact the likelihood that it will be modified?

When a class is modified, there is a certain probability that the branching statements are altered. To appreciate if the number of branching instructions has an impact on the probability of change, we observed the evolution of the Branch Count of the classes that have been changed. On average, classes with a higher Branch Count are modified more than those with lower Branch Count. However, larger classes strongly correlate (coefficient is on average over 0.9) with higher Branch Count. We also observe that Branch Count does not strongly correlate with Fan-In (coefficient is on average around 0.2), which suggests that classes with complex code need not be more popular than simpler classes.

Again, to ensure that any size related effects are eliminated, we have normalized the Branch Count for each class based on its size and re-computed our profile. Even after normalization, we have observed that classes with higher number of branch instructions will tend to be modified more than those with fewer branch instructions.

4. Discussion

In the previous section, we summarized our observations from analyzing popular open-source software systems. We discuss these findings and offer possible interpretations of the results.

Probability of Change: We have observed that relatively little code is changed in a software project as it evolves. This reflects not only the small number of classes that change, but also the small amount of change within modified classes.

Our data also reveals that code is even less likely to be removed than changed. This suggests that developers tend to resist making substantial changes to the existing code once it has been released. We can conclude that any code that is released in early versions of a software system is likely to stay.

Evolution of Complexity Measures: Our observations show that the distribution profile of complexity mea-

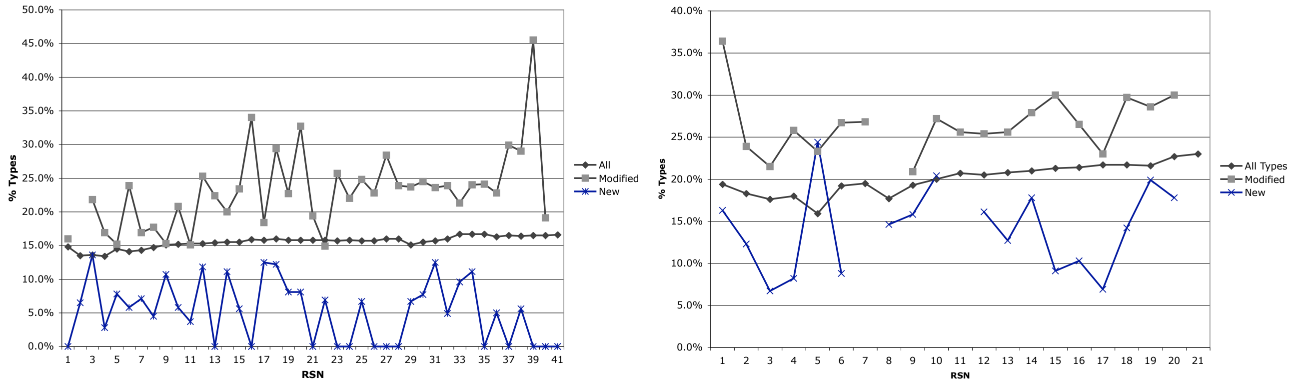

Figure 4. Spring (left) and Azureus (right) Fan-In evolution. Proportion of modified classes with high Fan-In is greater than that of new or all classes

sures does not change over time. On the other hand, we can clearly see that the systems we analyzed grew over time, but existing classes did not grow. Hence, we can conclude that system growth is by *addition*, and not by extension (of existing classes).

We can also see that Lehman's law of increasing complexity [15] does not apply to *individual* classes. Average size and complexity of individual classes rapidly stabilizes over time. Our data shows that most classes do *not* become more complex over time. This indicates that system complexity is mainly due to growth in general (new classes), not growth or increasing complexity of the individual parts.

The stability of the distribution profile of complexity measures indicates that a system generally keeps its character over time. We see that the size and complexity profile only changes when there is an architectural shift. This suggests that the underlying architecture plays a major role in determining the relative distribution of size and complexity for all time.

Fan-In of Modified Classes: Classes that are modified tend to have high Fan-In. This is suggestive of Lehman and Belady's first Law of Software Evolution which states that systems that are used will undergo continuing change [15, 16]. In this case we see that classes that are heavily used, *i.e.*, that have high Fan-In, are more likely to undergo change.

This observation does not square well, however, with Martin's Stable Dependencies Principle [17] which states that: "The dependencies between packages in a design should be in the direction of the stability of the packages. A package should only depend upon packages that are more stable that it is." On the surface, the principle appears sound: to improve the overall stability of our system, we should make new things depend on stable and mature components. Unfortunately, our new interpretation of Lehman's Law of Continuing Change suggests that the very fact of depending on a stable component will make it less stable.

This leads us to question Martin's popular Instability measure, which essentially considers a package to be maximally stable when it has only incoming dependencies and no outgoing dependencies, and maximally unstable when it has only outgoing dependencies. Martin's reasoning is based on "stable" meaning "not easily moved" [17]. We feel that this measure confuses *stable* with *inflexible* ("unwilling to change or compromise"[6]). A more usual definition of stable is "not likely to change or fail" (*op. cit.*). In this case what Martin calls stable we consider to be *unstable*, and vice versa.

Note that Martin is interested in *package* dependencies, which we have not considered in this present work. A serious investigation of the implications of our work on Martin's Instability measure in the context of package dependencies remains to be done.

We have also not investigated *why* changing classes have higher than normal Fan-In. We speculate that the introduction of new clients creates the need for adaptations. Other possibilities are that (i) new clients introduce new requirements, but that would suggest new growth in existing classes, which we did not find, or (ii) new clients exercise existing classes in new ways, thus uncovering previous unknown defects. Further work is needed to discover which, if any of these hypotheses is correct.

Fan-In of New Classes: We have seen that, since the profile of size and complexity remains constant, it cannot be that growth mainly occurs in existing classes, but rather in the creation of new classes. But where does this growth occur — at the top or at the bottom of the system?

6 Oxford American Dictionary, 2005

Since new classes have lower than average Fan-In but normal Fan-Out, it seems clear that growth is on top of existing classes. It is highly unusual for a new class to have high Fan-In, so there must be little growth below classes of the existing system.

However, we have also seen that the overall profile of Fan-In over time is constant, so even new classes will eventually become part of the infrastructure, and tend towards average levels of Fan-In.

Since open-source projects are known to be developed in an incremental and iterative fashion, our observations are consistent with the notion that these systems are built bottom-up, rather than top-down.

Branch Count of modified classes: We have observed that classes that are modified tend to have a higher Branch Count than the ones that remain unchanged. Why is this the case?

Earlier research [24] suggests that a complex class will tend to have more defects. Our observation that complex classes attract a higher proportion of modification is consistent with the fact that complex classes tend to have more defects and, therefore, will tend to undergo more modifications to correct the defects.

Our observations are incomplete, however, since we do not have access to defect data to allow us to state that changes to complex classes are principally concerned with correcting defects. Furthermore, it is reported that corrective changes account to only 21% of changes [2], so we cannot conclude that defects are the main reason for change.

A class that changes is likely to have higher Fan-In and higher Branch Count. Simply put, *complex classes have a higher probability of change*.

5. Related Work

Lehman and Belady pioneered the study of evolution of large-scale procedural systems, and established the well-known "laws of software evolution" [15]. Until recently, however, there have been few empirical studies focusing on a *micro-level* in order to gain insight into where and how present-day object-oriented systems evolve.

We have previously presented empirical evidence that cyclomatic complexity of classes essentially does not vary over time and that, in general, more than 50% of all methods have a cyclomatic complexity of 1 [22]. We have presented a simple growth estimation model built on top of an observed *power-scaling relationship* between the total number of nodes and the total number of links in a software dependency graph [23].

We have also studied typical growth patterns in open-source software and shown that although software grows over time, the structure and scope of growth is in general not erratic, but is predictable using a power-scaling relationship [21]. For example, we have observed that the percentage of derived classes in a given software system does not change significantly over time. Hence, if our estimation models fail to accurately predict the growth and the associated changes in a software system, this means that significant architectural shifts have occurred that require special analysis and documentation.

Gîrba *et al.* have tested the hypothesis that classes that have changed in the past are likely to change in the future [8]. The reliability of this measure of "yesterday's weather" seems to vary according to the "climate" of a software project. Gîrba *et al.* have also studied the relationship between change and developers [9]. Rather than predicting change, the goal here is to understand which developers are most knowledgeable about different parts of an evolving system.

Similar to our work, Capiluppi *et al.* have analyzed the evolution history of a number of open source systems [3, 4]. However, their studies mainly focused at a macro-level, in particular on relative changes in the code size and on complexity at a module level [4] as well as the influence of the number of developers on the release frequency [3].

Lanza and Ducasse have introduced the Evolution Matrix [13] as a means to visualize the evolution of object-oriented software. Here the emphasis is on detecting patterns of change, rather than to establish which parts of the system are likely to change. Gall *et al.* have analyzed the history of changes in order to detect hidden dependencies between modules [7]. Grosser *et al.* have applied case-based reasoning to versions of object-oriented software systems to predict the preservation of class interfaces [11]. Zimmerman *et al.* analyzed relationships between changes to predict when certain classes are changed which other classes are also likely to change [25]. Finally, Hassan and Holt have also analyzed numerous open source projects and concluded that historical co-change is a better predictor of change propagation than structural dependencies [12]. None of these approaches, however, are directly applicable to the question of understanding how and where contemporary object-oriented software systems change over time.

6. Conclusions

Although the long-term effects of evolution on software systems have been studied now for over three decades, there has been little research into understanding how change is distributed over the parts of software systems. We have analyzed 12 open-source Java systems that have evolved over at least 15 releases and over a period of at least 2 years to evaluate which size and complexity measures are indicative of high rates of change.

Our study shows that most of the code base in a software system is unchanged as it evolves over time. On average around 20% of the classes are modified and around 8%

of the classes are newly added. Furthermore, the probability that a class is removed is very low.

The distribution of all studied measures tends to retain a remarkably uniform profile over time. So, for example, the relative percentage of classes with high Fan-In or low Fan-In will normally not change during the course of a project, except at points where a major restructuring or rewriting of the code base takes place.

We also show that the classes with the highest rates of change tend to be the most popular ones, *i.e.*, those with high Fan-In. On the other hand, new classes tend to start out with low Fan-In. Nevertheless, as new classes evolve, their profile tends towards the norm for the code base. Growth tends to occur in new classes, not old ones, which is consistent with the observation that the distribution of size and complexity measures remains constant over time.

Common wisdom states that one should build new software on top of stable components, that is, on mature classes with low rates of change. This leads, however, to the paradox that by relying on stable components, we increase their popularity, and thus cause them to become less stable. On the one hand, this suggests that Lehman and Belady's Laws of Software Evolution also apply to some degree at a micro scale: a class that is used will undergo continuing change or become progressively less useful. On the other hand, since the profile of size and complexity measures stays constant, we cannot conclude that the classes of an evolving software system necessarily become more complex on average. Complexity at the system level is not a consequence of complexity of the parts, but rather of the sheer size of the system as a whole.

This work opens up a series of further questions which we plan to explore. In our current study we have noted change, but not computed the amount of change. We are investigating the use of a distance measure to indicate how much change a class undergoes. Our early results suggest that most classes do not undergo a significant amount of change; we intend to collect detailed data and report our findings.

Acknowledgments: The authors would like to thank Philip Branch and Tudor Gîrba for their detailed comments on drafts of this paper.

Oscar Nierstrasz gratefully acknowledges financial support of the Swinburne University of Technology Visiting Professor Award Scheme. He also gratefully acknowledges the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008).

References

[1] F. Aherne, N. Thacker, and P. Rockett. The bhattacharyya metric as an absolute similarity measure for frequency coded data. *Kybernetika*, 34(4):363–368, 1998.

[2] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 73–87, New York, NY, USA, 2000. ACM Press.

[3] A. Capiluppi. Models for the evolution of OS projects. In *Proceedings International Conference on Software Maintenance (ICSM 2003)*, pages 65–74, Los Alamitos CA, 2003. IEEE Computer Society Press.

[4] A. Capiluppi, M. Morisio, and P. Lago. Evolution of understandability in OSS projects. In *Proceedings 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 58–66, Los Alamitos CA, 2004. IEEE Computer Society Press.

[5] D. Cox and P. Lewis. The statistical analysis of series of events. In *Monographs on Applied Probability and Statistics*. Chapman and Hall, 1966.

[6] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1996.

[7] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM'97)*, pages 160–166, Los Alamitos CA, 1997. IEEE Computer Society Press.

[8] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 40–49, Los Alamitos CA, Sept. 2004. IEEE Computer Society.

[9] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 113–122. IEEE Computer Society Press, 2005.

[10] M. Godfrey and Q. Tu. Growth, evolution, and structural change in open source software. In *Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE '01)*, pages 103–106, Vienna, Austria, 2001. ACM Press.

[11] D. Grosser, H. A. Sahraoui, and P. Valtchev. Predicting software stability using case-based reasoning. In *Proceedings of the 17th IEEE International Conference on Automated Software Engienering (ASE '02)*, pages 295–298, 2002.

[12] A. Hassan and R. Holt. Predicting change propagation in software systems. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 284–293, Los Alamitos CA, Sept. 2004. IEEE Computer Society Press.

[13] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of Langages et Modèles à Objets (LMO'02)*, pages 135–149, Paris, 2002. Lavoisier.

[14] M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept. 1980.

[15] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[16] M. Lehman, D. Perry, J. Ramil, W. Turski, and P. Wernick. Metrics and laws of software evolution–the nineties view. In *Proceedings IEEE International Software Metrics Symposium (METRICS'97)*, pages 20–32, Los Alamitos CA, 1997. IEEE Computer Society Press.

[17] R. C. Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

[18] T. McCabe. A measure of complexity. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[19] A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in oo programs. *Communications of the ACM*, 48(5):99–103, May 2005.

[20] S. Valverde, R. F. Cancho, and R. Sole. Scale-free networks from optimal design. *Europhysics Letters*, 60(4):512–517, 2002.

[21] R. Vasa, M. Lumpe, and J.-G. Schneider. Patterns of component evolution. In M. Lumpe and W. Vanderperren, editors, *Proceedings of the 6th International Symposium on Software Composition (SC 2007)*, pages 244–260, Braga, Portugal, Mar. 2007. Springer.

[22] R. Vasa and J.-G. Schneider. Evolution of cyclomatic complexity in object oriented software. In F. Brito e Abreu, M. Piattini, G. Poels, and H. A. Sahraoui, editors, *Proceedings of 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE '03)*, Darmstadt, Germany, July 2003.

[23] R. Vasa, J.-G. Schneider, C. Woodward, and A. Cain. Detecting structural changes in object-oriented software systems. In J. Verner and G. H. Travassos, editors, *Proceedings of 4th International Symposium on Empirical Software Engineering (ISESE '05)*, pages 463–470, Noosa Heads, Australia, Nov. 2005. IEEE Computer Society Press.

[24] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for Eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE '07)*, pages 9–15, Minneapolis, MN, May 2007. IEEE Computer Society Press.

[25] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, Los Alamitos CA, 2004. IEEE Computer Society Press.