Patterns of Component Evolution

Rajesh Vasa[1], Markus Lumpe[2], and Jean-Guy Schneider[1]

[1] Faculty of Information & Communication Technologies
Swinburne University of Technology
P.O. Box 218
Hawthorn, VIC 3122, AUSTRALIA
{rvasa,jschneider}@swin.edu.au

[2] Department of Computer Science
Iowa State University
Ames, IA 50011, USA
lumpe@cs.iastate.edu

Abstract. Contemporary software systems are composed of many components, which, in general, undergo phased and incremental development. In order to facilitate the corresponding construction process, it is important that the development team in charge has a good understanding of how individual software components typically evolve. Furthermore, software engineers need to be able to recognize abnormal patterns of growth with respect to size, structure, and complexity of the components and the resulting composite. Only if a development team understands the processes that underpin the evolution of software systems, will they be able to make better development choices. In this paper, we analyze recurring structural and evolutionary patterns that we have observed in public-domain software systems built using object-oriented programming languages. Based on our analysis, we discuss common growth patterns found in present-day component-based software systems and illustrate simple means to aid developers in achieving a better understanding of those patterns. As a consequence, we hope to raise the awareness level in the community on how component-based software systems tend to naturally evolve.

1 Introduction

The *Laws of Software Evolution*, as formulated by Lehman et al. [11], establish the fact that regardless of domain, size, or complexity, software systems evolve, they become more complex, and require more resources to preserve and simplify their structure. Software systems *must be continually adapted*, or else they become progressively less useful in a real-world environment. Many well-known techniques exist to facilitate system evolution in the presence of changing requirements. The key to a successful software evolution approach lies, however, not only in anticipating new requirements and adapting a system accordingly [7], but also in understanding of the nature and the dynamics of change.

Evolution is at the heart of component-based software engineering, which has become the major approach to develop modern, large-scale software systems [22, 23]. Component-based software technology has emerged from the object-oriented software

development approach, which is the predominant engineering method to build software systems today. Already a decade ago, Nierstrasz et al. [18] showed that objects provide a suitable organizational paradigm for both *decomposing* large applications into cooperating software entities and *composing* applications from pre-packaged software components. In addition, Dami [5] pointed out in his work that *extensibility* is another crucial aspect of software composition, which one must not underestimate. The desire to achieve substitutability and "plug-compatibility" of components imposes a certain discipline to structure, use, and connect component plugs that also impacts the overall design, architecture, and interaction patterns of an application.

Although we a have good grasp of the technological issues involving the evolution of class-based systems (e.g., the modular refinement of classes [2, 3, 13] and the injection of orthogonal behavior into classes [1, 3]), we have a less clear understanding of the nature and dynamics underlying change. So, how do software systems really evolve? How do components evolve? How does the interface of a component evolve? Can we provide a sufficient answer to those questions, while the definitive description of the term "component" is still elusive? In an attempt to offer some answers to these questions, the goal of this work is to provide a new perspective to the way software systems change over time. In particular, in this paper we shall (i) study selected recurring structural and evolutionary growth patterns that we have observed in present-day software systems and (ii) identify simple means that can help development teams improve the overall component-based product and process quality.

Our study focuses on *growth estimation*, an approach that offers a powerful means for proactive risk management [20]. In particular, we are interested in a *normalized ratio* of change in terms of size and complexity of component-based systems. Component-based software engineering emphasizes reuse rather than the creation of software artifacts. The evolution of a component-based system consequently involves the reuse and adaptation of existing *components off-the-shelf* (COTS) and naturally all of their embodied contextual relationships [23]. The size and complexity of these software artifacts is known and can be easily factored into the overall growth estimation of a system. However, a refined version of a system may also require new components and *partially*-developed software artifacts. The size and the complexity of these elements is largely unknown and, as a result, their development involves a fair degree of risk. However, this risk can be proactively assessed, as the growth value of these new elements is governed by a predefined and system-specific growth factor that is *unique* for each system and cultural environment within which the actual development is being carried out.

Precise assessment of the nature of how a software system evolves in the future is both an art and science [20]. The underlying development approach and utilized associated project metrics are invaluable assets that can provide a historical perspective to generate quantitative estimates on how a given system may evolve in the future. Common project metrics are, for example, the number of lines of code (LOC), the number of key classes, and the number of secondary classes. However, these metrics are based on absolute values, which can be misleading, as they only capture the absolute growth of a system. In order to achieve a better proactive risk management, we need to *normalize* the quantitative data to obtain a *relative* growth value that should be *constant* over the system's lifetime.

The rest of this paper is organized as follows: in Section 2, we introduce the concept of *software dependency graphs* and define a number of metrics based on such graphs used throughout our studies. In Section 3, we discuss our selection methodology and illustrate what techniques were used to extract information out of the software systems under investigation. Section 4 details our observations and presents the growth patterns identified. We discuss related work in Section 5 and conclude this paper in Section 6 with a summary of the main insights as well as directions for further work.

2 Understanding Software Structure

2.1 Software Metrics

Software systems exhibit two broad quantitative aspects that we can measure using a wide range of software metrics [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 rigorously collecting and analyzing these measures over time, we can distill a temporal dimension, which is capable of revealing new, yet invaluable information like the rate of size growth [11, 12] and evolutionary jumps in the complexity of a software system [8], respectively. Moreover, recent results have shown that *evolution measures* can be used to detect architectural shifts automatically [26].

Numerous approaches have been proposed and verified (e.g., purely size-oriented measures like the number of lines of code (LOC) or function-oriented measures to analyze process aspects like costs and productivity) that can help us to understand the size as well as the complexity of a software system. For object-oriented systems, common *size* measures include, for example, the *number of classes*, the *number of methods*, and the *number of public methods.* Size measures provide an indication of the *volume of functionality* provided by a software system and can be used as a broad indicator of effort required to build that system, as it takes usually more effort to create a larger-size system than a smaller one.

Complexity is commonly captured by measuring structural attributes of a software system [10]. An attractive approach to measure the complexity of class-based systems is to analyze the *fan-in* and *fan-out* ratios for a given class [26]. Theses measures naturally capture the number of classes a given class X depends upon and the number of classes that depend on X. In combination, theses ratios provide a precise information about the degree of *coupling* of X with other classes in the system. For example, a class X with a high fan-in (relative to other classes in the system) is being considered “complex”, since any changes made to X have the potential to significantly impact other classes that depend on X. Similarly, a class X that has a very high fan-out is also considered “complex”, since X makes use of a large number of different functional aspects of the system in order to satisfy its responsibilities. As a consequence, developers cannot alter X in a meaningful way before they understand all classes that X uses.

2.2 Type Dependency Graphs

In order to measure fan-in and fan-out, we need to construct a *type dependency graph* of a software system [26]. For the purpose of this presentation, a type dependency graph

G^T is an ordered pair (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., $E \subseteq V \times V$). In addition, we shall use N to denote the number of nodes and L to denote the number of directed types links of a given type dependency graph G^T throughout the rest of this paper.

In order to capture both fan-in and fan-out of a given type, represented by node $n \in V$ in G^T, we use $l_{in}(n)$ to denote the *in-degree* and $l_{out}(n)$ to denote the *out-degree* of node n. More precisely, $l_{in}(n)$ is the number of inbound links into n (i.e., $l_{in}(n)$ = $\mathbf{card}(\{l \mid l = \{n_i, n_j\} \wedge n_i, n_j \in V \wedge n = n_j \wedge i \neq j\})$) and $l_{out}(n)$ is the number of outbound links from the node n (i.e., $l_{out}(n)$ = $\mathbf{card}(\{l \mid l = \{n_i, n_j\} \wedge n_i, n_j \in V \wedge n = n_i \wedge i \neq j\})$). 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].

We can further refine the notions of in-degree and out-degree in the context of the analysis of component-based applications. Each component in a given system may comprise several classes and interfaces. The most frequent techniques used to construct these composites are *aggregation*, an approach based on inheritance and interface composition, respectively, and *containment*, a delegation-based approach to compose the often orthogonal behavior.[3] These techniques give rise to a refinement of the measures in-degree and out-degree in which we also distinguish between *intra-* and *inter*-component links. A given link to or from a node n may or may not cross the boundary of the containing component, depending on some organizational, structural, and/or functional features. For example, if an outbound link from node n ends in a node n_j that occurs within the boundary of the component in question, then we call this link an *internal* outbound link and denote by $l^i_{out}(n)$ it's corresponding *internal out-degree*. On the other hand, if an outbound link ends in a node $n_{j'}$ that lies outside of the component's boundary, then we call this link an *external* outbound link and denote by $l^e_{out}(n)$ it's corresponding *external out-degree*. Hence, the out-degree of node n is $l_{out}(n)$ = $l^i_{out}(i) + l^e_{out}(n)$.

The refinement of in-degree is defined similarly. That is, we denote by $l^i_{in}(n)$ the *internal in-degree* and by $l^e_{in}(n)$ the *external in-degree* of a type node n. The total in-degree of a node n is $l_{in}(n)$ = $l^i_{in}(i) + l^e_{in}(n)$.

Finally, for any given node n in a type dependency graph G^T, we can measure a number of additional, yet very meaningful, attributes related to its size, such as

- the total number of methods m^n defined by n;
- the number of defined *public* methods p^n;
- the number of branch instructions b^n; and
- the digital measure of inheritance i^n indicating whether node n inherits from another node n_k (apart from the language default).

2.3 Layering

The measures *in-degree* and *out-degree* provide a powerful, size-oriented tool to discover, monitor, and analyze the architectural composition of a software system. In par-

[3] The notions aggregation and containment have been popularized by the COM/ActiveX component model [21].

ticular, we are able to observe the forming of application-specific boundaries or *layers*. These layers are constituted by types, whose in-degree and out-degree share similar characteristics. Every software system (i.e., the whole application and its individual components) exhibits the layer-building behavior. For the purpose of this study, we classify each type to belong to one of four distinct layers. These layers are:

- **Foundation**: The types in the Foundation Layer F only provide their services to other types occurring *within* a given component (or system). The types in the Foundation Layer do not depend on any types except those defined in external libraries and the runtime environment. For every type $n \in F$ it holds that $l^i_{in}(n) > 0$, $l^e_{in}(n) \geq 0$, $l^i_{out}(n) = 0$, and $l^e_{out}(n) \geq 0$.
- **Central**: The types in the Central Layer C both provide services to and require services from other types occurring within a given component (or system). For every type $n \in C$, we have $l^i_{in}(n) > 0$, $l^e_{in}(n) \geq 0$, $l^i_{out}(n) > 0$, and $l^e_{out}(n) \geq 0$.
- **Top**: The types in the Top Layer T do not provide any services to other types occurring within a given component (or system). However, types in the Top Layer depend on at least one other type occurring within a given component (or system). As a result, for every $n \in T$, we have $l^i_{in}(n) = 0$, $l^e_{in}(n) \geq 0$, $l^i_{out}(n) > 0$, and $l^e_{out}(n) \geq 0$.
- **Free**: The types in the Free Layer U neither provide any services to types occurring within a given component (or system) nor require any services from the other three layers. For every type $n \in U$ it holds that $l^i_{in}(n) = 0$, $l^e_{in}(n) \geq 0$, $l^i_{out}(n) = 0$, and $l^e_{out}(n) \geq 0$. Types of the Free Layer denote either dormant software artifacts that do not contribute to the overall behavior of a component (or system) or their usage cannot be detected statically.

The reader should note that all types can be assigned to exactly one of the four layers given above, hence if V is the set of all types of a given component, then it holds that $V = T \cup C \cup F \cup U$. It also holds that $l^e_{in}(n) \geq 0$ and $l^e_{out}(n) \geq 0$ for types in any of the four layers. Hence, we can optimize our analysis and do not need to consider external inbound links and external outbound links when assigning types to layers.

Once a software system has been represented as a dependency graph G^T, we can apply appropriate graph theoretical techniques to discover, monitor, analyze, and predict how this given system will evolve in the future. By constructing a dependency graph G^T for each new version of a given software system and comparing the new graph with the one built for previous versions, we can refine the analysis process and obtain early indicators for potential risks.

3 Methodology

Both the version-specific type dependency graphs and the deduced structural layer-based decomposition of live systems provide us with suitable means not only to unveil the inherent nature of component-based software systems, but also to predict reliably their anticipated growth patterns with respect to size and complexity.

To demonstrate our approach in greater detail, we have selected five representative open-source projects and present our findings in the remainder of this work. For all

Name	Releases	Initial Size N_1	Current Size N_k	Description
Acegi	17	135	368	Role-based security framework
Active MQ	26	205	2295	Message queue framework
Hibernate	44	120	1053	Object-relational mapping framework
Spring	39	386	1527	Light-weight container
Xalan	13	207	919	XML parsing/transformation library

Table 1. Components under analysis.

systems, we have catalogued their corresponding reoccurring size and growth patterns. Based on these patterns, we can create simple *predictive models* being able to capture relevant aspects associated with the nature of how a given component-based software system evolves.

3.1 Input Data Set Selection

Our primary focus in this work is on open-source Java-based systems. In order to identify suitable systems for our empirical study, we define a number of selection criteria that are expected to yield the best results with respect to assessing the complexity, size, and evolution history of a given system. Our selection criteria are as follows:

1. The system is a single *component* or *application framework*, that is, it cannot be used as a stand-alone application and, as a consequence, has to be analyzed as a part of a bigger, composite system.
2. At least *10 builds* of the system are available. Only complete releases are considered builds. Branches and releases not derived from the main system tree are ignored.
3. The system has been in active development and use for at least *24 months* to increase the likelihood of the existence of a significant development history.
4. The system should comprise of at least *200 types* (i.e., classes and interfaces) at some point in its lifetime and should consist of no less than 100 types when being analyzed. This ensures the anaysis of components of a realistic complexity.
5. Availability of change logs that indicate defect fixes, addition of new features, and highlight structural changes. This data aids us in understanding and attributing changes.

Using these selection criteria, we have identified five systems (cf. Table 1), which provide a best match for our study with respect to time and resource constraints. The reader should note that we have initially been able to identify over 100 systems that met our selection criteria [26].

In addition, we use a *Release Sequence Number* (RSN) [4] as the pseudo-time measure for all systems under investigation. RSNs are universally applicable and independent of any release numbering scheme and/or schedule. A RSN is a sequential number allocated based on release dates, where the first version is 1 and then each subsequent version increases by one.

3.2 Extracting Metrics

In order to perform the required data mining, we developed a *metrics extraction* tool [26], especially designed to analyze Java programs and to extract growth-related 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 version of the system being investigated and generates the desired metric data.

Our extraction tool uses `ASM`, a Java Bytecode manipulation framework,[4] to collect static dependency information from the classes contained within the core JARs. For each type, 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 [26]:

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

The reader should note that all systems that we have analyzed require also some additional Java-based third party libraries. However, these third party libraries as well as all Java standard libraries do not impact the size and complexity of the analyzed software system. For this reason, our metric extraction tool ignores dependencies associated with types originating from those libraries, an approach that does not compromise the overall quality and precision of the collected data in our study.

4 Observations and Analysis

In the previous sections, we have outlined the selection criteria required to identify five suitable Java-based systems for analysis and briefly discussed the measures in which we are particularly interested in. In this section, we illustrate how the notion of *power-scaling relationship* [16] in software dependency graphs can be used as a means to discover and analyze recurring structural and evolutionary patterns in the component-based systems.

4.1 Power-Scaling Relationship

In our previous work on detecting structural changes in object-oriented software systems [26], we developed a *growth estimation model* for calculating the total number of type links L in a software dependency graph G^T given the total number of type nodes N. This model is based on a *power-scaling relationship* [16] that can be established between L and N. More specifically, if N denotes the total number of types (as defined in Section 2) and L the total number of dependencies between types (i.e., associations, inheritance, and interface refinements) [26], then it holds that

$$L = N^{\beta}, \quad \text{where } \beta \approx 1.4 \tag{1}$$

[4] `ASM` is available at *http://asm.objectweb.org*.

The reader should note, however, that β is *not* a constant! It changes marginally between successive versions and usually stabilizes around the value 1.4 as a software system matures [26].

As we will outline in the following, power-scaling relationships are central to our study as they are commonly found in software dependency graphs, most notably between the total number of types and methods, both public and private. Therefore, such relationships will allow us to create estimation models for the corresponding dependencies over a number of versions of a given software component.

4.2 Relationship between Types and Methods

As the first item in our study, we analyze the relationship between the number of types in a component with (i) the total number of methods (denoted by M) and (ii) the total number of *public* methods (denoted by P). The number of methods in a software component[5] can be considered as a measure that denotes the *volume of functionality* of this component. The public methods of a component, on the other hand, can be seen to be the potential *external interface* of that component, although in practice only a small part of this interface may be used by external parties. However, analyzing the use of the external interface of a component is beyond the scope of this study.

Analyzing the five selected systems, our data reveals that the total number of methods M and the number of public methods P grow predictably along with the number of types N. More significantly, there is a power-scaling relationship between the total number of types and the total number of (public) methods. We observe that

$$M = N^{\beta^M}, \quad \text{where } \beta^M \approx 1.35 \tag{2}$$

$$P = N^{\beta^P}, \quad \text{where } \beta^P \approx 1.30 \tag{3}$$

The reader should note that a linear model (i.e., $M = \alpha N$) for the relationship between types and (public) methods cannot be used, as it is not sensitive enough to pick up slight slopes in data and, therefore, would be unsuitable to recognize medium-sized architectural changes from one version of a system to another.

Our data also indicates that as the size of a component increases, the rate at which the public methods are added starts to decrease. Additionally, we notice that public methods are added at a faster rate early in the development life-cycle. Once a component becomes more mature and has stabilized, our data shows that it will resist an increase in its public interface. A similar observation can also be made for non-public methods.

Based on these two observations, we can estimate the total number of public methods that a given system will have in the current version given the corresponding information from the previous version. If N_v and β_v^P denote the total number of types and the scaling factor for version v of a given component, respectively, then it holds that the estimated scaling factor for version $v+1$, denoted by ${}^*\beta_{v+1}^P$, is

$${}^*\beta_{v+1}^P \;=\; \beta_v^P \;+\; \gamma^P \cdot\; \mathbf{sng}(N_v - N_{v+1}) \;\; \text{where} \;\; \gamma^P \approx 0.001 \tag{4}$$

[5] We use the term *component* as a synonym for both a whole software system and individual, self-contained software artifacts.

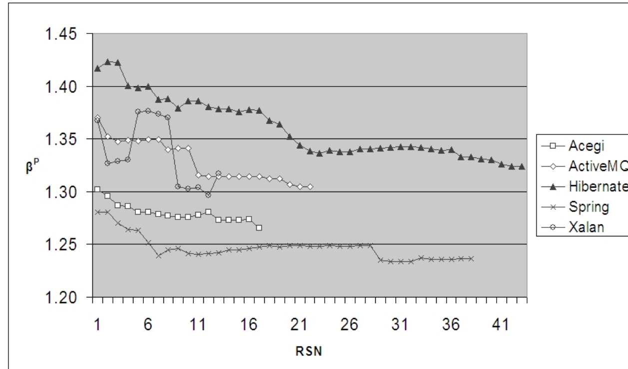

Fig. 1. Evolution of scaling factor β^P.

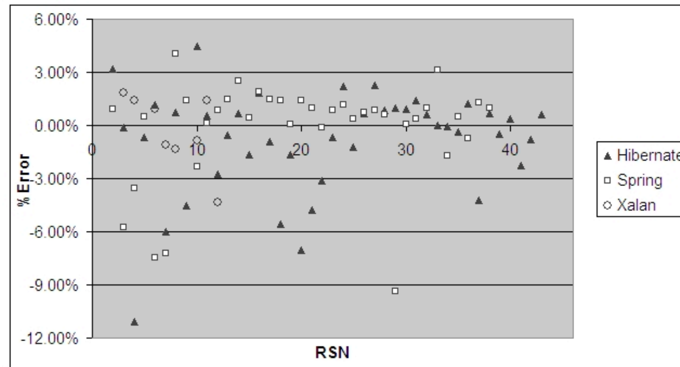

Fig. 2. Estimation error for public methods P.

with **sng** being the *signum* function.[6] γ^P in the formula above encodes the observation that β^P tends to slowly decrease as the size of a component increases. Furthermore, the more mature a component becomes, the rate at which β^P changes decreases. The distribution of the scaling factor β^P for all five components under investigation is illustrated in Figure 1. A similar relationship can be defined for methods M with a scaling factor β^M, and the estimation model holds with $\gamma \approx 0.002$. The reader may note that γ is a constant in our model, but further analysis may result in a revised model where γ is a function.

We have derived this model based on our analysis of Acegi and ActiveMQ and then applied it to the other three components to verify our growth estimation model. Our data shows that this model is able to estimate the total number of public methods in a component with no more than 3% error rate in 75% of the time (cf. Figure 2). If the error rate is beyond a given, project-dependent threshold, then this is an indication that

[6] The signum function returns $+1$, -1 or 0 if it is applied to a positive number, negative number, or zero, respectively

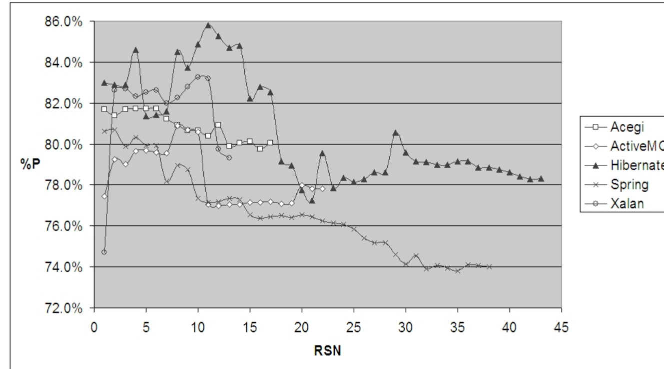

Fig. 3. Percentage of public methods $\%P$.

substantial changes have been made between two versions, a situation that deserves special documentation and analysis, in particular with respect to an emerging risk.

4.3 Percentage of Public Methods

Accross all five analyzed systems, our data shows that between 74% and 86% of all methods are public. This is unusually large and deserves special attention. We suspect that there is a natural tendency in the Java Open Source development community to create a larger number of public methods, indicating that the language makes it very easy to define public methods and does not offer a set of features that would allow the developers to choose a greater level of control over the "visibility" of the code that they write. We feel that additional modifiers in the language, in combination with appropriate training of developers, might yield better, more maintainable, outcomes. Ideally, a component should only have a very small public interface that is available to outside developers.

Our data also reveals that, with few exceptions, the proportion of public methods in a given component tends to fluctuate in the 5% range. Hence, if in the initial version 75% of the methods are public, then our data indicates that we are likely to see a range of in-between 70% and 80% over the duration of the project. However, the tendency for this measure is to decrease rather than to increase. So, over time the percentage of public methods will tend to go down from 80% to 75% rather than the other way.

Similar to the growth estimation model illustrated in Section 4.2, we are able to define a model to estimate the number of public methods in the next version of a component or systems, denoted by $^*P$, as

$$^*P_{v+1} \;=\; N_{v+1}^{\,^*\beta^P_{v+1}} \tag{5}$$

This model allows us to estimate the total number of public methods in a system (i) within 2% accuracy 70% of the time and (ii) within 5% accuracy 90% of the time. When the estimated value is not within a small margin of error compared to the "real" value, it provides a good indicator for the development team to reflect on what changes were

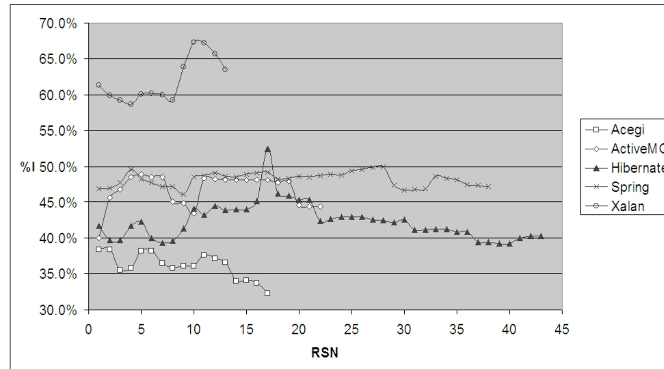

Fig. 4. Percentage of derived types.

made to the component/framework in order to provide additional documentation. The threshold should be determined by the development team based on observations from their product data over the recent past. Further, our data shows that as time progresses, the estimation accuracy increases because less changes are made and, as a consequence, the error threshold should be reduced accordingly. Figure 3 illustrates the evolution of the percentages of public methods for all analyzed systems.

4.4 Growth Proportion in Inheritance

Our study also reveals a rather intriguing property, namely that the proportion of types that extend (i.e., are derived from) another type is strongly bounded and does not change significantly. We found that, in general, between 35% and 50% of the types are derived from another type. On average, there is a 45% chance that a type is derived from another type. The exception is Xalan, which has a much higher number of derived types (cf. Figure 4). Further analysis showed that Xalan, being an XML parser, exhibits a strong relationship to the underlying hierarchical structure of XML documents. This seems to be an architectural choice made by the development team.

Our data also demonstrates that the variation in the proportion of types that are derived from another type, although system dependent, is very low. Generally, the observed variation is around 5%. There is no strong tendency over time for the components/frameworks to either have more or less types that (i) are derived or (ii) derive another type. This might indicate that there is a certain cultural bias and developer habit at work and the overall size/maturity does not have a direct impact on the rate of change. This, however, is something that needs further investigation. In the five systems that we have analyzed, the probability of significant change in the number of types deriving from another type (i.e., more than 2%) between two subsequent versions is very low (on average, less than 0.15%). This is further illustrated in Figure 4.

4.5 Type Distribution in Layers

In all five systems, we consistently find that the Central Layer (c.f. Section 2.3) contains the majority of the types. On average, the Central Layer C contains about 80% of the

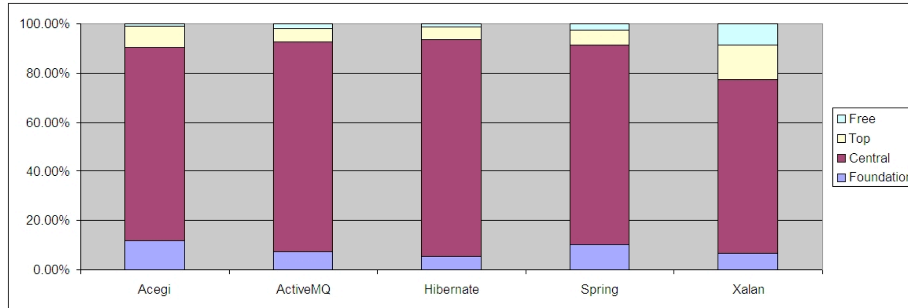

Fig. 5. Type distribution in layers.

types in the component, the Top Layer T and the Foundation Layer F contain 9% each, and the Free Layer U contains the remaining 2% of the types. Figure 5 illustrates the type distribution of the *last* available release for each analyzed system.

Although the precise values change from component to component, it is interesting to note that the Foundation Layer has a small number of types (i.e., around 9%). This indicates that developers naturally tend to keep the number of the types in the layer that has the highest *ripple impact* low.

The natural interface exposed by the components is most likely located in the Top Layer since it contains types that are not directly used internally. Our study shows that compared with the Central Layer, the Top Layer is also fairly small, indicating that component designers tend to keep the interface of a component as small as possible. Although the Top Layer contains the set of types that external users of this component are most likely to access, it does not restrict types in other layers to be used. It would be ideal, if languages like Java provide better language abstractions that allow developers to explicitly tag the external interface to a component, allowing for further analysis.

To further illustrate the distribution of types between layers, consider Table 2, where the layer distribution for the first 18 versions of Hibernate is recorded. Besides the version number/name, Table 2 lists the number of classes for a given RSN, the number of types (classes or interfaces) that are derived from another type (denoted by I), and the percentage of classes in the Foundation, Central, Top, and Free layers (denoted by $\%F$, $\%C$, $\%T$, and $\%U$, respectively). Table 2 also lists the total number of methods as well as the number of public methods (denoted by M and P, respectively).

We would like to highlight that between RSN 3 and 4, an unusual amount of change can be observed in type distribution across layers. An inspection of the available change logs reveals that a major refactoring of the core code was performed. The reader may note that this change could also be detected purely by observing the growth in size (i.e., N and P). However, between RSN 14 and 15, another change in the distribution of types between layers occurs. This change cannot be detected by observing size measures. N, P, and I all remain unchanged.

Further, as documented in the change logs and discussion groups, we would like to note that the team changed the underlying structure between version 1.2.2 and 2.0. This change can be observed in the distribution of types between layers (cf. Figure 6).

RSN	Version	N	I	$\%F$	$\%C$	$\%T$	$\%U$	M	P	β^M	β^P
1	0.9.1	120	50	0.075	0.850	0.075	0.000	1065	873	1.456	1.415
2	0.9.2	126	50	0.087	0.841	0.071	0.000	1175	963	1.462	1.421
3	0.9.3	126	50	0.079	0.849	0.071	0.000	1174	962	1.461	1.420
4	0.9.5	144	60	0.049	0.833	0.104	0.014	1246	1044	1.434	1.399
5	0.9.6	163	69	0.049	0.834	0.104	0.012	1523	1229	1.439	1.397
6	0.9.9	185	74	0.065	0.827	0.092	0.016	1829	1473	1.439	1.397
7	0.9.10	201	79	0.060	0.821	0.104	0.015	1923	1553	1.426	1.386
8	0.9.14	222	88	0.068	0.815	0.104	0.014	2136	1789	1.419	1.386
9	1.0.0	242	100	0.062	0.822	0.103	0.012	2312	1920	1.411	1.377
10	1.1.0.b7	270	119	0.059	0.826	0.104	0.011	2762	2329	1.415	1.385
11	1.1.0	296	128	0.071	0.821	0.098	0.010	3102	2647	1.413	1.385
12	1.1.6	342	152	0.070	0.816	0.102	0.012	3690	3131	1.408	1.379
13	1.2.0	367	161	0.084	0.798	0.104	0.014	4047	3337	1.406	1.374
14	1.2.1	377	166	0.082	0.801	0.103	0.013	4198	3469	1.406	1.374
15	1.2.2	377	166	0.069	0.878	0.040	0.013	4260	3439	1.409	1.373
16	2.0b1	390	176	0.074	0.885	0.028	0.013	4489	3630	1.410	1.374
17	2.0.0	364	191	0.038	0.926	0.033	0.003	4083	3280	1.410	1.373
18	2.1.0	446	206	0.040	0.910	0.047	0.002	5301	4154	1.406	1.366

Table 2. History of Hibernate.

However, this level of change cannot be detected by purely looking at size growth – the number of classes N only increases by 13 (cf. Table 2). It is interesting to note that both, β^M and β^P, do not change significantly between version 1.2.2 and 2.0, either, yet another indication that a range of measures is needed to truly understand evolution of software.

4.6 Proportional Growth in Layers

Another result of our study is that components undergo phases of changes and that the proportion of types in the various layers slightly changes over time. When a component is not very mature (i.e., early in its development life span), the distribution of types in the various layers changes much more frequently than in a component that has reached a certain level of maturity. Hibernate, for example, underwent three distinct phases so far. Early in its life span, the proportion of types in various layers has changed much more than in later releases of Hibernate, as is illustrated in Figure 6.

But why is it interesting to study the evolution in the layers themselves? In software development, there are two distinct development strategies, namely *top-down* and *bottom-up*. In the top-down approach, we develop the interface of a component and then slowly add the functionality to support this interface. Hence, as per our layer definition, the Top Layer components are defined first, followed by the Central Layer and finally the Foundation Layer components are added. In the bottom-up development approach, however, this sequence is reversed and the Foundation Layer is developed first. In practice, one would expect a mixture of both approaches to be used by software developers, with the tendency to favor one approach over the other, when we take a distinct time interval.

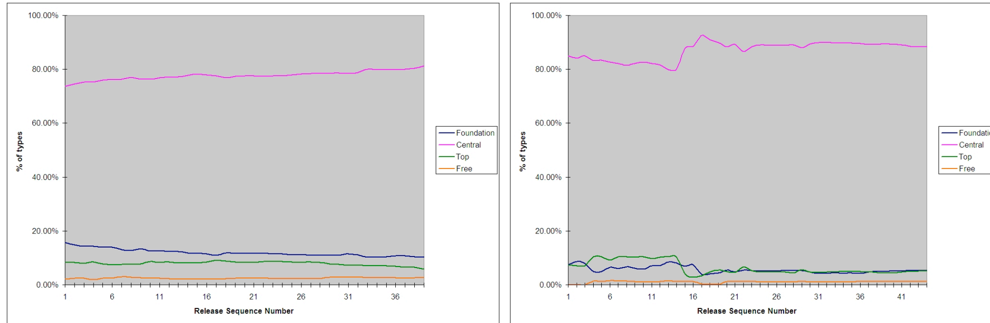

Fig. 6. Type evolution of the Spring framework (left) and Hibernate (right).

In order to identify a bottom-up development approach, we should be able to observe the proportion of the number of classes in the Foundation Layer decreasing over time, while the Central Layer and the Top Layer increase. This would indicate that the Foundation layer types have matured and the developers are working on the code in other layers. However, only the Spring framework shows more of a bottom-up approach rather than the top-down approach (cf. Figure 6). We hypothesize that this upwards trend in the Central Layer will stabilize as the product matures. It does not seem to be practical, however, to have a system where more than 95% of the types are in the Central Layer. On the contrary, our study shows that there is a natural tendency to host around 80% of the types in the Central Layer.

Developers working on the Hibernate project exhibit a tendency to add types equally in all layers. This seems to be the case for all analyzed systems and we cannot identify any periods where either a top-down or the bottom-up approach is clearly visible. Our data reveals that evolution of components tends to happen in vertical slices, where the types are distributed over all layers. A larger sample size may provide further information and other trends, however this is beyond the scope of our current anaylsis.

5 Related Work

Modern software systems are built from of a large number of interacting and mutual-dependent parts. One way to study these systems is to use *complex systems theory* [14], which suggests that in order to understand a complex system, one should use a *top-down* approach in which the system properties are inferred from its observable behavior rather than focusing on the individual parts in the beginning. This position is also taken by Newmann [17], who argues that complex systems exhibit a certain set of *emergent* properties, which become only visible at the system level and may not have been intentionally created by the system designers. Understanding and cataloguing emergent properties can provide us with valuable insights and a new prospective to grasp the complexity and evolutionary growth patterns of modern, complex component-based software systems.

Much of the seminal work in the field of software evolution has been done over a number of years by Lehman et al. [11]. Their work suggests that at the system level, the

evolutionary behavior of a software system is systemic and not completely under the control of the individual developers. Turski [25] made a similar observation. Based on his analysis of the nature of software evolution, he presented an *inverse-square* model [25] that suggests that the growth of a software system is inversely proportional to its complexity, and as complexity increases, the rate of growth is constrained. However, while analyzing the Linux operating system, Godfrey et al. [8, 9] discovered that some systems have a *hyper-linear* growth curve. As a consequence, the models proposed by Turski and Lehman et al. do not always hold. Godfrey et al. concluded that the architecture of a system determines the evolutionary growth potential and a modular architecture allows for a system to grow faster than otherwise possible.

Mockus et al. [15] studied the evolution of Mozilla and the Apache web server, both open source software systems. They, like Godfrey et al., argue that the design structure of the software system has a direct impact on the development speed. A highly modular, component-based architecture allows for fast evolution, whereas a highly interdependent architecture generally requires a longer period of time between released versions.

In our previous work on software evolution [26], we presented a 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 G^T. We also showed that when the estimation model fails to predict the growth within a 7% error margin, then this error is induced by significant architectural shifts in the analyzed version of the software system. However, these architectural shifts may also signify a potential risk that needs to be addressed in a proactive manner. This proactive risk management is at the heart of our growth estimation model as it provides a powerful feedback mechanism for both to improve the development process and to offer guidance to a more controllable evolution practice of software systems. Furthermore, the results of our work also lend renewed support to Turski's hypothesis [24] that, as system complexity increases, the rate of growth of the software systems tends to decrease.

6 Conclusions and Future Work

Developers constructing components in a software system can mitigate risks by better understanding typical patterns of software evolution. Using an emperical investigation of popular software components with a substantial development history, we presented recurring structural and evolutionary patterns in this work. Supported by quantitative analysis and the patterns identified, we have highlighted atypical evolution of components and discussed the reasons that caused such changes. In this context, we have observed that for certain types of changes, more than one measure may be needed to highlight deviations from normal growth patterns. However, we have not yet been able to identify which measure can detect what kind of atypical change; this is a topic for future investigations.

Although our analysis shows that software grows over time, the structure and scope of growth is in general not erratic. We have observed that the percentage of derived types in a given component does not change significantly over time. Also, the naturally exposed interface of a component does not change as much as commonly perceived.

Our investigation into the nature of software evolution lead us to construct type dependency graphs and classify the types within a component into four orthogonal layers. We noticed that, in general, the distribution of the proportion of types in these layers is relatively stable. Where substantial changes in the proportional distribution were observed, they could always be attributed to significant architectural changes. As a side-effect of the layering, we were also able to detect the high-level construction methodology (e.g, top-down, bottom-up) that was most likely used to build a given component.

The growth estimation models presented in this work rely on information of the previous version of a component and offer low error rates. It is likely that we can improve the accuracy of our estimations by using information of more than one previous version. This, however, is still the topic of ongoing work.

References

1. Andrew W. Appel with Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, Second edition, 2002.
2. Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling Visibility of Class Extensions. *Journal of Computer Languages, Systems & Structures*, 31(3–4):107–126, May 2005.
3. Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proceedings OOPSLA 2000*, volume 35 of *ACM SIGPLAN Notices*, pages 130–146, October 2000.
4. D.R. Cox and P.A.W. Lewis. The Statistical Analysis of Series of Events. In *Monographs on Applied Probability and Statistics*. Chapman and Hall, 1966.
5. Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. PhD thesis, Centre Universitaire d'Informatique, University of Geneva, CH, 1994.
6. Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. Thomson Publishing, second edition, 1996.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.
8. Michael Godfrey and Qiang 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.
9. Michael W. Godfrey and Qiang Tu. Evolution in Open Source Software: A Case Study. In *Proceedings of the 16th International Conference on Software Maintenance (ICSM '00)*, San Jose, California, October 2000. IEEE Computer Society.
10. C.F. Kemmerer. Empirical Research on Software Complexity and Software Maintenance. *Annals of Software Engineering*, 1(1):1–22, 1995.
11. M. M. Lehman, D. E. Perry, J. C. F. Ramil, W. M. Turski, and P. Wernik. Metrics and Laws of Software Evolution – The Nineties View. In *Proceedings of the Fourth International Symposium on Software Metrics (Metrics '97)*, pages 20–32, Albuquerque, New Mexico, November 1997.
12. Meir M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980.
13. Markus Lumpe and Jean-Guy Schneider. On the Integration of Classboxes into C#. In Welf Löwe and Mario Südholt, editors, *Proceedings of the 5th International Symposium*

on Software Composition (SC 2006), LNCS 4089, pages 307–322, Vienna, Austria, March 2006. Springer.
14. Melanie Mitchell and Mark Newmann. Complex Systems Theory and Evolution. In M. Pagel, editor, *Encyclopedia of Evolution*. Oxford University Press, 2002.
15. Audris Mockus, Roy Fielding, and James Herbsleb. A Case Study of Open Source Software Development: The Apache Server. In *Proceedings ICSE 2000*, pages 263–272, Limerick, Ireland, June 2000.
16. C.R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, 68(4):46–116, 2003.
17. Mark E.J. Newmann. The Structure and Function of Complex Networks. *SIAM Review*, 45(2):167–256, 2003.
18. Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. Component-Oriented Software Development. *Communications of the ACM*, 35(9):160–165, September 1992.
19. Alex Potanin, James Noble, Marcus Frean, and Robert Biddle. Scale-free Geometry in OO Programs. *Communications of the ACM*, 48(5):99–103, May 2005.
20. Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Sixth edition, 2005.
21. Dale Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.
22. Johannes Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.
23. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, Second edition, 2002.
24. Wladyslaw M. Turski. Reference Model for Smooth Growth of Software Systems. *IEEE Transactions on Software Engineering*, 22(8):599–600, August 1996.
25. Wladyslaw M. Turski. The Reference Model for Smooth Growth of Software Systems Revisited. *IEEE Transactions on Software Engineering*, 28(8):814–815, 2002.
26. Rajesh Vasa, Jean-Guy Schneider, Clinton Woodward, and Andrew Cain. Detecting Structural Changes in Object-Oriented Software Systems. In June Verner and Guilherme H. Travassos, editors, *Proceedings of 4th International Symposium on Empirical Software Engineering (ISESE '05)*, pages 463–470, Noosa Heads, Australia, November 2005. IEEE Computer Society Press.