Software Change Dynamics: Evidence from 35 Java Projects

Markus M. Geipel
Chair of Systems Design
ETH Zurich, Kreuzplatz 5
8032 Zurich, Switzerland
mgeipel@ethz.ch

Frank Schweitzer
Chair of Systems Design
ETH Zurich, Kreuzplatz 5
8032 Zurich, Switzerland
fschweitzer@ethz.ch

ABSTRACT

In this paper we investigate the relationship between class dependency and change propagation in Java software. By analyzing 35 large Open Source Java projects, we find that in the majority of the projects more than half of the dependencies are never involved in change propagation. Furthermore, our analysis shows that only a few dependencies are transmitting the majority of change propagation events. An additional analysis reveals that this concentration cannot be explained by the different ages of the dependencies. The conclusion is that the dependency structure alone is a poor measure for the change dynamics. This contrasts with current literature.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design; D.2.8 [**Software Engineering**]: Metrics

General Terms

Design, Measurement

1. INTRODUCTION

As Parnas [11] noted, software is aging and constant change effort is needed to keep a piece of software up to date. Therefore, as Bohner and Arnold [3] and Bennett and Rajlich [2] point out, a good software architecture should be evolvable, flexible. An obstacle is the fact that, a change in one part often induces changes in dependent parts. Rajlich [12] and Lehman et al. [7] call this problem "change propagation" or "ripple effects". Consequently, to pass judgment on the evolvability of a given software architecture it is important to understand these propagation dynamics. Many scholars believe that the network of dependencies between modules is particularly relevant in this context as changes propagate along exactly these dependencies. The predominant analysis methodology is the Design Structure Matrix, originally developed by Steward [14] to model engineering tasks. The idea is to represent a system as an adjacency matrix in which the entries codify the strength of dependency. Sullivan et al. [15] applied this approach to software architectures. Based on this MacCormack et al. [10] present a method to calculate the so-called "Propagation Cost" of a software based on the dependency structure. Also Sangal et al. [13] point out that a high concentration of dependencies acts as a propagator of change. We argue that this entirely dependency focused approach to change propagation leaves several important questions unanswered. Assuming that dependencies matter, are really all of them relevant? The answers to this question has important practical ramifications: If the majority of dependencies are a significant transmitter of change propagation, they should be minimized just as Lieberherr and Holland [8] suggest. On the other hand, if only some dependencies matter, we could profit from code reuse without worrying about the generated dependencies. Apart from this, we cannot assume that propagated changes are evenly distributed. In this paper we answer these questions by contrasting the dependency structure of 35 large Java projects with their change dynamics. The analysis of the change dynamics takes its cues from previous work in this field. In particular Ball et al. [1], Clarkson et al. [4], Gall et al. [5], Hassan and Holt [6], Rajlich [12]

2. RESEARCH DESIGN

This section describes our research design. In the first two subsections we explain how data on change coupling are extracted from version control systems as well as how the dependency structure is calculated. In the next two subsections the analyzes are presented which measure the impact of dependencies on the change dynamics and the concentration of change propagation. Finally, we describe the particular data set used.

2.1 Class Dependencies

We model Object Oriented software as network of classes which are dependent on each other. Consequently, this network may be represented as a graph containing nodes and edges or as an adjacency matrix. In this paper we stick to the adjacency matrix view used in the Design Structure Matrix community. We refer to the dependency matrix as D were $D_{i,j} = 1$ means that i depends on j. $D_{i,j} = 0$ on the other hand is interpreted as independence. There is a dependency between classes i and j if i extends or implements j, i calls a method provided by j, i references members of j, i uses j as member or variable. In each of these cases we set $D_{i,j} = 1$. Please note that D is an asymmetric matrix. This procedure is in line with previous works by Sullivan et al. [15] and Sangal et al. [13]. Of course, D is only valid at one particular point in time as not only elements of a system change but also the structure of the dependency network itself: Software projects grow; new modules are added and sometimes old ones are removed. In our case we take the latest snapshot of D. Removed nodes are no longer of interest and that the currently existing nodes are of different age does not interfere with our analysis. The only serious issue are changing dependencies. Fortunately, with rare exceptions, the following rule holds true throughout the projects in our data set: A dependency between two classes i and j comes into existence simultaneously with the creation of the younger class. This also facilitates the calculation of dependency ages. The dependency is removed with the removal of either i or j. In between dependencies are basically constant. Thus, only the different ages of the dependencies might interfere with our analyzes. We address the issue in section 2.5.

2.2 Change Coupling

In the previous section the dependency matrix was defined. In contrast to this static view, we now define a dynamic view on the software architecture based on change propagation. We show how a matrix similar to the dependency matrix D can be calculated. Its entries do not indicate dependency, but the number of times classes have been changed simultaneously. Let us refer to this matrix as C and to the event of two classes being changed at the same time as “co-change”. To construct C we need: First, the set of classes. Let us use n to denote their number. Second, change events which record modification of the classes. Henceforth we use m to refer to the number of recorded change events. An event in the change history can be expressed as an n-dimensional vector $\vec{h}$. Each entry shows in binary form whether a class has been modified. Imagine for example a piece of software with three classes. The change event $\vec{h}_1 = (011)^T$ indicates that classes two and three were modified. Each $\vec{h}$ thus corresponds to one commit in the version control system. The change history consisting of all $\vec{h}$ can be written in matrix form: Each change vector forms a column in the change history matrix H.

$$H = (\vec{h}_1 \vec{h}_2 \dots \vec{h}_m) \tag{1}$$

H is of size $n \times m$. By multiplying H with its transposed H^T the co-change matrix C is derived:

$$C = HH^T. \tag{2}$$

C has dimension $n \times n$ and indicates how many times each element has been modified concurrently with other elements. An entry $C_{i,j} = 3$ tells us that classes i and j have been modified 3 times together. Please note that C in contrast to D is symmetric.

2.3 Dependencies and Change Propagation

To reveal the connection between dependency and change propagation we need to compare the change dynamics of dependent classes with the one between independent ones. The fact that two classes (i and j) have been modified at least once simultaneously is expressed by $C_{i,j} \geq 1$. Further, the fact that two classes are connected by a dependency is expressed by $D_{i,j} = 1$. A straight forward measure for the influence of dependencies on change propagation is the conditional probability $P_D := P(C_{i,j} \geq 1 | D_{i,j} = 1)$ given $i \neq j$. Read: the probability that two different classes have at least once been modified together, given that they are connected by a dependency. P_D is calculated from the data as follows:

$$P_D = \frac{|\{D_{i,j} = 1 \wedge C_{i,j} \geq 1\}|}{|\{D_{i,j} = 1\}|} \tag{3}$$

As a reference we also compute the conditional probability $P_{\neg D} := P(C_{i,j} \geq 1 | D_{i,j} = 0)$ that two unconnected classes $D_{i,j} = 0$ have at least once been modified together. $P_{\neg D}$ is calculated as follows:

$$P_{\neg D} = \frac{|\{D_{i,j} = 0 \wedge C_{i,j} \geq 1\}|}{|\{D_{i,j} = 0\}|} \tag{4}$$

2.4 Concentration of Propagated Change

In 1905 the economist Max Otto Lorenz proposed a concise method to analyze and visualize income inequalities [9]. Since then the so-called Lorenz curve has be used to describe concentration and inequality in various contexts. We will use it now to analyze the concentration of propagated changes. To calculate the Lorenz curve, we first need the number of propagated changes for each single dependency. Let us refer to this set as $\mathbf{\Pi} = \{C_{i,j} : D_{i,j} = 1\}$ First, $\mathbf{\Pi}$ is normalized such that its entries sum up to one. Let us refer to the result as $\boldsymbol{\pi}$. The elements of $\boldsymbol{\pi}$ are addressed by using subscripts. Second, the entries of $\boldsymbol{\pi}$ are rearranged in ascending order such that $r < s \Rightarrow \pi_r \leq \pi_s$ Finally, the Lorenz curve $\mathrm{L}(x)$ with $x \in [0, 1]$ is calculated by cumulating the first x percent of the elements of $\boldsymbol{\pi}$.

$$\mathrm{L}(x) = \sum_{k=0}^{\lfloor x * |\boldsymbol{\pi}| \rfloor} \pi_k \tag{5}$$

$\mathrm{L}(x)$ is interpreted as follows: The x percent least active dependencies cause $\mathrm{L}(x)$ percent of the propagated changes; the x percent most active dependencies cause $1 - \mathrm{L}(1 - x)$ percent of the propagated changes. For clarity's sake, we refer to $1 - \mathrm{L}(1 - x)$ as $\neg\mathrm{L}(x)$. Please note that an equal distribution leads to $\mathrm{L}(x) = x$. Any unequal distribution results in a convex curve. The higher the concentration or inequality the higher the curvature and thus the deviation from the diagonal line.

2.5 Excluding the Effect of Age

Concentration of co-changes is per se not an indication that dependencies are heterogeneous concerning their change propagation. Different ages of the dependencies could cause the concentration. This means that we cannot compare the Lorenz curve found in the data with the line of perfect equality. Instead as reference curve we use the hypothetical concentration which would be caused by the existing age distribution under the assumption of homogeneous change propagation. Let us assume that dependencies accumulate co-changes linearly in time. This means that the expected number of co-changes c depending on age t of the dependency can be expressed by the following function:

$$c(t) = b_1 + b_2 t \tag{6}$$

b_1 and b_2 are constants specific to a given project. Next, the average number of co-changes per dependency in the real data is calculated. We thus have an $\hat{c}(t)$ to which $c(t)$ can be fitted. Based on $c(t)$ we can calculate the number of co-changes each of the dependencies in the data would exhibit if the earlier discussed homogeneity of change propagation assumption held true. From this hypothetical co-change distribution, the hypothetical Lorenz curve $\mathrm{L}^*(x)$ is compiled. $\mathrm{L}^*(x)$ serves as a reference, giving the concentration of co-changes, the system would exhibit if age was the only difference between the dependencies. If $\mathrm{L}(x)$ is stronger bent than $\mathrm{L}^*(x)$, co-changes concentrate more than we would expect, based on the existing age differences.

2.6 Data

Our empirical study focusses on Java projects for several reasons: Java, unlike other popular languages such as C++, was designed from scratch to be an object-oriented language. Each class is defined in a separate file. For this reason, file changes can be directly mapped to class changes. Furthermore, Java enjoys a high popularity in the Open Source community. The project set used in this paper comprises 35 projects, and is listed in table 1. SourceForge served as our main data source, contributing 33 of the projects. Detailed information on each of them can be found at `http://NAME.sf.net`, where `NAME` stands for the name of the project. As for the selection, we took the 36 largest Java projects using CVS as version control system. Next we verified the data quality which led to the exclusion of three projects: EasyEclipse was excluded as it only constitutes a repackaging of the Eclipse IDE. Furthermore, the projects OpenQRM and OACBPMF were excluded as their version log files show hardly any activity. Finally, the set of SourceForge projects was complemented with two further projects: AspectJ (see `http://www.eclipse.org/aspectj/`) and Eclipse (see `http://www.eclipse.org/`), both hosted by IBM. This makes all in all 35 projects.

3. EMPIRICAL RESULTS

In this section we present and interpret the empirical results of the analysis described in the previous section.

3.1 Active and Change Neutral Dependencies

Table 1 (columns one and two) shows P_D and $P_{\neg D}$ for our sample set of 35 projects. P_D ranges between approximately 15% and 60%, $P_{\neg D}$ is less than 5% for all projects except the project AspectJ. This means that the existence of a dependency between two classes significantly raises the chance of change propagation. Yet, we have to acknowledge that in all analyzed projects except for three, the majority of the dependencies are "change neutral": More than half of the dependencies never propagated change in the entire development history. As pointed out in the introduction, many authors implicitly assume an equal distribution. In the light of the just presented results, this seems to be problematic. If half of the dependencies are change neutral, the dependency structure of a piece of software is a very crude measure for its change dynamics.

3.2 Co-Changes Concentrate

Figure 1 shows in black the Lorenz curve for the Eclipse project considering all dependencies. It can be seen, that this curve is strongly bent. As a reference, the dashed line marks the line of equality (diagonal line) and the dotted lines mark the point where $x = 90\%$. The graph indicates that the most active 10% of the dependencies are responsible for over 70% of the change propagation along dependency links. This result is based on approximately 235 000 dependencies and nearly 284 000 thousand co-changes.

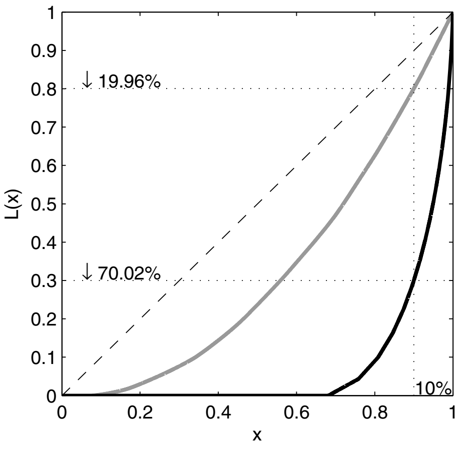

Figure 1: Concentration of co-changes. Solid black line: The Lorenz curve $\mathrm{L}(x)$ for Eclipse. Solid gray line: The reference curve $\mathrm{L}^*(x)$ (see section 2.5).

The reference curve $\mathrm{L}^*(x)$ is shown in gray. It indicates the hypothetical co-change distribution resulting from an architecture with homogenous change propagation but the same age distribution as the real world architecture (see section 2.5 for the definition). $\mathrm{L}^*(x)$ shows clearly less concentration than $\mathrm{L}(x)$: It predicts only 19.96% of co-changes for the most active 10% dependencies. This means that different ages of the dependencies are not a sufficient explaination for the high concentration of co-changes found in the data. We ran the same analysis for the other projects in the data set. To reproduce the results here in a compact way, we condensed each curve to the point $\neg\mathrm{L}(0.1)$ (respectivly $\neg\mathrm{L}^*(0.1)$) which indicates how much change events are propagated along the 10% most active dependencies. See table 1, column three and four. In the majority of the projects the value is above 50%. This means that in most projects only 10% of the dependencies are responsible for over half of the change propagation along dependency links. In all projects $\neg\mathrm{L}(0.1) > 0.2$. The higher the concentration of change propagation and thus the value of $\neg\mathrm{L}0.1)$ the better the refactoring effort can be targeted. In all projects except for Jazilla, $\neg\mathrm{L}(0.1)$ is higher than $\neg\mathrm{L}^*(0.1)$. The case of Jazilla is particular as the curves $\mathrm{L}^*(x)$ and $\mathrm{L}(x)$ take very different forms and cross each other[1].

4. CONCLUSIONS

The analysis of P_D and $P_{\neg D}$ as well as the concentrations of co-changes show, that any estimation of change propagation based on the dependency network will be highly inaccurate, as the implicit assumption of homogeneity of dependencies does not hold. We therefor conclude, that a differentiated view on dependency between classes is needed. Theoretical models must take heterogeneity of dependencies into account. For software engineering practice, we conclude the following: As many dependencies are never involved in

[1]Please see the complete set of plots on our website `www.sg.ethz.ch/research/oss`

project	$\mathbf{P_D}$	$\mathbf{P_{\neg D}}$	$\neg\mathbf{L}(0.1)$	$\neg\mathbf{L}^*(0.1)$
architecturware	$33.2_{\pm 1.3}$	$0.5_{\pm 0.1}$	58.32	17.96
aspectj	$57.8_{\pm 1.6}$	$18.5_{\pm 0.1}$	45.53	15.60
azureus	$38.2_{\pm 1.3}$	$1.2_{\pm 0.1}$	60.16	14.92
cjos	$36.3_{\pm 1.1}$	$0.2_{\pm 0.1}$	44.36	14.50
composestar	$49.0_{\pm 2.0}$	$1.7_{\pm 0.1}$	47.68	28.01
diee-mad	$23.0_{\pm 2.4}$	$1.1_{\pm 0.1}$	64.16	16.70
eclipse	$21.5_{\pm 0.3}$	$0.3_{\pm 0.1}$	70.02	19.96
enterprise	$16.5_{\pm 1.3}$	$0.5_{\pm 0.1}$	64.98	11.56
findbugs	$21.1_{\pm 0.7}$	$0.6_{\pm 0.1}$	64.91	21.42
fudaa	$34.5_{\pm 0.9}$	$1.6_{\pm 0.1}$	60.06	19.32
gpe4gtk	$28.9_{\pm 1.4}$	$1.1_{\pm 0.1}$	66.12	19.79
hibernate	$45.6_{\pm 1.3}$	$1.1_{\pm 0.1}$	58.59	19.46
jaffa	$29.5_{\pm 1.5}$	$0.9_{\pm 0.1}$	58.71	10.92
jazilla	$62.0_{\pm 1.6}$	$7.1_{\pm 0.1}$	22.02	36.44
jedit	$58.0_{\pm 1.8}$	$1.2_{\pm 0.1}$	41.80	15.62
jena	$33.0_{\pm 1.1}$	$1.0_{\pm 0.1}$	61.41	14.94
jmlspecs	$48.2_{\pm 1.9}$	$3.1_{\pm 0.1}$	52.87	14.51
jnode	$28.0_{\pm 0.8}$	$0.5_{\pm 0.1}$	65.36	18.14
jpox	$35.1_{\pm 1.2}$	$0.8_{\pm 0.1}$	66.80	18.69
openhre	$34.6_{\pm 2.0}$	$1.1_{\pm 0.1}$	29.25	12.69
openjacob	$24.0_{\pm 1.3}$	$1.2_{\pm 0.1}$	57.94	14.27
openuss	$26.8_{\pm 1.5}$	$1.6_{\pm 0.1}$	55.60	13.24
openxava	$28.0_{\pm 1.8}$	$2.0_{\pm 0.1}$	82.95	11.08
pelgo	$18.1_{\pm 1.3}$	$0.3_{\pm 0.1}$	55.28	10.38
personalaccess	$29.8_{\pm 1.9}$	$1.3_{\pm 0.1}$	65.48	15.48
phpeclipse	$49.5_{\pm 1.9}$	$2.0_{\pm 0.1}$	42.10	12.96
rodin-b-sharp	$21.8_{\pm 1.0}$	$0.5_{\pm 0.1}$	77.01	18.94
sapia	$42.4_{\pm 1.7}$	$1.1_{\pm 0.1}$	50.70	16.70
sblim	$12.7_{\pm 0.6}$	$1.5_{\pm 0.1}$	88.65	11.26
springframework	$33.6_{\pm 1.2}$	$0.5_{\pm 0.1}$	60.87	14.39
squirrel-sql	$27.4_{\pm 1.5}$	$1.2_{\pm 0.1}$	63.81	11.76
university	$26.8_{\pm 1.6}$	$4.0_{\pm 0.1}$	37.49	10.25
xendra	$19.0_{\pm 1.3}$	$1.5_{\pm 0.1}$	52.52	12.95
xmsf	$34.8_{\pm 1.6}$	$2.4_{\pm 0.1}$	67.37	16.56
yale	$31.5_{\pm 1.5}$	$2.6_{\pm 0.1}$	51.03	13.22

Table 1: $P(\mathcal{C}|\mathcal{D})$, $P(\mathcal{C}|\neg\mathcal{D})$, $\neg\mathrm{L}(0.1)$ and $\neg\mathrm{L}^*(0.1)$ for each project. The error margins are given on a confidence level of 99% under the assumption of a binomial distribution.

change propagation, a lopsided minimization of dependencies will not necessarily improve the architecture in respect to its flexibility. We should keep in mind that dependencies serve a purpose: code reuse. Each dependency means that functionality was not duplicated but reused. Consequently, a highly connected class in the dependency network is not necessarily an evidence of flawed design but distinguishes the class as very important. Only if this importance goes along with change propagation, the design deserves a critical look at. This also means that approaches relying solely on the dependency structure to judge the change dynamics and flexibility of a software architecture are problematic. We argue that both the dependency view and the change coupling view need to complement each other to paint a full picture of the software architecture. Further investigation is needed to find out which properties of the dependencies generate the observed heterogeneity.

References

[1] T. Ball, J. Kim, A. Porter, and H. Siy. If your version control system could talk. In *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, 1997.

[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. ISBN 1-58113-253-0.

[3] S. Bohner and R. Arnold. *Software change impact analysis.* IEEE Computer Society Press, 1996.

[4] P. Clarkson, C. Simons, and C. Eckert. Predicting Change Propagation in Complex Design. *Journal of Mechanical Design*, 126:788, 2004.

[5] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13. IEEE Computer Society Washington, DC, USA, 2003.

[6] A. Hassan and R. Holt. Predicting change propagation in software systems. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 284–293, 2004.

[7] M. Lehman, J. Ramil, P. Wernick, D. Perry, and W. Turski. Metrics and laws of software evolution – the nineties view. In *4th International Software Metrics Symposium (METRICS' 97)*, 1997.

[8] K. J. Lieberherr and I. M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, 1989.

[9] M. Lorenz. Methods of measuring the concentration of wealth. *Publications of the American Statistical Association*, 9(70):209–219, 1905.

[10] A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030, 2006.

[11] D. Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering, ICSE-16*, pages 279–287, 1994.

[12] V. Rajlich. A model for change propagation based on graph rewriting. In *Proceedings of the International Conference on Software Maintenance*, pages 84–91, 1997.

[13] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 167–176, NY, USA, 2005. ACM. ISBN 1-59593-031-0.

[14] D. V. Steward. The design structure system- A method for managing the design of complex systems. *IEEE Transactions on Engineering Management*, 28:71–74, 1981.

[15] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. *SIGSOFT Software Engineering Notes*, 26(5):99–108, 2001.