The Link between Dependency and Cochange: Empirical Evidence

Markus Michael Geipel and Frank Schweitzer

Abstract—We investigate the relationship between class dependency and change propagation (cochange) in software written in Java. On the one hand, we find a strong correlation between dependency and cochange. Furthermore, we provide empirical evidence for the propagation of change along paths of dependency. These findings support the often alleged role of dependencies as propagators of change. On the other hand, we find that approximately half of all dependencies are never involved in cochanges and that the vast majority of cochanges pertain to only a small percentage of dependencies. This means that inferring the cochange characteristics of a software architecture solely from its dependency structure results in a severely distorted approximation of cochange characteristics. Any metric which uses dependencies alone to pass judgment on the evolvability of a piece of Java software is thus unreliable. As a consequence, we suggest to always take both the change characteristics and the dependency structure into account when evaluating software architecture.

Index Terms—Modularity, class dependency, open source

1 Introduction

According to Lehman [30] and common experience, continual change effort is needed to keep a piece of software up-to-date, and Parnas noted that effort is needed to compensate for software "aging" [39]. Therefore, as Bohner and Arnold [3] and Bennett and Rajlich [2] point out, a good software architecture should be evolvable, flexible; in other words, easy to modify. This means that the change behavior of software plays a primordial role. Generally speaking, software is considered flexible if changes are contained to the target module and the rest of the system is *"isolated from change"* [35]. If changes *"propagate"* [41], the architecture is considered suboptimal for several reasons: First, a larger change comprising many modules causes more effort than a small surgical change. Second, a larger change may introduce more defects than a smaller one [35]. And, finally, a newly introduced defect may be harder to fix as each one of the changed files might contain it. Thus, the more modules are affected by change, the harder it becomes narrowing down the problem.

It is important to realize that there is an abundance of best practice rules and common wisdom about the change behavior of software architecture and how to isolate modules from change. In general, simultaneous change of several modules, in other words, *cochange* [22], is believed to arise when modules exhibit strong coupling, for instance, if responsibilities are not clearly separated or implementation details are not hidden behind interfaces. See Martin [35] for a comprehensive overview.

• *M.M. Geipel and F. Schweitzer are with the Chair of Systems Design, ETH Zurich, Kreuzplatz 5, CH-8032 Zurich, Switzerland. E-mail: markus.geipel@alumni.ethz.ch, fschweitzer@ethz.ch.*

Manuscript received 15 June 2010; revised 25 July 2011; accepted 19 Aug. 2011; published online 31 Aug. 2011.
Recommended for acceptance by H. Gall.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-06-0182.
Digital Object Identifier no. 10.1109/TSE.2011.91.

Given this concept of strong coupling, it does not come as a surprise that works on change and change propagation regularly emphasize the role of dependencies, which are seen as measurable manifestations of this coupling. Tsantalis et al. [45], for instance, call certain types of dependencies between classes *axes of change*. Sangal et al. [42] point out that a high concentration of dependencies acts as a propagator of change. They warn that—*"[c]hange propagators make systems brittle because they increase the likelihood that the effect of a change will propagate to a disproportionately large portion of the system."* Finally, MacCormack et al. [33] go so far as to present a method to calculate the so-called *Propagation Cost* of a software based on the dependency structure.

These examples indicate that several authors assume—in some cases implicitly—two issues:

1. *Changes propagate from one module to another along dependencies between these modules.* Without this assumption, analyzing the dependency structure in the context of cochanges would not make sense.
2. *All dependencies are more or less comparable in their propensity to propagate change.* Only with this assumption, it makes sense to calculate measures such as "propagation cost" based on the dependency structure, as done by MacCormack et al. [33] for instance.

In one sentence: To use the dependency structure as an indicator, predictor, or metric for the cochange behavior of a software architecture, there must be a link between dependencies and cochange, and this link must be more or less linear. Yet, to our knowledge, this assumption has never been the dedicated subject of any large-scale analysis, except in the form of preliminary results we presented in [17].

A review of previous research on cochange or change propagation with an empirical framing reveals two aspects. First, the question about the causes of change propagation has been skipped by many researches in favor of a predictive approach in which the causes of change

propagation are implicitly contained in a prediction function or as inputs to a machine learning algorithm. Examples for predictive approaches include: Hassan and Holt [22], Tsantalis et al. [45], Ying et al. [47], Zimmermann et al. [48]. Second, there are two abstraction levels targeted by researchers: We can mainly differentiate between fine-grained analysis (e.g., [48], in the tradition of program slicing [14], [24]), targeting single functions or even statements in the code, and coarse-grained analysis, having entire classes or files as their subject (e.g., [22], [29], [33], [42], [44], [45], [47]).

In respect to these different research lines, the present paper is positioned as follows: We add to the discussion on the causes of cochange directly. We are not concerned with the prediction of future changes or cochanges by means of machine learning. We decided to set this focus because we believe that the mechanism of change propagation is as yet not sufficiently established, as we pointed out in the previous paragraphs. Furthermore, we follow the line of coarse-grained analysis for mainly two reasons: Our motivation is the quality of the software architecture; thus, we focus on the mechanisms which spread change across the entire architecture. We do not extend our analysis to the ones at work within single source files. Our aim is to keep our model simple and focused. Consequently, we define dependencies as follows: There is a directed dependency between module A and B if A depends on B in such a way that A is not operational without module B. In the case of Java, this means that A would not compile in the absence of B. We look at the details in Section 3.1.

In the following section, we condense the just-presented discussion to research questions, and explain why we attribute importance to them. Section 3 presents the empirical approach with which we address these questions. The results are presented in Section 4, followed by a detailed discussion in Section 5. Finally, conclusions are drawn in Section 6.

2 Research Questions

If changes propagate along dependencies, we should be able to make two observations: First, if code modules are linked by dependency, they should be linked by cochange significantly more often than unconnected modules. More formally, this means that the probability that two modules are subject to cochanges given that they are dependent should be higher than the respective probability for independent modules. The first research question is whether this is really the case: *Are dependent modules more likely subject to cochange than independent ones?*

The second observation concerns the propagation along a path of dependencies. It is often assumed (e.g., [33], [42], [45]) that a change may cause a cascade of subsequent changes traveling through the dependency network. In this case, not only direct dependencies matter but also indirect ones: Thus, if A depends on B and B on C, A depends indirectly on C with distance 2. In this case, we should be able to observe a decreasing probability of cochange with increasing distance between the modules in the dependency graph. The second research question is thus: *How does the probability of cochange change with distance in the dependency graph?*

Finally, let us take a closer look at the second assumption formulated in the previous section: In the absence of further knowledge, all dependencies are assumed to be equal. Yet, to our knowledge no large-scale empirical study has ever confirmed this. Thus, the final question we investigate in this paper is the following: *Are dependencies homogeneous in their correlation with co-changes?* The answer 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 [31] suggest. On the other hand, if only specific dependencies matter, general dependency minimization is not a sensible approach. It can even be counterproductive: The simplest way to eliminate a dependency would be to integrate the functionality a class depends upon into the class itself. This would lead to classes with low cohesion and a vague responsibility profile. Both properties to be avoided [35]. Refactoring should go the opposite way: The responsibilities are to be divided, each implemented in a separate class. This would actually generate dependencies; weak ones though, as opposed to strong ones. A clear separation of responsibilities also enhances readability of the code. Furthermore, these new dependencies are a sign of code reuse, a cornerstone of good software design (see Hunt and Thomas [25, ch. 2]). Finally, we cannot per se assume that propagated changes are evenly distributed. In software engineering, as in other disciplines, Pareto's rule—also known as the 80-20 rule—is a recurrent theme. For example, as a rule of thumb, 80 percent of the defects are concentrated in only 20 percent of the code.[1]

In the same way, change propagation might not be distributed uniformly over the set of dependencies. If we find a high concentration, this has consequences for refactoring, the process of restructuring code [38]. If we are able to identify change propagation "hot spots," refactoring effort can be targeted more efficiently.

3 An Empirical Approach

To answer the questions presented in the previous section, we propose an empirical approach based on data from 35 large Java projects. This section documents this approach, starting with a definition of dependency, followed by the methods by which dependency and cochanges are extracted from the data. Next, the statistical analysis designed to answer the research questions is presented. The section concludes with a description of the data sources.

3.1 Class Dependencies

As already pointed out in the introduction, we take a coarse-grained approach to dependency. By doing this, we follow the example of Sullivan et al. [44], Sangal et al. [42], MacCormack et al. [33], and LaMantia et al. [29]. All these authors focused on the file level to calculate dependency. They followed the Design Structure Matrix methodology. The analysis of software dependencies is, however, not bound to this methodology and various representations of the concept of dependency are used in the literature. Challet and Lombardoni [7], for instance, use a graph notation and

1. New empirical support was published by Fenton and Ohlsson [10].

```
1  public class SingleUserRatingInfo extends RatingInfoList {
2    private final TOTorrent torrent;
3
4    public SingleUserRatingInfo(TOTorrent torrent) {
5      this.torrent = torrent; }
6      [...]
7    }
```

Fig. 1. The source code of the class `SingleUserRatingInfo` from the Azureus project. ([...] omitted parts).

refer to the dependencies as dependency network, while Hassan and Holt [22], employing basically the same methodology, use the term call graph.

As we use Java programs as our data source, we speak of class dependencies instead of file dependencies. As in Java, public classes are defined in separate files, one file per class; the conceptual connection to the just cited publications is very strong.

How can the network of class dependencies be represented? Two mathematically equivalent notations are at our disposition: the graph notation and the adjacency matrix notation. In this paper, we stick to the adjacency matrix view, staying in line with the Design Structure Matrix community. The adjacency matrix view is the most widely used notation in this specific area. See the work of Steward [43] for an introduction to the Design Structure Matrix methodology and Sullivan et al. [44] for its application to software. Furthermore, we think that the adjacency matrix notation results in more readable mathematics.

We refer to the dependency matrix as D, where $D_{i,j} = 1$ 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: 1) i extends or implements j, 2) i uses j as member or variable, 3) i references members or calls a method of j. In each of these cases, we set $D_{i,j} = 1$. Please note that D is an asymmetric matrix. To add more discriminatory power to this approach, we differentiate between these three types of dependencies by using a subscript to D: D_a, D_b, and D_c. D_a refers only to the dependencies generated by extension or implementation, D_b the ones where j is used as a member of i, and finally, D_c refers to dependencies caused by all other references to j.

Please note that these dependencies on the class level are different from dependencies on the statement level which are the target of analysis, for instance, in program slicing [14], [24]. Also, the differentiation of statement-level dependencies into data and control dependencies [23] does not apply to class level dependencies.

To illustrate the extraction of dependencies from the source code, we take a closer look at the class `SingleUserRatingInfo` contained in the Azureus project. Fig. 1 shows a shortened version of the class. `SingleUserRatingInfo` extends the functionality of `RatingInfoList` (line 1) and thus is dependent on it. This dependency is recorded in D and D_a. Furthermore, `SingleUserRatingInfo` uses the class `TOTorrent` (lines 3, 5, and 6), another dependency, this time recorded in D and D_c. On the other hand, `SingleUserRatingInfo` is used by the class `PlatformRatingMessenger` (source not listed here), meaning that `PlatformRatingMessenger` is dependent on `SingleUserRatingInfo`. To perform this dependency extraction automatically, we used the Eclipse Java development tools (JDT), as did Gall et al. [13] before us.

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. Furthermore, the fact that the currently existing nodes are of different age does not interfere with our analysis. The only serious issue is 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 3.6.

3.2 Cochange

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 cochanges. 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 the 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 *cochange* (see [22]).

To construct C, we need: first, the set of classes. Let us use n to denote their number. Second, we need 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:

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

H is of size $n \times m$. By multiplying H with its transposed H^T the cochange 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. The diagonal $\mathrm{diag}(C)$ tells us how often each element was changed in total, while the sum of the diagonal $\sum \mathrm{diag}(C)$ gives us the total number of changes to the system. An entry $C_{i,j} = 3$ tells us that classes i and j have been modified three times together. Please note that C in contrast to D is symmetric.

Until now, our definition of cochange has been abstract. We now define it in the context of our change data, which consists of CVS logs (see also Section 3.7): A cochange event comprises *all classes* whose changes have been committed at exactly the *same time* by exactly the *same* author. The same

definition has been used by Ball et al. [1], although they used the term *Modification Record* instead of cochange. It is important to note that we only consider *change* events and not *add* events. The reason for this is that projects often start outside of a version control system and are initially imported in one *add* event. Even later, occasionally whole subsystems are added at once. We argue that such creation cochanges do not provide information on the evolvability of the software and thus exclude them.

We argue that the just presented commit-based cochange construction reflects the real cochange relationships rather accurately for the following reason: There is psychological pressure to make one cohesive set of changes one and only one commit. First, there is pressure not to split the cohesive set of changes into more than one commit as a split would generate an inconsistent version in the repository. A common rule in development, though, is to avoid an inconsistent commit at all costs because it may seriously impede the work of the group. Some groups even collect symbolic fines for contributors who cause a nightly built to fail. Second, there is pressure to put as few changes as possible in one commit as a commit should ideally comprise a small, self-contained, easily described fix or contribution. Furthermore, frequent commits prevent merge conflicts. In the Open Source community, this is referred to as the "Commit early, commit often!"-rule.

It should not be left unmentioned though that alternative ways of cochange calculation exist. A more complex alternative to the method just described was introduced by Zimmermann et al. [48]. They propose a sliding window such that two classes are connected if they have been modified by the same author within a time window of Δt seconds. The motivation behind this approach is that the commit-based method does not count changes as cochanges which are temporally very close but do not belong to the same commit. For the same reason, German [18] extended the method of Ball et al. [1] with two tunable parameters.

For predictive works, for instance, the sliding window approach is a very sensible approach as the parameter Δt can be optimized based on prediction success. In our case, however, we do not have such a benchmark as our work is exploratory. See also the discussion in the introduction. We thus explicitly decided against the sliding window approach as it introduces a new degree freedom, Δt (or even more parameters). Its concrete value is subject to discussion, and argues for a specific value rest subjective or bound to situation specific optimization. We did not find convincing arguments for a universally valid value of Δt. Following Occam's Razor, we thus favor the simpler solution even if it might introduce a bias. In any case, the sliding window parameter Δt would, depending on its value, also introduce a bias. We discuss possible problems caused by such a bias in detail in Section 5.1.

3.3 Dependencies and Cochange

To explore the connection between dependency and change propagation, we need to compare the change behavior 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 $C_{i,j} \geq 1$. Furthermore, the fact that two classes are connected by a dependency is expressed by $D_{i,j} = 1$. A straightforward 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 been modified together at least once, given that they are connected by a dependency. P_D is calculated from the data as follows:

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

This means that we divide the number of dependencies (positive entries in the dependency matrix) which accumulated at least one cochange by the total number of dependencies. In the same way, we define P_{D_a}, P_{D_b}, and P_{D_c}.

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 been modified together at least once. $P_{\neg D}$ is calculated as follows:

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

This means that we divide the number of empty entries in the dependency matrix for which there exists a corresponding positive entry in the cochange matrix by the total number of empty entries in the dependency matrix. The comparison of P_D with $P_{\neg D}$ reveals the possible degree of correlation between dependencies and cochange. A comparison of P_{D_a}, P_{D_b}, and P_{D_c} enables us to check whether the different dependencies play different roles with respect to the cochange behavior.

To compute error margins for the empirical measures, we assume a binomial distribution. Let us consider P_D: To calculate P_D, we iterate through all dependencies, sum up the ones which where involved in a cochange, and take the average. This corresponds to process behind the binomial distribution where the positive outcomes of n independent yes/no experiments are summed up and averaged (for more information, see [37, Section 1.3.6.6.18.]). This means that the binomial distribution will provide more accurate error estimates than the normal distribution, which would be the standard choice in the absence of further knowledge. Please note that in any case, for large n, the normal distribution approximates the binomial distribution.

3.4 Indirect Dependencies and Cochange

P_D indicated the probability that two-dependent classes change at least once together (cochange) in the project history. We will now investigate the existence of changes transitively propagating along dependencies. In other words: Do indirect dependencies matter? as we asked in research question two. To address this question, let us extend the concepts of Sections 3.1 and 3.3 to indirect dependencies. Let $D^l_{i,j} = 1$ if a path of length l exists between two classes i and j. With the concept of D^l, we extend P_D to P_{D^l}, giving the probability that two classes connected by a dependency path[2] of length l are changed together at least once.

2. In practice, the distances can be calculated by any algorithm calculating the convex hull of a graph. A standard algorithm is the Floyd-Warshal algorithm [11], [46]. A discussion of the pros and cons of convex hull algorithms is beyond this paper. See, for instance, [20] for more information.

If cochanges propagate along a path of dependency, which function in l would P_{D^l} delineate? Leaving apart parallel paths of change propagation, we can assume exponential decay: $(P_D)^l$. Yet, we need to keep in mind that change may also propagate between independent classes: $P_{\neg D}$ may be nonzero. We thus add an intercept c to the equation. Consequently, the most simple approximation for P_{D^l} is

$$P_{D^l} = (P_D)^l + c. \tag{5}$$

This equation can be checked against data: We fit the empirically measured P_{D^l} to (5) with the ordinary least squares method. To check whether the model explains the patterns found in the data, the quality of the fit needs to be quantified.

Different measures for the quality of fit exist. We are considering as candidates the standard measures offered by the statistics toolbox of the Matlab software:

1. Sum of squares due to error (SSE): Measures the total deviation of the data from the fit.
2. Coefficient of determination (R^2): Measures the proportion of variability in a data set that is accounted for by the model. The closer R^2 is to 1, the better the fit.
3. Degree-of-freedom adjusted coefficient of determination (adj. R^2): R^2 adjusted to account for the residual degrees of freedom (number of observations minus the number of fitted coefficients).
4. Root mean squared error (standard error): Estimate of the standard deviation of the residuals.

We decided to use adj. R^2 for the following reasons: First, the R^2 are bound between -1 and 1, which makes a comparison between the different projects easy. The other measures are unbound and values are thus harder to interpret and compare in a list overview such as the one we present. Second, the concept of proportion of variability explained by the model fits our intent to show the expedience of the proposed model best as it basically gives its "explanatory power" in percent. Finally, standard error and SSE can only be interpreted in a comparative way: to compare different models, for instance. But this is not our objective. We are only interested in the explanatory power of one model.

We put forth the proposition that good fits between the data and expression (5)—values of adj. R^2 close to 1—provide empirical evidence for the existence of change propagation along paths of dependencies and thus indirect dependencies would matter.

3.5 Homogeneity of Dependencies

In 1905, the economist Max Otto Lorenz proposed a concise method to analyze and visualize income inequalities [32]. Since then, the so-called Lorenz curve has been used to describe concentration and inequality in various contexts. In this paper, we use it to analyze the concentration of propagated changes, and thus answer our third research question. In the following paragraphs, we briefly explain how the Lorenz curve is constructed from our data. For a more in-depth description of Lorenz curves, we refer the reader to the original work of Lorenz [32] or Gastwirth [15].

To begin with, we need the number of propagated changes for each single dependency. Let us refer to this set as $\mathbf{\Pi}$:

$$\mathbf{\Pi} = \{C_{i,j} : D_{i,j} = 1\}. \tag{6}$$

In this case, we considered all dependencies. Alternatively, dependencies with no observed cochanges can be excluded. The resulting $\mathbf{\Pi}'$ is defined by

$$\mathbf{\Pi}' = \{C_{i,j} : D_{i,j} = 1 \wedge C_{i,j} > 0\}. \tag{7}$$

The next steps are equally valid for $\mathbf{\Pi}$ and $\mathbf{\Pi}'$ even though we only write $\mathbf{\Pi}$.

First, $\mathbf{\Pi}$ is normalized such that the entries of the result π sum up to one:

$$\pi_k = \Pi_k \Big/ \sum_{l=1}^{|\mathbf{\Pi}|} \Pi_l. \tag{8}$$

Second, the entries of π are rearranged in ascending order such that the following constraint is satisfied:

$$r < s \Rightarrow \pi_r \leq \pi_s. \tag{9}$$

Finally, the Lorenz curve $\mathrm{L}(x)$ with $x \in [0, 1]$ is calculated by cumulating the first x percent of the elements of $\boldsymbol{\pi}$:

$$\mathrm{L}(x) = \sum_{k=0}^{\lfloor x*|\boldsymbol{\pi}| \rfloor} \pi_k. \tag{10}$$

$\mathrm{L}(x)$ is interpreted as follows: The x percent least active dependencies accumulate $\mathrm{L}(x)$ percent of the propagated changes; the x percent most active dependencies are associated with $1 - \mathrm{L}(1 - x)$ 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 $\mathrm{L}(x) = x$. In a plot, this results in a straight diagonal line (line of equality) and serves as a reference. Any unequal distribution results in a convex curve. The higher the concentration or inequality is, the higher the curvature and thus the deviation from the diagonal line.

While the Lorenz curve is a very instructive representation of concentration, we cannot show the respective curve of each one of the 35 projects. To compress a Lorenz curve to one number, Gini [19] introduced the so-called Gini-Coefficient g:

$$g = 1 - 2 \int_0^1 L(x)dx. \tag{11}$$

g ranges from 0 to 1. As mentioned previously, an equal distribution leads to a straight diagonal line. In this case, $g = 0$. The maximum concentration on the other side is indicated by $g = 1$. As the Gini-Coefficient represents concentration as one scalar value, it also enables us to compare different concentrations.

3.6 Excluding the Effect of Time

Concentration of cochanges is per se not an indication that dependencies are heterogeneous concerning their change propagation. It can be assumed that older dependencies had more time to propagate change. A system in which dependencies of different age are present will thus necessarily exhibit an uneven distribution of cochanges.

TABLE 1
Summary of Statistical Analysis

project date of analysis	$\mathbf{P}_D$	$\mathbf{P}_{D_a}$	$\mathbf{P}_{D_b}$	$\mathbf{P}_{D_c}$	$\mathbf{P}_{\neg D}$	$\mathbf{g}$	$\mathbf{g^*}$	$\mathbf{adj.R^2}$
architecturware (2008/02/04)	$33.2_{\pm 1.3}$	$42.7_{\pm 3.2}$	$33.8_{\pm 4.0}$	$30.6_{\pm 1.5}$	$0.5_{\pm 0.1}$	0.79	0.26	0.99
aspectj (2008/02/01)	$57.8_{\pm 1.6}$	$76.0_{\pm 4.3}$	$60.5_{\pm 5.7}$	$55.2_{\pm 1.8}$	$18.5_{\pm 0.1}$	0.66	0.21	0.83
azureus (2008/01/01)	$38.2_{\pm 1.3}$	$45.6_{\pm 3.6}$	$38.2_{\pm 3.2}$	$36.9_{\pm 1.5}$	$1.2_{\pm 0.1}$	0.78	0.21	0.99
cjos (2008/02/04)	$36.3_{\pm 1.1}$	$49.3_{\pm 2.2}$	$31.9_{\pm 3.0}$	$31.4_{\pm 1.4}$	$0.2_{\pm 0.1}$	0.74	0.20	0.96
composestar (2008/07/04)	$49.0_{\pm 2.0}$	$68.3_{\pm 4.8}$	$57.8_{\pm 5.0}$	$43.1_{\pm 2.3}$	$1.7_{\pm 0.1}$	0.68	0.47	0.89
diee-mad (2008/07/04)	$23.0_{\pm 2.4}$	$33.0_{\pm 7.4}$	$34.2_{\pm 9.9}$	$20.3_{\pm 2.6}$	$1.1_{\pm 0.1}$	0.85	0.19	0.98
eclipse (2008/03/01)	$21.5_{\pm 0.3}$	$38.6_{\pm 1.1}$	$25.4_{\pm 1.0}$	$19.1_{\pm 0.3}$	$0.3_{\pm 0.1}$	0.85	0.36	0.93
enterprise (2008/02/04)	$16.5_{\pm 1.3}$	$2.1_{\pm 1.8}$	$26.6_{\pm 6.2}$	$17.6_{\pm 1.5}$	$0.5_{\pm 0.1}$	0.85	0.07	0.99
findbugs (2008/02/04)	$21.1_{\pm 0.7}$	$33.7_{\pm 2.1}$	$18.5_{\pm 1.8}$	$19.2_{\pm 0.8}$	$0.6_{\pm 0.1}$	0.85	0.16	0.95
fudaa (2008/07/01)	$34.5_{\pm 0.9}$	$48.8_{\pm 2.4}$	$37.3_{\pm 2.1}$	$31.1_{\pm 1.0}$	$1.6_{\pm 0.1}$	0.80	0.37	0.96
gpe4gtk (2008/07/04)	$28.9_{\pm 1.4}$	$40.1_{\pm 4.2}$	$34.3_{\pm 4.1}$	$26.0_{\pm 1.6}$	$1.1_{\pm 0.1}$	0.83	0.28	0.97
hibernate (2008/02/04)	$45.6_{\pm 1.3}$	$72.9_{\pm 3.1}$	$56.9_{\pm 4.1}$	$39.4_{\pm 1.5}$	$1.1_{\pm 0.1}$	0.77	0.26	0.98
jaffa (2008/01/28)	$29.5_{\pm 1.5}$	$28.3_{\pm 4.5}$	$28.0_{\pm 4.0}$	$29.9_{\pm 1.7}$	$0.9_{\pm 0.1}$	0.80	0.03	1.00
jazilla (2008/01/28)	$62.0_{\pm 1.6}$	$75.2_{\pm 3.0}$	$56.1_{\pm 4.4}$	$58.8_{\pm 2.0}$	$7.1_{\pm 0.1}$	0.42	0.28	0.98
jedit (2008/02/28)	$58.0_{\pm 1.8}$	$76.0_{\pm 4.0}$	$57.9_{\pm 4.4}$	$54.1_{\pm 2.2}$	$1.2_{\pm 0.1}$	0.61	0.17	0.50
jena (2008/02/01)	$33.0_{\pm 1.1}$	$65.6_{\pm 3.0}$	$29.6_{\pm 3.4}$	$28.0_{\pm 1.2}$	$1.0_{\pm 0.1}$	0.81	0.15	0.95
jmlspecs (2008/01/28)	$48.2_{\pm 1.9}$	$80.1_{\pm 5.0}$	$50.5_{\pm 6.7}$	$44.5_{\pm 2.1}$	$3.1_{\pm 0.1}$	0.71	0.18	0.15
jnode (2008/02/03)	$28.0_{\pm 0.8}$	$43.4_{\pm 2.2}$	$29.8_{\pm 2.2}$	$24.6_{\pm 0.9}$	$0.5_{\pm 0.1}$	0.83	0.32	0.97
jpox (2008/01/28)	$35.1_{\pm 1.2}$	$57.1_{\pm 3.4}$	$41.9_{\pm 3.9}$	$30.5_{\pm 1.4}$	$0.8_{\pm 0.1}$	0.82	0.28	0.97
openhre (2008/02/05)	$34.6_{\pm 2.0}$	$55.8_{\pm 4.0}$	$25.4_{\pm 5.5}$	$27.8_{\pm 2.3}$	$1.1_{\pm 0.1}$	0.66	0.03	0.98
openjacob (2008/07/01)	$24.0_{\pm 1.3}$	$39.5_{\pm 3.6}$	$31.0_{\pm 5.1}$	$20.5_{\pm 1.3}$	$1.2_{\pm 0.1}$	0.82	0.11	0.94
openuss (2008/07/01)	$26.8_{\pm 1.5}$	$29.0_{\pm 3.5}$	$34.7_{\pm 5.2}$	$25.2_{\pm 1.7}$	$1.6_{\pm 0.1}$	0.80	0.07	0.88
openxava (2008/02/04)	$28.0_{\pm 1.8}$	$28.9_{\pm 3.8}$	$23.7_{\pm 4.4}$	$28.7_{\pm 2.3}$	$2.0_{\pm 0.1}$	0.91	0.06	0.99
pelgo (2008/07/01)	$18.1_{\pm 1.3}$	$41.1_{\pm 4.6}$	$23.7_{\pm 4.0}$	$13.3_{\pm 1.3}$	$0.3_{\pm 0.1}$	0.82	0.01	1.00
personalaccess (2008/07/04)	$29.8_{\pm 1.9}$	$36.5_{\pm 5.3}$	$40.5_{\pm 5.2}$	$26.2_{\pm 2.2}$	$1.3_{\pm 0.1}$	0.83	0.16	1.00
phpeclipse (2008/07/04)	$49.5_{\pm 1.9}$	$70.6_{\pm 4.9}$	$45.5_{\pm 5.1}$	$46.8_{\pm 2.2}$	$2.0_{\pm 0.1}$	0.66	0.13	0.94
rodin-b-sharp (2008/07/04)	$21.8_{\pm 1.0}$	$47.0_{\pm 3.3}$	$18.8_{\pm 3.1}$	$18.2_{\pm 1.0}$	$0.5_{\pm 0.1}$	0.88	0.32	0.96
sapia (2008/07/01)	$42.4_{\pm 1.7}$	$43.1_{\pm 3.9}$	$45.2_{\pm 4.9}$	$41.7_{\pm 2.0}$	$1.1_{\pm 0.1}$	0.73	0.31	1.00
sblim (2008/07/01)	$12.7_{\pm 0.6}$	$25.0_{\pm 2.3}$	$28.6_{\pm 3.1}$	$10.1_{\pm 0.6}$	$1.5_{\pm 0.1}$	0.92	0.04	0.63
springframework (2008/02/03)	$33.6_{\pm 1.2}$	$50.6_{\pm 2.9}$	$35.8_{\pm 4.0}$	$29.3_{\pm 1.3}$	$0.5_{\pm 0.1}$	0.80	0.16	1.00
squirrel-sql (2008/07/04)	$27.4_{\pm 1.5}$	$34.7_{\pm 4.2}$	$28.9_{\pm 3.0}$	$25.3_{\pm 1.8}$	$1.2_{\pm 0.1}$	0.82	0.08	1.00
university (2008/07/01)	$26.8_{\pm 1.6}$	$38.6_{\pm 4.1}$	$12.0_{\pm 2.6}$	$28.6_{\pm 2.1}$	$4.0_{\pm 0.1}$	0.73	0.02	0.58
xendra (2008/02/04)	$19.0_{\pm 1.3}$	$23.0_{\pm 3.3}$	$10.9_{\pm 2.2}$	$20.9_{\pm 1.8}$	$1.5_{\pm 0.1}$	0.81	0.19	0.75
xmsf (2008/07/04)	$34.8_{\pm 1.6}$	$29.1_{\pm 3.6}$	$46.4_{\pm 4.7}$	$34.3_{\pm 1.9}$	$2.4_{\pm 0.1}$	0.82	0.13	0.97
yale (2008/02/01)	$31.5_{\pm 1.5}$	$64.2_{\pm 4.1}$	$46.8_{\pm 5.6}$	$24.6_{\pm 1.5}$	$2.6_{\pm 0.1}$	0.77	0.06	0.98
mean	33.16	46.68	35.63	30.34	1.94	0.78	0.18	0.90
std. dev.	12.15	18.35	13.05	11.65	3.16	0.09	0.11	0.18

For the definitions and information on the error margins, see Section 3.

This means that we cannot compare the Lorenz curve found in the data with the line of perfect equality. Instead, as the reference curve, we use the hypothetical concentration which would be caused by the existing age distribution under the assumption of homogeneous change propagation. This homogeneous change propagation is the implicit assumption of any method inferring change behavior from the dependency network only, as pointed out in Section 1.

Let us assume that dependencies accumulate cochanges linearly in time. This means that the expected number of cochanges c depending on age t of the dependency can be expressed by the following function:

$$c(t) = b_1 + b_2 t. \tag{12}$$

b_1 and b_2 are constants specific to a given project. Next, the average number of cochanges 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 cochanges each of the dependencies in the data would exhibit if the earlier discussed homogeneity of change propagation assumption held true. From this hypothetical cochange distribution, the hypothetical Lorenz curve $\mathrm{L}^*(x)$ is compiled. $\mathrm{L}^*(x)$ serves a reference, giving the concentration of cochanges, the system would exhibit if age was the only difference between the dependencies. If $\mathrm{L}(x)$ has a stronger bent than $\mathrm{L}^*(x)$, cochanges concentrate more than we would expect based on the existing age differences. To facilitate the comparison of $\mathrm{L}(x)$ and $\mathrm{L}^*(x)$, we calculate the respective Gini-Coefficients g and g^*. Similar values of g and g^* imply that dependencies are homogeneous in their influence on cochanges. If g is significantly larger than g^*, dependencies are heterogeneous and consequently the cochange behavior is not a mirror image of the dependency structure.

3.7 Data Sources

For several reasons, our study focuses on Java projects: 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: On SourceForge (http://www.sf.net)—the largest Open Source incubator site—Java is used in approximately 25 percent of the projects (in August 2007). This makes Java the most popular language used.

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 could, at the moment of writing, be found at http://NAME.sf.net, where `NAME`

stands for the name of the project. They were selected as follows: We took the 36 largest[3] 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 taking into account their size. They have been developed outside SourceForge and were later just copied to SourceForge.

Finally, the set of SourceForge projects was complemented with two further projects: AspectJ (see http://www.eclipse.org/aspectj/) and Eclipse,[4] both hosted by IBM. This makes, all in all, 35 projects. We added Eclipse and AspectJ for two reasons: first to alleviate the bias toward only one hosting site and second, to also include projects which are developed mainly by a firm (IBM). As we argue in Section 5.1 in more detail, adding such projects to the analysis alleviates the bias toward noncommercial Open Source software.

The data used in this study is available online: http://www.sg.ethz.ch/research/social_organizations/structure_and_dynamics_of_open_source_software.

4 Results

In this section, we present and interpret the empirical results of the analysis described in the previous section. The structure of this section parallels the list of research questions presented in Section 1.

4.1 Dependencies and Cochange

As pointed out in Section 3.3, the probabilities P_D and $P_{\neg D}$ provide evidence for the influence dependencies exert on the cochange behavior.

Table 1 (columns one and five) shows the empirical values of P_D and $P_{\neg D}$ for our sample set of 35 projects. P_D ranges between approximately 15 and 60 percent, $P_{\neg D}$ is less than 5 percent for all projects except the project `aspectj`.[5] 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 change in the entire development history propagated. This is an important point which we will revisit in the discussion on homogeneity of dependencies (Section 4.3).

Let us now differentiate between the three types of dependencies considered. Columns two to four depict the empirical values of P_{D_a}, P_{D_b}, and P_{D_c}. It can be seen that dependencies generated by inheritance or extension (P_{D_a}) exert the strongest influence on the cochange dynamics, followed by dependencies generated by a part-of relationship (P_{D_b}). The question is: "Does this difference matter?"

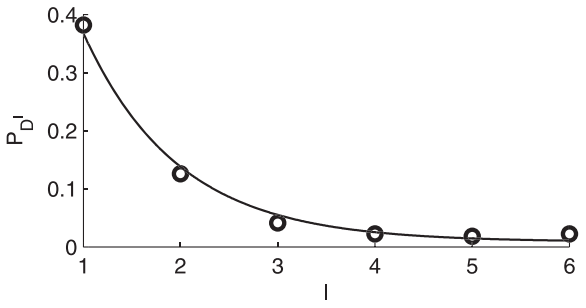

Fig. 2. Share of active dependencies depending on distance.

We first approach this question with statistics. Normally, a T-Test would tell us whether the means of the distributions P_{D_a}, P_{D_b}, and P_{D_c} differ in a statistically significant way. We would need to assume the data to be normally distributed though. However, we do not even know if they are at least from the same distribution. Thus, we decided to conduct a two-sample Kolmogorov-Smirnov test (two-sample KS-test) to check whether the distributions are similar. A two-sample KS-test is nonparametric and distribution free, which means that we do not have to make assumptions about the distribution of the data (not to be confused with the one-sample KS-test). The null hypothesis is that two data sets (for instance, P_{D_a} and P_{D_b}) are from the same continuous distribution. The alternative hypothesis is that they are from different continuous distributions.

The results from the two-sample KS-test support the hypothesis that the three respective distributions are distinct. The statistical significance level reached is 1 percent, meaning that the probability of a false negative for the given test and data is 1 percent.

Nonetheless, as Carver [5], [6] points out, one should not overestimate statistical significance. Statistical tests are not a substitute for common sense: Not every difference that is statistically significant is necessarily important in practice. Looking at the three distributions not from a statistician's point of view but from a software engineer's point of view, we doubt that the difference between P_{D_b} and P_{D_c} plays a role in practice. The difference between P_{D_a} and P_{D_b}/P_{D_c} seems more relevant. It means that class inheritance generates significantly stronger dependencies between classes than any other form of dependency such as call or composition.

Coming back to our first research question, "Are dependent modules more likely subject to cochange?", we can conclude that the pronounced difference between P_D and $P_{\neg D}$ supports the view of dependencies as propagators of change.

4.2 Indirect Dependencies and Cochange

In Section 3.4, we argued that if indirect dependencies matter and change propagation along a path of dependencies exists, the relation between the length l of this path and the probability of a cochange occurring at least once in the project history P_D^l should fit approximately $(P_D)^l + c$.

Fig. 2 shows the fit for the Azureus project. It can be seen that in the case of Azureus the empirical data indeed follow the theoretical model. And, the Azureus project is no

3. The size was measured as the number of Java classes in the code repository.

4. See http://www.eclipse.org/ for further information and dev.eclipse.org:/cvsroot/eclipse for the exact source we used in the analysis.

5. `Aspectj` is a special case as many classes in its software repository make use of aspect-oriented programming [28]. This means that many dependencies are generated by the aspect weaver. Such dependencies are invisible in our analysis. We assume that these "invisible" dependencies are responsible for the high value of $P_{\neg D}$ in `aspectj`.

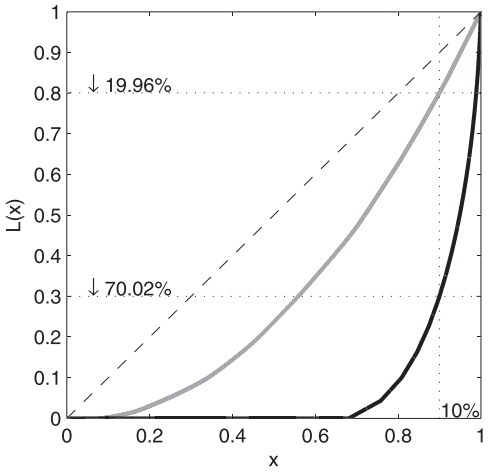

Fig. 3. Concentration of cochanges. Solid black line: The Lorenz curve $\mathrm{L}(x)$ for the distribution of cochanges among all dependencies in Eclipse (see (6)). Solid gray line: The Lorenz curve $\mathrm{L}^*(x)$ which would be generated by the age differences among dependencies only (see Section 3.6). The dashed line marks the line of perfect equality.

exception: Table 1 lists the goodness of the fit (adjusted R^2) for all 35 projects.

As indicated by the high values of R^2, the model describes the empirically measured P_{D^l} quite well for the majority of the analyzed project. Twenty-nine fits out of 37 have an adjusted R^2 larger than 0.9, meaning that more than 90 percent of the variance in the empirically measured $P_{D^l}b$ is explained by the model.

This high level of congruence between the model and the data is evidence for change propagation along a path of dependencies and answers our second research question: "How does the probability of cochange change with distance in the dependency graph?" Consequently, this means that indirect dependencies are also important and should be considered when modeling cochanges.

4.3 Heterogeneity in the Cochange Distribution

In Section 3.5, we presented the tools to quantify the concentration of cochange and thus check the homogeneity assumption. The results are now presented in two steps: First, we focus on a single project, Eclipse, and second, we take a broader look at the whole sample of 35 projects.

Fig. 3 shows in black the Lorenz curve for the Eclipse project considering all dependencies, according to (6). 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 percent of the dependencies are responsible for over 70 percent of the change propagation along dependency links. This result is based on approximately 235,000 dependencies and nearly 284,000 cochanges.

The reference curve $\mathrm{L}^*(x)$ is shown in gray. It indicates the hypothetical cochange distribution resulting from an architecture with homogenous change propagation but the same age distribution as the real-world architecture (see Section 3.6 for the definition). $\mathrm{L}^*(x)$ clearly shows less concentration than $\mathrm{L}(x)$: It predicts only 19.96 percent of cochanges for the most active 10 percent dependencies. This means that different ages of the dependencies are not a sufficient explanation for the high concentration of cochanges found in the data.

The situation is similar for the other projects. Table 1 lists the Gini Coefficients g for all 35 projects, and throughout the sample the concentration is very high; on average 0.78. Even if we factor in the age differences between classes, the picture does not change: The hypothetical concentration caused by age differences is just 0.18 on average (see g^*).

We conclude that cochanges highly concentrate consistently throughout the entire set of projects. Thus, the homogeneity assumption must be rejected. Our third research question,"Are dependencies homogeneous in their correlation with cochanges?", needs to be answered with a clear *no*.

5 Discussion

We start our discussion by exploring limits of the just presented results as well as possible threads to their validity. Next, implications and future research perspectives are analyzed.

5.1 Limitations and Threads to Validity

In this section, we discuss a number of threads to the validity of our study. The purpose is to give further arguments for our research decisions and to define the limits of this study.

5.1.1 Possible Bias in the Data Selection

As explained in Section 3.7, we tried to select a representative set of projects. Still, it is impossible to make a completely unbiased selection. The first objection suggesting itself is that taking the majority of the projects from the same hosting site might introduce a bias. Nevertheless, we argue that Sourceforge hosts a broad variety of projects and the projects in our selection have no obvious connections with each other. They differ in purpose and involved programmers. A second, more substantial objection is that all projects are Open Source projects and thus it is not sure whether the results also pertain to commercial projects. We counter that there are projects in our selection which are to a large part developed by firms, such as `Eclipse` (IBM). There is one similarity, though, that all these projects share: They are projects developed over years with a community associated with them. The study is missing short-term remittance work for a specific customer. We are not sure, though, whether these kinds of projects are actually valuable for research on software engineering as, by definition, they hardly evolve. Finally, we argue that, provided the results in Table 1 are quite clear and depict a rather homogeneous situation—which is the case—it is our belief that closed source or Open Source projects from other hosting sites exhibit vastly different behavior. Nonetheless, replication of our results for commercial products and further Open Source is essential to establish a solid theory of dependency and change.

5.1.2 Programming Language Differences

What differentiates us from most other authors who worked on dependency structures in software [29], [33], [42], [44] is that we used Java programs instead of C++ programs. There are two reasons for this: First, Java is the

most frequently used language on SourceForge, based on number of projects. Second, Java was designed as an object-oriented language from the beginning, in contrast to many other object-oriented languages which are often just procedural languages with an extension for the Object-Oriented paradigm. We believe that class dependencies can thus be extracted from Java programs with higher accuracy and less ambiguity than from C++ programs for instance. See also Section 3.7 where we argued that this higher accuracy is due to the stringent Object-Oriented design of Java as opposed to C++. In all other aspects, we followed the dependency concept used in the just cited publications.

Given the structural similarity of object-oriented languages, we argue that the general results presented in this paper should pertain to the whole group of Object-Oriented languages. Nonetheless, further studies are needed to confirm this (see also Section 5.2). Clearly, the results presented in this paper are limited to the Object-Oriented paradigm. In procedural languages, the concept of dependency is different and we cannot expect per se dependencies between functions to behave equivalently to class dependencies. Further investigations in this field would be worthwhile.

5.1.3 Possible Errors in the Cochange Calculation

As discussed in Section 3.2, there are mainly two methods to construct the cochange graph: the commit-based one, which we used in this paper, and the sliding window approach. In Section 3.2, we gave a number of arguments for our choice, but also pointed out that both methods may introduce errors. The purpose of this section is to analyze in how far such an error jeopardizes our results.

First of all, what is the nature of the bias for both methods of cochange calculation? In general, we can say that the commit-based method is underestimating the number of cochanges. The sliding window may either over- or underestimate the number of cochanges depending on the value of Δt. The latter would thus make it possible to "tune" the results, a possibility that should be avoided for a empirical study aimed at verifying or falsifying claims. For machine learning algorithms, on the other side, a tunable parameter Δt can be very useful: For prediction problems, we could determine a suitable Δt by minimizing an error function based on prediction errors in a test set. For our study, there is no such error function.

Apart from this consideration, possibly underestimating the number of cochanges might have the following consequences for the validity of the results in this paper: We first asked: Are dependent modules more likely subject to cochange? We found that independent modules have, on average, a probability of 33.16 percent to be involved in a cochange during the life span of a project, unconnected ones a probability of 1.94 percent (Table 1). This is a difference of one order of magnitude. Even if all additionally counted cochanges in a sliding window approach behaved in exactly the opposite way to ones counted by the commit-based approach, the number of cochanges would need to roughly double to invalidate our result. Second, we asked: How does the probability of cochange change with distance in the dependency graph? We found that the function $P_{D^l} = (P_D)^l + c$ is a good approximation of the probability of cochange dependent on distance l in the dependency graph (Table 1). Such a result might be disturbed more easily than the previously discussed one. Nonetheless, also in this case there is no plausible reason for additionally counted cochanges to be vastly different in behavior compared to the ones already counted by the commit-based approach. Finally, we found that cochanges highly concentrate (Fig. 3). To level out this very pronounced concentration (compare g and g^* in Table 1) we would need to double the cochanges and all of them would need to behave in the exact opposite way than the already counted ones, a situation that seems very unlikely. However, this is an aspect for further research.

5.1.4 Directed versus Undirected Dependencies

For the calculation of the conditional probabilities in Section 3.3, we relied on directed dependencies. This means that if class i depends on class j, then $D_{i,j} = 1$ but not necessarily $D_{j,i} = 1$. Alternatively, we could argue that dependencies should be treated as undirected ones. If i depends on j and j is changed, i might need to be modified as well. Equally, one could argue that also a change in i might trigger a change in j. Therefore, we might assume D to be symmetric ($D_{i,j} = D_{j,i}$).

The question is: Did we miss a part of the dependencies by treating them as directed (D unsymmetric)? We will now show that this difference would not qualitatively change our results. First of all, it should be noted that the difference of $P_{\neg D}$ and P_D is very pronounced (see Table 1) and a large distortion would be needed to invalidate our conclusion. To approach the problem in an analytical way, let us look at the expression to calculate the conditional probabilities and how they would change. P_D was defined in (3) as

$$\frac{|\{D_{i,j} = 1 \wedge C_{i,j} \geq 1\}|}{|\{D_{i,j} = 1\}|}.$$

What happens if we make D symmetric? We will roughly double the number of entries holding a 1 in D as circular dependencies are abhorred by software engineers and count among the so-called antipatterns of object-oriented software engineering: "The dependency structure between packages must be a directed acyclic graph (DAG). That is, there must be no cycles in the dependency structure" [34]. Empirical analysis performed by us using the database of this paper showed that direct circular dependencies are rare indeed. Based on this reasoning, we expect the denominator to roughly double. What happens to the numerator? Actually, it will also roughly double because C is already symmetric. The $\wedge$ connector will evaluate the same for the newly added dependencies as it did for their already existing symmetric counterparts. If both the denominator and the numerator roughly double, our results are stable.

The situation is a little bit different for (4). $P_{\neg D}$ was defined as

$$\frac{|\{D_{i,j} = 0 \wedge C_{i,j} \geq 1\}|}{|\{D_{i,j} = 0\}|}.$$

Roughly doubling the dependencies will reduce the number of nondependent class pairs ($|\{D_{i,j} = 0\}|$) by approximately $|\{D_{i,j} = 1\}|$. Please note that $|\{D_{i,j} = 1\}|$ is, in practice, much smaller than $|\{D_{i,j} = 0\}|$. For instance, in the project `azureus`, there are 3,147 classes. There are

17,012 dependencies and thus $3{,}147^2 - 17{,}012 = 9{,}886{,}597$ independent pairs of classes. There is a difference of several orders of magnitude and our results are thus stable.

5.1.5 Correlation versus Causation

It is important to note that a correlation between dependence and cochange is a *necessary* but can never be a *sufficient* condition for causation. If A and B are correlated, we cannot automatically conclude that A causes B. Correlation is symmetric. Thus, it could also be generated by B causing A. Finally, there could be an unobserved C causing both A and B. Also, in this case we would observe a correlation. What does this mean for our study? The common opinion in the literature is that dependencies cause cochange. The reversal seems not very plausible: It is hard to conceive of a mechanism by which the act of changing two classes simultaneously would also introduce a dependency. It is more plausible that the introduction of a dependency causes a cochange. The third possibility that an unobserved C causes both cochange and dependency cannot be excluded though. We could conceive another type of connection or influence, not captured by our definition of dependency, which causes both cochange *and* dependency. Finally, we need to keep in mind that change activity of a developer is a precondition to cochange. Change in software as it is is intrinsically linked to the intention of the developer, which, as are all human intentions, are difficult to quantify or determine (see also [8]). This means that by answering the just-mentioned research questions positively with our empirical analysis, we show that the classically assumed link between dependency and cochange has not been falsified. We do not prove, however, that it exists. In any way, from a philosophy of science point of view, there are convincing arguments that proofs of this kind are not possible at all: For an in-depth discussion see Popper [40].

5.1.6 Possible Errors in the Data

A final thread to the validity of this study is the possibility of erroneous data. The data as described in Section 3.7 consist of dependency relationships and change logs. The parsing of CVS change logs is straightforward and does not require sophisticated algorithms. We do not expect errors here. Furthermore, given the maturity of CVS, we judge the produced logs to be reliable. Still, there might be information missing: Often projects start without version control or not under the version control of SourceForge. This means that we miss events in the beginning of the project. We argue that the information missing is not problematic as the logs used capture, on average, a development history of nearly five years, which we consider a sufficient database for our study.

The construction of the dependency network is more complicated than the change data. As pointed out in Section 3.1, we used the Eclipse Java development tools, which are a core part of the Eclipse platform. Given its widespread use and maturity, we assume correct functioning. Yet, we cannot entirely rule out the possibility of bugs in code we wrote ourselves. Manual inspection did not reveil any problems. However, given the high amount of data, manual inspection was only feasible on the basis of spot samples. We therefore invite other researchers to independently reproduce the dependency data we published, employing tools different from ours.

5.2 Implications and Future Work

This section discusses the implications of the results presented in Section 4. It also gives an outlook on possible continuations of the work. We first concentrate on the theory and then look at practical perspectives.

5.2.1 Toward a Theory of Dependency and Change

Many of the results presented in this paper evidence that the general perception of dependencies as vectors or axes of change propagation (see Sangal et al. [42]) is correct. Furthermore, we were able to provide empirical evidence for the transitive propagation of changes via a path of dependencies. With this finding, we back assumptions made by MacCormack et al. [33], for instance.

Apart from these points, our findings also challenge common assumptions on the relationship between dependency and cochange. As pointed out in the introduction, several authors (e.g., [33], [42], [45]) implicitly assume an equal distribution of cochanges among dependencies. In light of our results, this seems to be problematic. If half of the dependencies are not involved in cochanges (see Section 4.1), the dependency structure of a piece of software is a very crude measure for its change behavior. This point is further strengthened by the inequality between the three types of dependencies we differentiated as well as by the results presented in Section 4.3: If the most active 10 percent of the dependencies are responsible for over 70 percent of the cochanges, as is the case in Eclipse, then the cochange behavior is hardly a mirror image of the dependency structure. The high values of inequality among the dependencies throughout the entire set of analyzed projects clearly show that the link between the dependency structure and the cochange behavior is more complicated than previously thought.

We draw the following conclusions: In theoretical investigations of software evolution, we cannot treat all dependencies alike. A differentiated view on dependency between classes is needed. Furthermore, as many dependencies are never involved in 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—independently of the programming language used—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 does the design deserve a critical look. This also means that approaches relying solely on the dependency structure to judge the change behavior and flexibility of a software architecture are problematic. We argue that both the dependency view and the cochange view need to complement each other to paint a full picture of the software architecture.

To round off this discussion, let us take a look at three open issues or possible future lines of research, respectively. We consider them to be the next steps in a comprehensive theory of dependency and change.

In this paper, we focused on the connection between dependency and cochange. Dependency is not the only

possible driver of cochange though. To predict change, a variety of factors have been considered in the literature [9], [22], [45], [47], [48]. Some of them might also be important in the context of cochange. As an example, let us look at complexity. Take, for instance, cyclic complexity by McCabe [36]. In this context, Capiluppi et al. [4] already explored the relationship between cumulative change and complexity in an evolving Open Source system. They found correlations between high cumulative change and high complexity. Such an analysis could be connected with the dependency view. A relevant research question, in our opinion, would be whether complexity of a class correlates with its involvement in cochange events. Another one would be: Are dependencies between complex classes more likely to be an axis of change? Similar questions can be asked for any further metric which was successfully used in change prediction.

A second direction of possible future research would be the extension of our analysis to non-object-oriented languages. One might argue that non-object-oriented languages belong to the past; still it cannot be dismissed that a number of large and important pieces of software are mainly written in plain C: the GNU/Linux operating system and Apache, to name only two. Besides C, COBOL is also still widely used in industry, government, and the military, especially in legacy software. The core question is whether function dependencies exhibit similar characteristics as class dependencies.

Finally, besides our definition of dependency, others are possible. Especially, more fine grained or weighted dependencies deserve consideration. In this study, we choose a very simple definition to build a baseline model. Future research could further test dependency definitions against this baseline model and thus determine the stability of the results in the face of varying concepts of dependency.

5.2.2 Toward a Practical Application

As for the analysis of change behavior, an early example of version control system log analysis is the work of Ball et al. [1]. They also underline the value of change logs for future investigations. Later, with the rise of version control systems and Open Source software, analyzing change behavior of software has become feasible on a large scale. Kazman and Carrière [27], for example, show how the software architecture of a project can be partially recovered based on evidence found in the version control system logs, and Graves et al. [21] show that under certain conditions metrics calculated based on the change histories are more useful in predicting fault rates than metrics calculated on the actual source code. Consequently, project histories and version logs became a popular research target. In particular, the change behavior caught the interest of researchers. To better control change, there has been substantial research in predicting cascading changes: Hassan and Holt [22], Clarkson et al. [9], Ying et al. [47], Zimmermann et al. [48], Tsantalis et al. [45]. Furthermore, from the analysis of these change logs the so-called *change coupling* between modules can be calculated, as Gall et al. [12] demonstrate. Recent work also led to practical tools which can be integrated into the development process: see, for example, the work of Gall et al. [13].

The main difference between previous work and the research presented in this paper is that it focuses on analyzing the possible connection between dependency and cochange while previous work aimed mainly at practical applications such as change prediction and online advice for the programmer. In this paper, we investigated the general relationship between dependency and cochange. We think that this approach generates strong synergy with more applied lines of research as our findings may inform future tools and change prediction approaches.

We will now sketch such a practical application and show how a *combination* of the dependency view (D) and the change coupling view (C) could be used to target refactoring effort; not in theory, but in practice. The basic idea is to filter the dependency matrix D and weight the entries with their cochange activity. This can be accomplished by multiplying D and C element-wise:

$$D'_{i,j} = D_{i,j} * C_{i,j}. \tag{13}$$

Fig. 4 compares the resulting D' with D visually for one specific project: AspectJ. In the right graph D', all change neutral dependencies are removed and the remaining edges are weighted by the amount of changes propagated according to expression (13). This graphical representation enables us to identify hot spots of change propagation in the architecture: Especially, cluster **E** shows high change propagation. In clusters **A** and **B**, change propagation is more centralized and forms star-like structures. This implies that the central node of the star should be the target of refactoring. Cluster **D**, on the other hand, disintegrates into a few strong change couplings. Finally, clusters **C** and **F** are densely connected but hardly cause change propagation, which means that their dependencies are change neutral. Consequently, refactoring effort should primarily concentrate on the classes in clusters **E** and **B**.

Besides the graphical representation, the raw cochange matrix may also provide guidance. The link with the highest number of cochanges can easily be identified. As an example, let us consider Azureus. The most change-active dependency, with 44 cochanges, is the one between the class `DownloadManagerImpl` (package `org.gudy.azureus2.core3.download.impl`) `DownloadManagerController` (same package). `DownloadManagerImpl` is also the class which involved in the highest number of cochange events (203). It can be considered *the* cochange hot spot in Azureus. The question is now: Did we find a design problem? Indeed, manual inspection of these classes reveals a number of design flaws we already mentioned in Section 1. First of all, both classes take too many responsibilities. They count 144 (the third highest value in the software) and 82 public methods, respectively. The methods are only loosely related to each other; their cohesion is low. This clearly violates the Single Responsibility and the High Cohesion rule ([35, ch. 10]). Furthermore, responsibilities are not properly divided between `DownloadManagerImpl` and `DownloadManagerController`. Coupling between the two classes is high. Even though `DownloadManagerImpl` provides an interface (`DownloadManager`), it is not consistently used by `DownloadManagerController`. Apart from that, the class fan-out complexity of `DownloadManagerImpl` is 64, which means that it uses immediately 64 subordinate classes (see [26] for the original definition). Code style tools such as Checkstyle (http://checkstyle.sourceforge.net) already consider a Fan-Out of 20 a rule violation. Finally, it is important

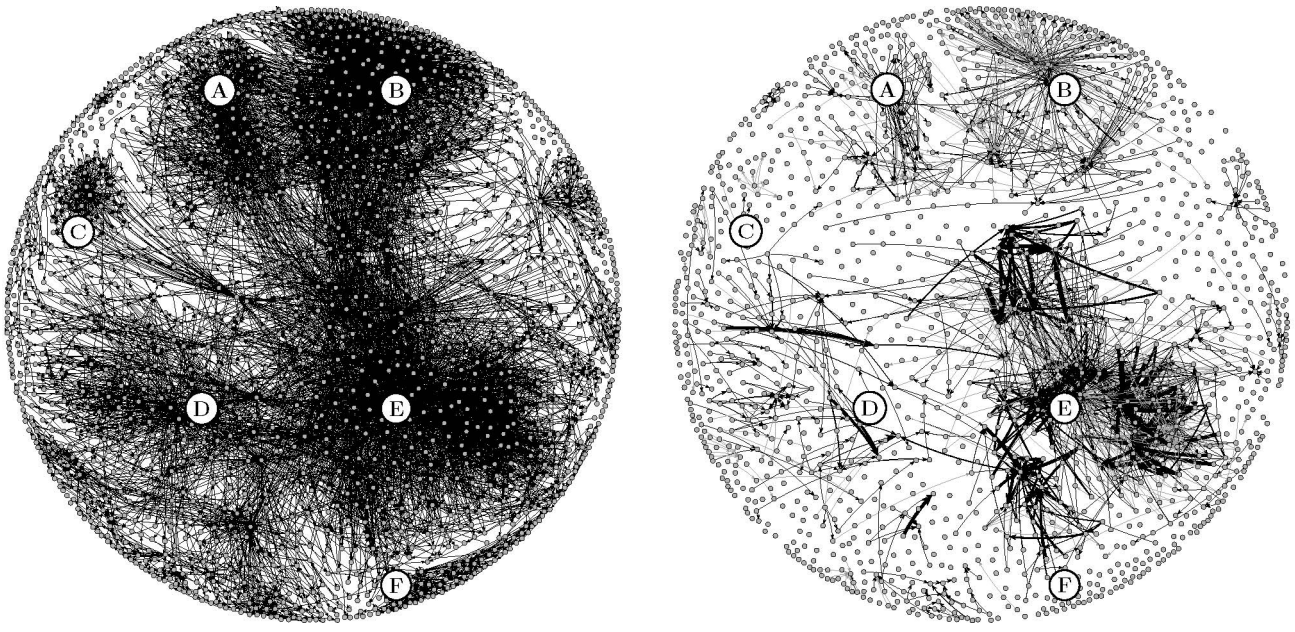

Fig. 4. Left: The code dependency network D of AspectJ. Right: Change coupling C superimposed on D (see (13)). While some clusters, i.e., **E**, constitute hot spots of change propagation, it can also be seen that some high interdependence in D is not necessarily associated with high cochange activity: Clusters **C** and **F** are densely connected (left) but do not show change propagation (right). For more details on the specific graph layout algorithm used, see [16].

to note that the just described problems are not present in the other classes of the same package or superpackage.

From these observations, we can conclude we have not only identified a cochange hot spot but also an example of particularly problematic software design. However, until the just-described method can be used as a productive tool several aspects need to be clarified in future research: We think that more case studies are needed to find out whether our example is representative.

6 Conclusions

For the dependency analysis, we built on work by Sangal et al. [42] and MacCormack et al. [33] which we adapted to Object Oriented software, namely, Java. For the cochange calculation, we followed Ball et al. [1]. The contribution of this paper is having brought both together to empirically analyze the link between dependency and cochange. In particular, we asked two questions: First, "Do changes propagate from one module to another along dependencies between these modules?" We provided positive evidence for this propagation of change by calculating conditional probabilities and fitting a propagation model to the collected data. Second, we asked "Are all dependencies more or less comparable in their propensity to propagate change?" We used Lorenz Curves and Gini Coefficients to quantify the skewedness of the distribution of cochanges on dependencies. The results clearly showed that cochanges distribute in a highly unequal manner.

The picture of dependencies resulting from these findings is two sided: On the one side, the classical picture of dependencies as propagators of cochange is supported. On the other side, the finding that the lion's share of actually propagated changes concentrates on a small subset of the dependencies is new and counterintuitive. As a consequence, we argue that inferring the cochange characteristics of a software architecture solely from its dependency network results in a severely distorted approximation of cochange reality. Furthermore, dependencies should not necessarily be seen as a nuisance because, as we argue in Section 5.2.1, change neutral dependencies enable code reuse, an important ingredient of a flexible architecture. Finally, to target refactoring effort, the dependency structure alone is not enough. It needs to be complemented with an analysis of the change logs of the project. In Section 5.2.2, we provided a brief practical example in which a combined analysis of dependencies and change coupling lead us to design flaws in the Azureus software.

In that sense, we hope that the results and approaches presented in this paper foster a better understanding of the nature of software dependencies, spawn discussion, and support the development of new software engineering tools.

Acknowledgments

The authors acknowledge financial support by the Swiss National Science Foundation (SNSF) (Grant CR12I1_125298).

References

[1] T. Ball, J. Kim, A. Porter, and H. Siy, "If Your Version Control System Could Talk," *Proc. ICSE Workshop Process Modelling and Empirical Studies of Software Eng.*, 1997.

[2] K.H. Bennett and V.T. Rajlich, "Software Maintenance and Evolution: A Roadmap," *Proc. Conf. Future of Software Eng.*, pp. 73-87, 2000.

[3] S. Bohner and R. Arnold, *Software Change Impact Analysis.* IEEE CS Press, 1996.

[4] A. Capiluppi, A. Faria, and J. Ramil, "Exploring the Relationship between Cumulative Change and Complexity in an Open Source System," *Proc. Ninth European Conf. Software Maintenance and Reeng.*, pp. 21-29, 2005.

[5] R. Carver, "The Case against Statistical Significance Testing, Revisited," *J. Experimental Education,* vol. 61, no. 4, pp. 287-292, 1993.

[6] R.P. Carver, "The Case against Statistical Significance Testing," *Harvard Educational Rev.*, vol. 48, no. 3, pp. 378-399, 1978.

[7] D. Challet and A. Lombardoni, "Bug Propagation and Debugging in Asymmetric Software Structures," *Physical Rev. E*, vol. 70, no. 4, p. 046109, 2004, doi: 10.1103/PhysRevE.70.046109.

[8] N. Chapin, J. Hale, K. Khan, J. Ramil, and W. Tan, "Types of Software Evolution and Software Maintenance," *J. Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 1, pp. 3-30, 2001.

[9] P. Clarkson, C. Simons, and C. Eckert, "Predicting Change Propagation in Complex Design," *J. Mechanical Design*, vol. 126, pp. 788-797, 2004.

[10] N. Fenton and N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System," *IEEE Trans. Software Eng.*, vol. 26, no. 8, pp. 797-814, Aug. 2000.

[11] R. Floyd, "Algorithm 97: Shortest Path," *Comm. ACM*, vol. 5, no. 6, p. 345, 1962.

[12] H. Gall, M. Jazayeri, and J. Krajewski, "CVS Release History Data for Detecting Logical Couplings," *Proc. Sixth Int'l Workshop Principles of Software Evolution*, pp. 13-23, 2003.

[13] H. Gall, B. Fluri, and M. Pinzger, "Change Analysis with Evolizer and Changedistiller," *IEEE Software*, vol. 26, no. 1, pp. 26-33, Jan./Feb. 2009.

[14] K. Gallagher and J. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Trans. Software Eng.*, vol. 17, no. 8, pp. 751-761, Aug. 1991.

[15] J.L. Gastwirth, "The Estimation of the Lorenz Curve and Gini Index," *The Rev. of Economics and Statistics*, vol. 54, no. 3, pp. 306-316, 1972.

[16] M.M. Geipel, "Self-Organization Applied to Dynamic Network Layout," *Int'l J. Modern Physics C*, vol. 18, no. 10, pp. 1537-1549, Oct. 2007.

[17] M.M. Geipel and F. Schweitzer, "Software Change Dynamics: Evidence from 35 Java Projects," *Proc. Seventh Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. Foundations of Software Eng.*, pp. 269-272, 2009.

[18] D. German, "An Empirical Study of Fine-Grained Software Modifications," *Empirical Software Eng.*, vol. 11, no. 3, pp. 369-393, 2006.

[19] C. Gini, "Measurement of Inequality of Incomes," *The Economic J.*, vol. 31, pp. 124-126, 1921.

[20] M. Goodrich, R. Tamassia, and J. Wiley, *Algorithm Design: Foundations, Analysis, and Internet Examples.* Wiley, 2002.

[21] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Trans. Software Eng.*, vol. 26, no. 7, pp. 653-661, July 2000.

[22] A. Hassan and R. Holt, "Predicting Change Propagation in Software Systems," *Proc. 20th IEEE Int'l Conf. Software Maintenance*, pp. 284-293, 2004.

[23] J. Hennessy, D. Patterson, D. Goldberg, and K. Asanovic, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2003.

[24] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Trans. Programming Languages and Systems*, vol. 12, no. 1, pp. 26-60, 1990.

[25] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley Professional, Oct. 1999.

[26] D. Kafura and G. Reddy, "The Use of Software Complexity Metrics in Software Maintenance," *IEEE Trans. Software Eng.*, vol. 13, no. 3, pp. 335-343, Mar. 1987.

[27] R. Kazman and S. Carrière, "Playing Detective: Reconstructing Software Architecture from Available Evidence," *Automated Software Eng.*, vol. 6, no. 2, pp. 107-138, 1999.

[28] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *Proc. 11th European Conf. Object-Oriented Programming*, M. Akşit and S. Matsuoka, eds., pp. 220-242, 1997.

[29] M. LaMantia, Y. Cai, A. MacCormack, and J. Rusnak, "Analyzing the Evolution of Large-Scale Software Systems Using Design Structure Matrices and Design Rule Theory: Two Exploratory Cases," *Proc. Seventh Working IEEE/IFIP Conf. Software Architecture*, pp. 83-92, 2008.

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

[31] K.J. Lieberherr and I.M. Holland, "Assuring Good Style for Object-Oriented Programs," *IEEE Software*, vol. 6, no. 5, pp. 38-48, Sept. 1989.

[32] M. Lorenz, "Methods of Measuring the Concentration of Wealth," *Am. Statistical Assoc.*, vol. 9, no. 70, pp. 209-219, 1905.

[33] 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*, vol. 52, no. 7, pp. 1015-1030, 2006.

[34] R. Martin, "Granularity," *C++ Report*, vol. 8, no. 10, pp. 57-62, 1996.

[35] R. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall, 2008.

[36] T.J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 308-320, Dec. 1976.

[37] M. Natrella, C. Croarkin, P. Tobias, J.J. Filliben, B. Hembree, W. Guthrie, L. Trutna, and J. Prins, "NIST/SEMATECH E-Handbook of Statistical Methods," *NIST/SEMATECH*, http://www.itl.nist.gov/div898/handbook/, 2011.

[38] W.F. Opdyke and R.E. Johnson, "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems," *Proc. Symp. Object-Oriented Programming Emphasizing Practical Applications*, 1990.

[39] D. Parnas, "Software Aging," *Proc. 16th Int'l Conf. Software Eng.*, pp. 279-287, 1994.

[40] K. Popper, *Conjectures and Refutations: The Growth of Scientific Knowledge.* Routledge, 2002.

[41] V. Rajlich, "A Model for Change Propagation Based on Graph Rewriting," *Proc. Int'l Conf. Software Maintenance*, pp. 84-91, 1997.

[42] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using Dependency Models to Manage Complex Software Architecture," *Proc. 20th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pp. 167-176, 2005.

[43] D.V. Steward, "The Design Structure System—A Method for Managing the Design of Complex Systems," *IEEE Trans. Eng. Management*, vol. 28, no. 3, pp. 71-74, Aug. 1981.

[44] K.J. Sullivan, W.G. Griswold, Y. Cai, and B. Hallen, "The Structure and Value of Modularity in Software Design," *SIGSOFT Software Eng. Notes*, vol. 26, no. 5, pp. 99-108, 2001.

[45] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, "Predicting the Probability of Change in Object-Oriented Systems," *IEEE Trans. Software Eng.*, vol. 31, no. 7, pp. 601-614, July 2005.

[46] S. Warshall, "A Theorem on Boolean Matrices," *J. ACM*, vol. 9, no. 1, pp. 11-12, 1962.

[47] A.T. Ying, R. Ng, and M. Chu-Carroll, "Predicting Source Code Changes by Mining Change History," *IEEE Trans. Software Eng.*, vol. 30, no. 9, pp. 574-586, Sept. 2004.

[48] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," *IEEE Trans. Software Eng.*, vol. 31, no. 6, pp. 429-445, June 2005.

Markus Michael Geipel studied computer science at the Technische Universität München (University of Technology, Munich) and the University of Texas at Austin. He graduated from the Technische Universität München with highest distinction and later received the doctorate degree from the ETH Zurich. He currently works in R&D at the German National Library, were he focuses on the topics software quality, linked data/Semantic Web, and architectures for large-scale data processing.

Frank Schweitzer has been a professor and chair of systems design at ETH Zurich since 2004. He is also an associated member of the Department of Physics at the ETH Zurich. His research interests focus on applications of complex systems theory to the dynamics of social and economic organizations. He is a principal investigator of several interdisciplinary research projects. The Swiss National Science Foundation has funded his proposal "On the Interplay between Social Interactions and Software Architecture in Open Source Software" for 24 months. A list of recent publications, talks, and other scientific activities can be found on his website: http://www.sg.ethz.ch/.