Advances in Complex Systems
Vol. 15, No. 6 (2012) 1250083 (22 pages)
© World Scientific Publishing Company
DOI: 10.1142/S021952591250083X

MODULARITY, DEPENDENCE AND CHANGE

MARKUS MICHAEL GEIPEL
Chair of Systems Design, D-MTEC,
ETH Zurich – CH-8032 Zurich, Switzerland
markus.geipel@alumni.ethz.ch

Received 9 April 2012
Revised 30 June 2012
Accepted 17 July 2012
Published 10 August 2012

Technological artifacts such as software often comprise a large number of modules; more than twenty thousand in the case of the Java software Eclipse. While on the micro-level this system is modular, how should the building blocks be arranged on the macro-level? In the literature this question has mainly been addressed with the same arguments already used to advocate modularity on the micro-level: Dependencies should be minimized as they impede optimization and flexibility of the system. In contrast to this I argue that along with a change from the micro view to the macro view also the argumentation has to change. In this paper, I analyze the theoretical ramifications of dependency between modules on the macro-level. In particular, I argue that macro-level dependencies are first weak dependencies, and second, foster flexibility and change efficiency. This argumentation is supported by an empirical analysis of 35 software architectures. Data show that dependency relations seldom cause change propagation. Furthermore, high dependency in the architecture negatively correlates with the occurrence of large change events. Thus, higher interdependency is associated with higher evolvability and more efficient change.

Keywords: Modularity; networks; evolution.

1. Introduction

In 1970, Cristopher Alexander stated: "*Today more and more design problems are reaching soluble levels of complexity*" [3, p. 3]. Now, several decades later, the trend to increasing complexity even intensifies. Not only do organizations and products become more and more complex but they also have to prove themselves in environments that are changing more and more rapidly. The idea of building a system from scratch, to last forever, has become a concept of the past. Thus, a modern system — be it an organization or a product — should not only be functional and efficient, but also flexible.

This challenge, scholars across disciplines addressed almost unanimously with a call for modularity: [5, 34, 35, 48, 55, 62, 64, 73] Modularity, as Langlois [34] puts it, is achieved "[*b*]*y breaking up a complex system into discrete pieces which can* *then communicate with one another only through standardized interfaces within a standardized architecture*". "The power of modularity" as Baldwin and Clark [6] put it, has proven to be a veritable boon. Not only theoretically, but also in real world scenarios. Langlois and Robertson [35] for example showed that modularizing computer architecture and stereo equipment was rewarded with an explosion of product variety and fast paced innovation.

The story does not end here, though. It is only the beginning. Let us now go one step further: Having organized a system by decomposing it into modules of manageable size we face another question, which, unlike modularity, has as yet not been sufficiently discussed. Often, the modular system is comprised of a large number of parts. The Java software Eclipse for example consists of more than 20,000 different modules (Java-classes). The question is: "Having a modular structure on the micro-level — or statement level in software — what structure should the macro-level exhibit?" Or, in other words, how should the modules or classes be combined to form the macro-level of the system? Empirical evidence by Geipel and Schweitzer [20] for instance shows that the simple dictum "minimize dependency" is a problematic one. A more complex view of dependency seems necessary. Especially as the structure of the macro-level has a strong impact on the evolvability of the system as well as modification costs, and should therefore be a central question of system design.

This paper addresses the challenge, both theoretically as well as empirically. The argument is that the mechanisms on the macro-level, which focuses on the dependency between modules, differ largely from the ones on the micro-level which comprises fine grained dependency structures between functions, variables and statements: Design principles valid on the micro-level are not necessarily effective on the macro-level. They make clear the difference between micro-level and macro-level, as well as the arguments connected to each view. Section 2 reviews the traditional micro-level perspective and the classical arguments against dependency. Section 3 introduces the macro-level perspective and explain how the role of dependencies differs for this view. The counterintuitive conclusion is, that lopsided minimization of dependence on the macro-level increases change costs and decreases evolvability. To back this argumentation, an empirical analysis is presented in Sec. 4. Section 5 presents the results of this analysis. They empirically back the theoretical arguments of the previous sections, based on evidence from 35 Java projects. Finally, the discussion is wrapped up in Sec. 6.

2. The Ubiquity of Dependency

A Microsoft official, working on the Windows operating system, was quoted by Guth [24] as follows: "*With each patch and enhancement, it became harder to strap new features onto the software since new code could affect everything else in unpredictable ways.*" The key concept here, is *dependency*. Different parts of the system are dependent, and *affect* each other.

Indeed, in the history of modeling complex systems, considering elements connected by a dependency relation has proven to be a dominant and timeless concept. Already Simon [58] modeled complex systems by means of matrices which reflected the dependency between parts of the system. He pointed out that real world systems tend to be organized as a hierarchical composition of approximately independent subsystems (Near Decomposability).

The system model used by Alexander [3, Appendix II] is designed along the same lines: Elements of the system are connected by either positive or negative influence links. He argues that designers should define subsystems that can be adjusted more or less independently. Von Hippel [66] later applied this interdependency view to innovation tasks.

More focused on practical application, Steward [61] developed the Design Structure Matrix methodology which evolved subsequently into a whole toolbox for planning engineering tasks [14]. Furthermore, the dependency view entered into practical management: The value chain model of Porter [50, Chap. 2] uses task dependency as a key concept. Also in software engineering, similar dependency-based approaches [27, 62] are common.

Again, with a theoretical focus, Kauffman [31] presented the NK-model which establishes a connection between the dependency structure of a system and the fitness landscape it generates. It thus establishes a connection between dependency in a system and its evolvability.

What are the ramifications of dependency? In the same vein as the quotation introducing this section, dependency is often viewed as a nuisance. A nuisance in two respects: First, in optimizing the system configuration, and second, in performing changes to the system. The next two subsections delve into these two aspects.

2.1. *Optimizing the system*

Given a system which is comprised of elements and dependencies between the elements as described in the previous section, the next step is to establish a connection between this dependency structure and the evolvability of the system. Which structure facilitates optimization and which structure hinders optimization?

With this NK-model, Kauffman [31] added this evolutionary perspective to the dependency discussion. While the NK-model was originally used to target biological questions, it spawned a plethora of follow-up literature in management and organization science (see Refs. [15, 16, 52, 55]).

In a nutshell, the NK-model defines a system as a set of n binary variables ($X \in \mathbb{B}^n$) with directed interdependencies between them. The parameter k denotes the average number of dependencies per system variable. A system configuration can be written as a binary string of length n, reflecting the states of the n system variables. The fitness of the system is calculated via the fitness landscape. A fitness landscape in general [70] is a function $F : X \to y$ that takes as input the (multi-dimensional) system configuration X and maps it to a scalar fitness value y.

In the case of the NK-model it is assumed that each binary variable x in the system has a fitness contribution f_x which depends on the state of x and the states of all variables connected to x via dependencies. F is realized by summing up these individual contributions:

$$F(X) = \sum_{x \in X} f_x(X). \tag{1}$$

Consequently, dependency generates "rugged" landscapes: The more dependencies the lower the correlation between neighboring points in the genotype space (for a mathematical treatment, see Ref. [68]). Thus, interdependence renders local search increasingly inefficient. Kauffman [31, p. 52] terms this the *Complexity Catastrophe*: "*As complexity increases, the height of accessible peaks fall toward the mean fitness.*" Figure 1 schematically shows the fitness landscapes corresponding to the two extremes of complete independence and complete interdependence.

Assuming that the system is optimized by evolutionary forces — *local search* — minimization of dependencies is obviously desirable. An evolutionary point of view is indeed deeply rooted in economic thought; starting from Schumpeter's "Entwicklung" [57], strong arguments for an evolutionary optimization process, based on local search, have repeatedly been advocated [1, 45].

Even if an evolutionary process is rejected and opted for optimizing rational agents, the *Complexity Catastrophe* keeps lurking: With increasing dependencies and system size, finding an optimal system configuration becomes unfeasible. Weinberger [68] showed that depending on the degree of dependency k the search for the optimal system configuration falls into different computational complexity classes: The two extremes being *solvable in linear time*: $O(N)$ for complete independence and *solvable only in exponential time*: $O(2^N N)$ for complete interdependence. For a boundedly rational agent [30, 41, 59], a k too high will thus thwart optimization effort. In the context of the NK-model reasoning along these lines is reflected in a number of works [4, 16, 32, 36, 37].

Likewise, argumentation is not restricted to an NK-model context. In software engineering Loyall and Mathisen [39], Bohner and Arnold [7], and Ryder and Tip [54] discuss the impact of changes. Furthermore, in the Design Structure

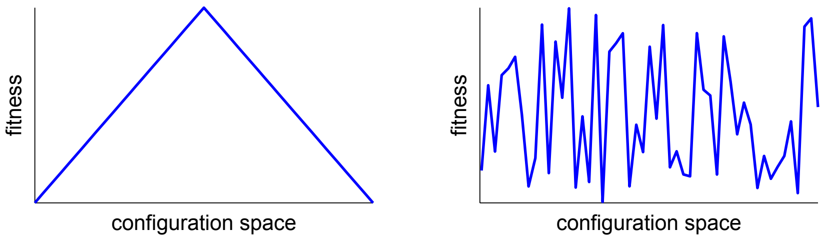

Fig. 1. (Color online) Schematic fitness landscapes: A system with independent parts generates a smooth fitness landscape (left). High interdependence between parts, however, results in a rugged landscape (right).

Matrix community, high dependency is linked to a more complex and time consuming design process [14].

Finally, Herbert Simon saw a significant connection between the evolvability of a system and its structure which he exemplified with his parable of the two Swiss watchmakers [60].[a] Based on this literature it can be concluded that system optimization is hindered by dependencies; at least on the micro-level.

Putting software in the NK model framework, the source code corresponds to the configuration space. Code defines functionality. If the resulting functionality is the one intended, it is assigned — using NK terminology — a high fitness. If dependency is low, small changes in the code translate to small changes in functionality and thus small changes of fitness (the situation depicted in on the left of Fig. 1). Conversely, with high dependency a small change may influence large parts of the software in maybe unpredictable ways and thus result in large jumps in fitness (right part of Fig. 1).

2.2. *Implementing change*

In the previous section, the problems interdependence inflicts on the optimization of a system were discussed. In this section, arguments are presented which lead to the conclusion that also the implementation of a configuration found in the optimization process is hindered by dependencies.

The question is, "What do we expect from a flexible architecture?". First of all, the architecture should be extendable (see also Ref. [49]). Adding a feature should leave the rest of the architecture more or less untouched. Next, the modifications needed to advance from one version of the software to the next should be as independent as possible. This means that they can be distributed on different people and on different points in time. Small independent changes also facilitate quality control as problems can be backtracked to the responsible modification more easily. Henceforth, a change event is modeled as a set of atomic changes C_t that were committed simultaneously at time t.

How does interdependence interfere with the objective of having small independent change events? Imagine that **b** depends on **a**, and **a** is modified. **b** will be influenced to some degree and there is a risk that this influence might be strong enough to force a modification of **b** to preserve the integrity of the system. In this case, the modification of **a** entails a subsequent modification of **b**. The change performed in **a** propagates to **b**. The result is a change event of size two: $C_t = \{a, b\}$. If other modules depend on **b** this change could even propagate further and cause a whole avalanche. The result: large change events.

[a]In a nutshell: One watchmaker is building watches in a monolithic fashion, the other one in a modular fashion. Damage in the monolithic watch renders the whole watch unusable. In a modular one, damage only renders one module useless. Given a certain rate of damage, a monolithic design becomes unfeasible.

Argumentation along these lines is common in literature, especially in the context of software: The law of Demeter, suggested by Lieberherr and Holland [38], aims at organizing and reducing dependencies between classes. In the same vein, MacCormack *et al.* [40] define a change cost metric by inferring the degree to which a change in any single element causes a (potential) change in other elements of the system. It is assumed that changes (potentially) propagate along links and consequently high dependency results in high change costs or less options for change. As will be shown in Sec. 3.2, as well as in the empirical results in Sec. 5, such an argumentation in the context software (an already modularized system) can be problematic. Propagation of changes along code dependencies has also been reported in several works [11,26,51]. Finally, Gorshenev and Pis'mak [22] argue along the same lines when explaining the size distribution of change events in software evolution.

3. The Macro-Level: In Favor of Dependency

"*A modular system is composed of units* (*or modules*) *that are designed independently but still function as an integrated whole.*" [5, box on page 86]. This section deals with this "functioning as an integrated whole". Arguments will be presented why the switch from micro- to macro-level changes the rules of the game. In particular, a system in which modules make heavy use of each other's functionality will allow for more efficient change. In other words, while decoupling and dependency minimization is desirable on the micro-level, as argued throughout Sec. 2, on the macro-level, high interdependence in the sense of module usage is desirable. There are two arguments backing this proposition. The first one (Sec. 3.1) is based on an evolutionary point of view and the second one (Sec. 3.2) takes a software engineering stance, highlighting the boon of module reuse.

3.1. *Large evolutionary steps*

Altering the fitness of a system requires changes in the system configuration, changes in the system parts. Previous research concentrated on the question: "How do we find the best configuration?". Let us ask now: "How do we implement a configuration if change is costly?". The focus thus shifts from finding solutions to implementing solutions.

In answering the search question it was assumed that optimization can be proxied as local adaption. While it can be considered as a fact that man's optimization capabilities are limited and his behavior is, at best boundedly rational, limiting system optimization to a blindfolded hill climbing mechanism is most probably an oversimplification. Let us assume more farsighted but still limited optimization capabilities. Let us further assume a cost of implementing solutions. A cost that might outweigh the cost of searching a solution. Looking at software, for example, this cost is important indeed: Besides consuming time and money, large changes in software bear several risks: First, with each changed line of code a bug could be introduced. Second, the more changes are made, the harder it will be to localize a newly introduced bug. Besides these technical costs and risks, large changes in a system are often hindered by sociological forces: People do not like change [13, Chap. 30].

Thus, a system should be designed in a way that its function/fitness can be changed significantly with only minimal modifications of the system's implementation. The efficiency of change can be defined as change in the functionality divided by necessary change in the implementation. Obviously, lower implementation costs and higher change efficiency are possible on rugged fitness landscapes: In the left landscape shown in Fig. 1, starting from a random system configuration, the risk is high that we need to "walk" a long distance in the configuration space to reach an optimum. This walk is costly. In the right landscape, however, optima are widely distributed. Any randomly chosen point in implementation space is close to an optimum. The walk will be short.

This argument for change efficiency in rugged landscapes is further strengthened if we assume that we are not searching *the* optimum but rather *one* sufficiently good local optimum. In other words, a satisficing strategy is adopted, as described by March and Simon [42] and Cyert and March [12]. A landscape offering many such "islands" of satisfactory fitness distributed throughout configuration space is superior to one that offers only one such island, as the next island can on average be reached with less effort.

This leads to the following conclusion: Given substantial implementation costs, compared to search costs, a rugged landscape gains attractiveness, as the fitness of such a system can be changed with less implementation effort. If these conditions are met in the case in software engineering reality, the following hypothesis should hold true:

Hypothesis 1. Good design and high flexibility are compatible with high levels of interdependency in the code.

3.2. *Module reuse*

It is only logical that if **b** depends on **a** and **a** is modified, **b** is affected. The question is: "Are we really forced to modify **b** subsequently?" If the micro-level of the system is already modular, meaning that **a** and **b** are realized as modules using information hiding and interfaces, it should be possible to modify them independently. After all, this is a central part of the very definition of modularity. Change would still affect dependent modules, but not necessitate changing them. The dependency structure we are dealing with thus resembles more a loosely coupled system in the sense of Orton and Weick [47] than a system with strong dependencies in the sense of the NK-model [31], Alexander's dependency model [3, Appendix II] or the innovation task partition model by von Hippel [66].

The following thought experiment shows that introducing (weak) dependencies can reduce change effort: Imagine a system with three units performing three different tasks **x**, **y**, **z**. To accomplish them they need to perform a sub-task **a**.

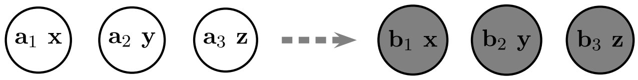

Fig. 2. A system with three modules, each using a helper function **a**. The design minimizes dependencies, however, in case **a** needs to be changed to **b** an avalanche of changes is caused (gray).

Figure 2 shows a configuration where each unit incorporates its own version of **a** ($\mathbf{a}_1$, $\mathbf{a}_2$, $\mathbf{a}_3$). Having dependency minimization in mind, this seems to be well done: There are no dependencies and no changes can propagate one might think. Also metrics looking only at the nodes and dependencies would give a favorable judgment.

Yet, a software engineer would instantly dismiss such a design. It contradicts one of the most basic principles of software design[b]: "Don't repeat yourself!" as formulated by Hunt and Thomas [29, Chap. 2]. Also Fowler denounce repetition as bad design. Software engineers would suggest the design shown in Fig. 3.

Sub-task **a** is separated from **x**, **y** and **z** and not duplicated. Each **x**, **y** and **z** uses the same **a** and as a consequence is dependent on it. Even though this process generates dependencies on the macro-level, it reflects perfectly the definition of modularity on the micro-level. The dependencies are governed by an interface contract and implementation details are hidden, just as Parnas [48] and Baldwin and Clark [5] recommend.

Furthermore, exactly because of these dependency links, it is more flexible. Suppose a better way — let us call it **b** — is found to perform the task formally performed by **a**. In the design in Fig. 2, we need to change three units (change is indicated by gray shading) to incorporate the better solution **b**. In Fig. 3, in contrast, only one change is needed. The performance of **x**, **y** and **z** is influenced via the dependency links. Imagine now that **x**, **y** and **z** need more helper functions. We will end up with more and more dependencies and at the same with more and more flexibility.

The conclusions to be drawn from this thought experiment are the following: First, in certain cases dependencies prevent change avalanches instead of

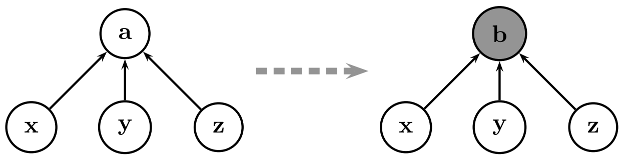

Fig. 3. A system with three modules, each using a helper function **a** which has been separated. The design has more dependencies than the previous one (Fig. 2), but limits modifications caused if **a** needs to be changed to **b** (gray).

[b]It should not be left unmentioned that also management scholars argue for reuse in this context: [25].

creating ones. This flexibility of interdependent systems should be reflected in the change history:

Hypothesis 2. Higher interdependence is associated with more small changes and less big changes.

Second, if the elements of the systems are already modules adhering to the laws of information hiding, it is unlikely that dependencies between them propagate changes.

Hypothesis 3. Change propagation along class dependencies is the exception, not the rule.

Taking into account the widespread use of modularity across disciplines, we can assume that this reasoning does not only pertain to software but likewise to a variety of social and technological systems.

4. Research Design

In the previous section, it was argued that high reuse between modules enables efficient changes and fosters flexibility. In this section, empirical evidence for this proposition is presented.

Software architecture is especially suitable for this endeavor: First, its structure can be retrieved in an automated and unbiased fashion. Although system architectures in other fields have been successfully analyzed the process of formalizing them, thus finding the building blocks and the dependencies between them, it is tedious and prone to subjective judgment. Furthermore, getting reliable data on the modification history is hardly feasible. Software elegantly avoids these problems: The dependency structure can be read from the source files and the modification history is recorded by a version control system such as SVN or CVS.

Furthermore, the burgeoning of Open Source Software has attracted many scholars and inspired studies in various fields [67]. This is a convenient situation as it allows us to build on a solid basis of established knowledge and best practice methods.

While analysis of software from the dependency point of view is an established procedure, the two level view argued for in the previous sections, is a novel aspect. The next section is thus dedicated to explain the mapping of the micro- and macro-system level to software (Sec. 4.1). Next, the data set used in the empirical analysis is described, including the data retrieved (Sec. 4.2). Finally, in Sec. 4.3, the measurements conducted are formally defined.

4.1. *Mapping the micro- and macro-level to software*

In the software industry it was soon clear that large projects needed to be structured and that programming languages needed to support a decomposition of the code into modules. Thus, many languages formally supported a certain degree of modularity. Among them Ada, D, F, Fortran, Pascal (partly), ML, and Modula-2. The decisive step toward modularity in the modern sense — as for example defined by Ulrich and Tung [64] or Baldwin and Clark [5] — was the introduction of "Information Hiding" by Parnas [48]. The code should not only be structured, but implementation details should be hidden from the user of the module. Modules are black boxes hiding their inner working and are exposing only the interface, necessary for calling their services. Developers should be relieved from knowing everything about the system. Such an architecture has two advantages: it eliminates the inevitable information overload a programmer has to bear in a growing project, and second it contains the effects of modification, as the hidden code can be changed without consequences, as long as the interface contract is not violated. Object-oriented programming languages such as C++, Java, Eiffel etc. incarnate these principles. They require each piece of code to reside in a class — a blueprint of a cohesive package of data and function — which hides its inner working behind a public interface. Classes form the basic modular building blocks of any object-oriented program. Figure 4 illustrates this principle.

The classical object-oriented software engineering methodology [8, 44, 53] not only describes the principles of classes and interfaces but also defines how real world problems are mapped to an object-oriented representation. In other words, how the problem should be decomposed into a system of interacting classes. Building on this basis, pattern oriented software engineering — inspired by Alexander *et al.* [2] — focuses on the composition of small ensembles of classes and suggests abstract solution patterns for recurring problems [18].

Besides these prescriptive approaches to software architecture, there also exist more empirical approaches where the focus lies on measuring aggregate properties of the architecture. A method especially suitable for object-oriented software is the Design Structure Matrix (DSM). It is a frequently used tool to depict and analyze the module dependencies in product architecture and was devised by Steward [61]. Applications of the Design Structure Matrix method in the field of software are discussed by Sullivan *et al.* [62], Sangal *et al.* [56] and MacCormack *et al.* [40]. The idea is that the modules and their dependencies form a directed network which can be interpreted as an adjacency matrix. Therefore this abstraction of the software resembles the dependency-based models discussed in the previous sections.

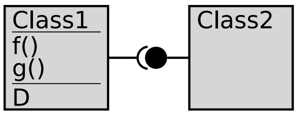

Fig. 4. A schematic view of classes in object-oriented programming. `Class1` contains functions (`f()` and `g()`) as well as data (`D`). From the outside visible, however, is only its interface. Other modules access `Class1`'s services only via this interface, without knowing the implementation details hidden inside `Class1`.

```
1   public class SingleUserRatingInfo extends RatingInfoList {
2     private final TOTorrent torrent;
3
4     public SingleUserRatingInfo(TOTorrent torrent) {
5       this.torrent = torrent;
6     }
7      [...]
8   }
```

Fig. 5. The source code of the class `SingleUserRatingInfo` from the Azureus project. ([...] omitted parts).

The concept of class dependency shall now be explained based on the class `SingleUserRatingInfo` which forms part of the Azureus project. Figure 5 shows a shortened version of the class. `SingleUserRatingInfo` extends the functionality of `RatingInfoList` (line 1) and thus is dependent on it. Furthermore, it uses the class `TOTorrent` (lines 3, 5 and 6); another dependency. On the other hand, `SingleUserRatingInfo` is used by the class `PlatformRatingMessenger` (source not listed here), meaning that `PlatformRatingMessenger` is dependent on `SingleUserRatingInfo`. In a nutshell, two classes are dependent if one class calls a function of the other one, inherits functionality of the other one or comprises the other one as a member. This practice is in line with previous works by Challet and Lombardoni [10], Sangal *et al.* [56] and MacCormack *et al.* [40].

4.2. *Description of the data set*

The empirical analysis presented in this chapter is based on 35 Java projects. The Java programming language was developed at Sun Microsystems (http://java.sun.com) and is officially defined by Gosling *et al.* [23]. For several reasons it is a natural choice for a study such as ours. First, unlike other popular languages such as C++, Java was designed from scratch to be an object-oriented language. There are no interpretation ambiguities. An advantage which adds to the quality and reliability of the results. Furthermore, Java enjoys a high popularity in the Open Source community: On one of the world-wide largest Open Source incubator sites — SourceForge (http://www.sf.net) — Java is used in approximately 25%[c] of the projects, making it the most popular language used. C++ qualified for the second place with 21%.

Accordingly, SourceForge served as main data source in the study. The 34 largest Java projects using CVS as version control system entered our analysis. Size was measured in number of class files and the project EasyEclipse was excluded as it only constitutes a repackaging of the Eclipse IDE. The data is complemented by the Eclipse IDE (http://www.eclipse.org) and AspectJ (http://www.eclipse.org/aspectj/), both hosted by IBM. The complete list of the

[c]Data gathered in August 2007. Note that projects may use more than one language.

35 projects is printed in Appendix A, alongside with the size of each project and the change history's coverage. Further information on each SourceForge project can be found at http://NAME.sf.net, where `NAME` stands for the name of the project.

For each project in the list the change history and the class dependency structure was gathered. The class dependency structure is stored as a directed graph containing the set of classes C and the set of dependencies $D \subseteq C \times C$. $(c_1, c_2) \in D$ reads as "c_1 depends on c_2". The data is retrieved by analyzing the source files of the classes. Any reference to another class is counted as a dependence. The CVS change log is stored as an ordered list L of sets of classes: $L = (C_t)_{t \in T}$, where T stands for time and C_t for the set of classes, that have been changed simultaneously at time t.

4.3. *Definition of measurements*

To test the argumentation presented in Sec. 3, three aggregate measures are needed: The first one is the average dependency of the modules (d). This means the average number of other classes, a class depends on. It is defined by dividing the number of dependencies ($|D|$) by the number of classes ($|C|$):

$$d = \frac{|D|}{|C|}. \tag{2}$$

In graph theoretic terms this is the average node in- or out-degree in the directed dependency graph. Please note that average in- and out-degree are the same in any directed graph as every edge adds one to both the in- and out-degree count. Conclusion: The higher d, the higher the degree of interdependency in the software architecture.

The next aggregate measure needed is the share of multi-class changes among the change events. In Secs. 2.2 and 3.2 it was argued that changes should not propagate. Or, in other words, being able to modify modules one by one in an independent fashion is a desirable property of any system architecture. The degree was measured to which this is possible by calculating how many change events affected only one class (single class change, $|C_t| = 1$) and how many affected many classes at once (multi-class change, $|C_t| > 1$). The share of multi-class changes among all change events provides a measure for the architecture's rigidity. The higher the share, the harder it is to implement change. The formal definition of the share of multi-class changes (m) is

$$m = \frac{|\{C_t \in L : |C_t| > 1\}|}{|L|}. \tag{3}$$

Finally, in Sec. 3.2, it was argued that in modular systems dependencies are loose couplings. Changes of one module will only seldom necessitate a modification of the modules depending on it. To verify this hypothesis, the influence of dependencies on the propagation of changes needs to be measured.

Let us use $\mathcal{C}$ (cursive, not to be confused with C) to denote the fact that two classes have been modified at least once simultaneously, thus $\exists\, C_t \in L : c_i \in C_t, c_j \in C_t$. Let us further use $\mathcal{D}$ (cursive) to denote the fact that two classes are connected by a dependency, thus $(c_i, c_j) \in D$. A straight forward measure for the influence of dependencies on change propagation is the conditional probability $P(\mathcal{C}|\mathcal{D})$. Read: the probability that two classes have at least once been modified together, given that they are connected by a dependency $P(\mathcal{C}|\mathcal{D})$, is calculated from the data as follows:

$$P(\mathcal{C}|\mathcal{D}) = \frac{|\{(c_i, c_j) \in D : \exists\, C_t \in L : c_i \in C_t, c_j \in C_t\}|}{|D|}. \tag{4}$$

As a reference, we also compute the probability $P(\mathcal{C}|\neg\mathcal{D})$ that two unconnected classes have at least once been modified together. We use $\neg\mathcal{D}$ to denote the fact that two classes are not connected by a dependency, thus $(c_i, c_j) \notin D$. $P(\mathcal{C}|\neg\mathcal{D})$ is calculated as follows:

$$P(\mathcal{C}|\neg\mathcal{D}) = \frac{|\{(c_i, c_j) \notin D : \exists\, C_t \in L : c_i \in C_t, c_j \in C_t\}|}{|C|^2 - |D|}. \tag{5}$$

5. Empirical Results

Figure 6 visualizes the dependency network for two different Open Source projects. Surprisingly, a high degree of dependency is evident. Just a first indication that dependency minimization is not the dominant maxim of design. In this section, the measures defined in the previous section are applied with the end to quantitatively describe these networks.

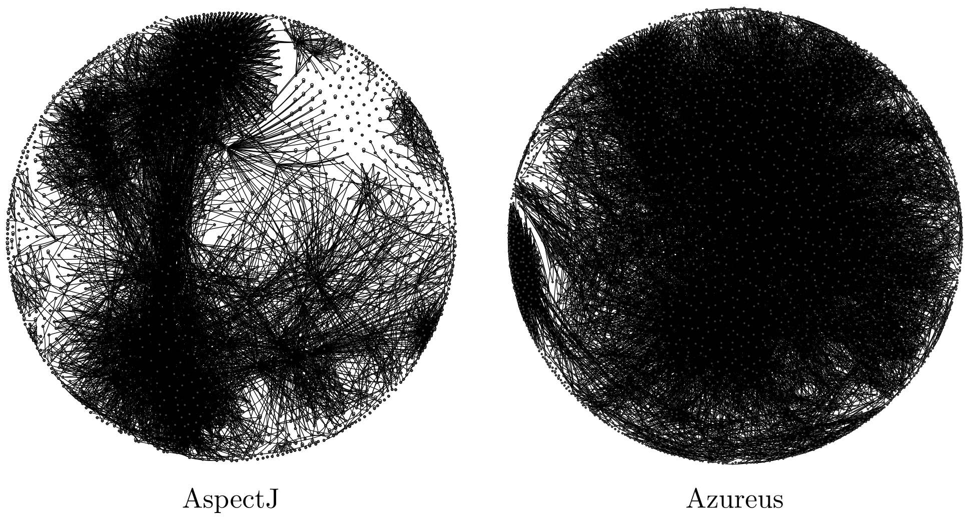

Fig. 6. Dependency on the macro-level in two OSS projects. Nodes stand for Java classes and edges for dependencies. Both pictures represent the state of the projects on 1st January, 2008.

The results are presented in the following way: first one prominent project — the Eclipse development environment — is discussed. This discussion addresses Hypothesis 1. Next, we take a look at the complete set of projects introduced in Sec. 4.2. Applying the measurements defined in Sec. 4.3 empirical support for Hypotheses 2 and 3 is provided.

5.1. *The case of eclipse*

Eclipse is an Open Source integrated development environment (IDE) developed by IBM. With over 20,000 classes, it is one of the largest Open Source projects written in Java. Furthermore, according to a survey conducted by QA-Systems,[d] Eclipse is the market leader in the Java IDE sector. With 45% market share it is clearly ahead of its closest competitor, JBuilder (16%). Yet, more interesting in the context of this paper is the fact that Eclipse is considered a show case of good software engineering. The leading software architect of Eclipse is Eric Gamma, one of the founders of pattern-oriented software engineering [18]. As Eclipse is a multi-functional IDE intended to work for any programming language and any programming-related activity, the whole architecture is designed with flexibility and extendibility in mind [17]. In fact, the open architecture attracted a large community of developers contributing a vast variety of extensions[e] and supports development in many different languages (Java, C++, Phython etc.), manages SVN and CVS repositories, supports user interface design, UML modeling and database management (SQL).

To check Hypothesis 1, the following question needs to be answered: "Does Eclipse exhibit a significantly higher or lower interdependency between classes, compared to other projects?" The answer is quite unmistakable: Eclipse has the second highest value of d in the set of analyzed projects: The average number of dependencies per class d of Eclipse is 8.9; the mean of d in the sample is 4.1 with a standard deviation of 1.8. As already mentioned, Eclipse is the by far the largest project in our sample (see Appendix A). Before drawing a conclusion from the high value of d, the size effect needs to be excluded. Important question: Is the average degree in the dependency network growing with size?

The dependencies in Eclipse, however, do grow only slightly super linear with the number of nodes (see Fig. 7). Fitting the growth to $f(x) = ax^b$ with least squares gives $a = 3.36$ and $b = 1.09$. For a more in depth treatment see [19], and for a mathematical modeling approach to growth in software dependency networks see [63].

This slight growth of d does not change the finding, though. Even in March 2002 when Eclipse had only 7467 classes — comparable in size to the other projects in the sample set — it already had 57,762 dependencies and thus a d of 7.7. Eclipse

[d]Data collected from the last week of August to the end of September 2003. See http://www.qa-systems.com/products/qstudioforjava/ide_marketshare.html
[e]see the Eclipse plug-in portal: http://www.eclipseplugincentral.com/

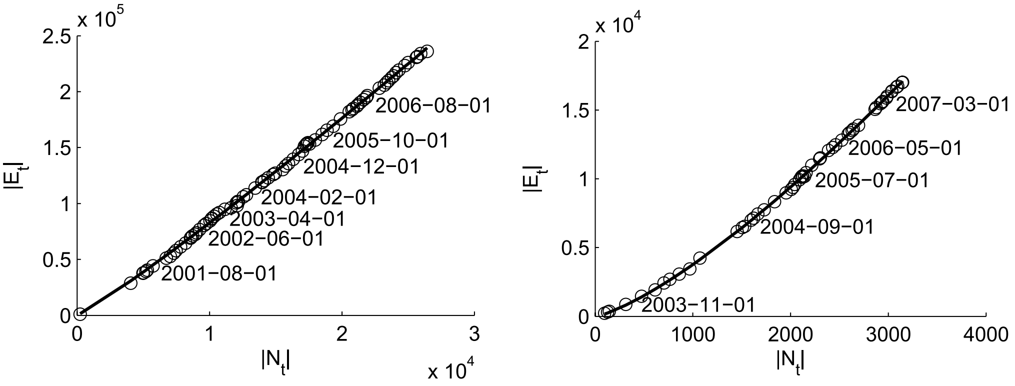

Fig. 7. The evolution of network density expressed by the number of nodes $|N_t|$ versus the number of edges $|E_t|$. Left: Eclipse. Right: Azureus. Circles represent data points, the solid lines represent fits of equation $f(x) = ax^b$ to the data.

back then would have equally been the project with the second highest value of d in the set of analyzed projects. Therefore, the high value of d cannot be explained by the large size of Eclipse. A high d is inherent in the design of Eclipse.

The conclusion is that flexibility and extendibility are compatible with a high degree of interdependency as Eclipse evidences. We might even suspect that they go hand in hand. A thought further elaborated on in the next section.

5.2. *Evidence across 35 Java projects*

To test Hypotheses 2 and 3 a comparative analysis of the whole set of 35 projects was conducted. First, the question "To which degree does change propagation occur?" will be addressed. It is linked to Hypothesis 2. Next, we look at the link between dependency and the share of multi-class changes to test Hypothesis 3.

5.2.1. *To which degree does change propagation occur?*

To answer this question and thus verify Hypothesis 2 $P(\mathcal{C}|\mathcal{D})$ and $P(\mathcal{C}|\neg\mathcal{D})$, defined in Eqs. (4) and (5), need to be compared. Table 1 shows the results for all projects in the sample. To recapitulate, $P(\mathcal{C}|\mathcal{D})$ is the probability that two classes, connected by a dependency, are at least once modified together. $P(\mathcal{C}|\neg\mathcal{D})$ is the probability that two independent classes are at least once modified simultaneously. The error margins of the estimates are given on a confidence level of 99% under the assumption of a binomial distribution.

It can be seen that less than half of the classes linked by a dependency, change together at least once in their common history. This is significantly higher than the values for $P(\mathcal{C}|\neg\mathcal{D})$, which, except for one outlier, lie between 0.1% and 7.1%. With respect to Hypothesis 3, this result means two things: First, change propagation along dependencies exists even though the system is modular. Second, not all

Table 1. $P(\mathcal{C}|\mathcal{D})$ and $P(\mathcal{C}|\neg\mathcal{D})$ for each project. For the definitions see Eqs. (4) and (5) in Sec. 4.3. The error margins are given on a confidence level of 1% under the assumption of a binomial distribution.

Project	$P(\mathcal{C}\|\mathcal{D})$	$P(\mathcal{C}\|\neg\mathcal{D})$	Project	$P(\mathcal{C}\|\mathcal{D})$	$P(\mathcal{C}\|\neg\mathcal{D})$
architecturware	$33.2_{\pm 1.3}$	$0.5_{\pm 0.1}$	jpox	$35.1_{\pm 1.2}$	$0.8_{\pm 0.1}$
aspectj	$57.8_{\pm 1.6}$	$18.5_{\pm 0.1}$	openhre	$34.6_{\pm 2.0}$	$1.1_{\pm 0.1}$
azureus	$38.2_{\pm 1.3}$	$1.2_{\pm 0.1}$	openjacob	$24.0_{\pm 1.3}$	$1.2_{\pm 0.1}$
cjos	$36.3_{\pm 1.1}$	$0.2_{\pm 0.1}$	openuss	$26.8_{\pm 1.5}$	$1.6_{\pm 0.1}$
composestar	$49.0_{\pm 2.0}$	$1.7_{\pm 0.1}$	openxava	$28.0_{\pm 1.8}$	$2.0_{\pm 0.1}$
diee-mad	$23.0_{\pm 2.4}$	$1.1_{\pm 0.1}$	pelgo	$18.1_{\pm 1.3}$	$0.3_{\pm 0.1}$
eclipse	$21.5_{\pm 0.3}$	$0.3_{\pm 0.1}$	personalaccess	$29.8_{\pm 1.9}$	$1.3_{\pm 0.1}$
enterprise	$16.5_{\pm 1.3}$	$0.5_{\pm 0.1}$	phpeclipse	$49.5_{\pm 1.9}$	$2.0_{\pm 0.1}$
findbugs	$21.1_{\pm 0.7}$	$0.6_{\pm 0.1}$	rodin-b-sharp	$21.8_{\pm 1.0}$	$0.5_{\pm 0.1}$
fudaa	$34.5_{\pm 0.9}$	$1.6_{\pm 0.1}$	sapia	$42.4_{\pm 1.7}$	$1.1_{\pm 0.1}$
gpe4gtk	$28.9_{\pm 1.4}$	$1.1_{\pm 0.1}$	sblim	$12.7_{\pm 0.6}$	$1.5_{\pm 0.1}$
hibernate	$45.6_{\pm 1.3}$	$1.1_{\pm 0.1}$	springframework	$33.6_{\pm 1.2}$	$0.5_{\pm 0.1}$
jaffa	$29.5_{\pm 1.5}$	$0.9_{\pm 0.1}$	squirrel-sql	$27.4_{\pm 1.5}$	$1.2_{\pm 0.1}$
jazilla	$62.0_{\pm 1.6}$	$7.1_{\pm 0.1}$	university	$26.8_{\pm 1.6}$	$4.0_{\pm 0.1}$
jedit	$58.0_{\pm 1.8}$	$1.2_{\pm 0.1}$	xendra	$19.0_{\pm 1.3}$	$1.5_{\pm 0.1}$
jena	$33.0_{\pm 1.1}$	$1.0_{\pm 0.1}$	xmsf	$34.8_{\pm 1.6}$	$2.4_{\pm 0.1}$
jmlspecs	$48.2_{\pm 1.9}$	$3.1_{\pm 0.1}$	yale	$31.5_{\pm 1.5}$	$2.6_{\pm 0.1}$
jnode	$28.0_{\pm 0.8}$	$0.5_{\pm 0.1}$			

dependencies propagate changes. More than half of the dependencies seem indeed to be "*change neutral*". They foster code reuse without polluting the system with higher change effort. This second part is a novel aspect and contrasts with the literature reviewed in Sec. 2.2.

The project `Aspectj` seems to be an outlier: Many classes in its software repository make use of aspect-oriented-programming [33]. This means that many dependencies are generated by the aspect weaver. Such dependencies are invisible to the analysis presented in this paper and my thus distort the result.

5.2.2. *How is interdependency linked to flexibility?*

Hypothesis 2 formulated the conjecture that higher interdependence (higher d) is associated with more small changes and less big changes (lower m). Figure 8 shows a scatter plot of d against m based on the 35 projects analyzed. A calculation of the Pearson product-moment correlation coefficient supports this argumentation: the average degree in the class dependency network d and the frequency of multi-class changes m exhibit negative correlation. The correlation coefficient r being -0.49 while $p < 0.002$. Thus, the probability of this correlation being generated by chance by two uncorrelated random variables is below 1%. The error margins of r are $[-0.70, -0.18]$ based on a 95% confidence level. Therefore, the negative correlation is significant. The gray line serves as orientation. It was fitted to the data with the least squares method. The conclusion is that a higher interdependency does not necessarily lead to a higher frequency of change avalanches. Quite the contrary, a high level of interdependency between classes is, according to the analyzed

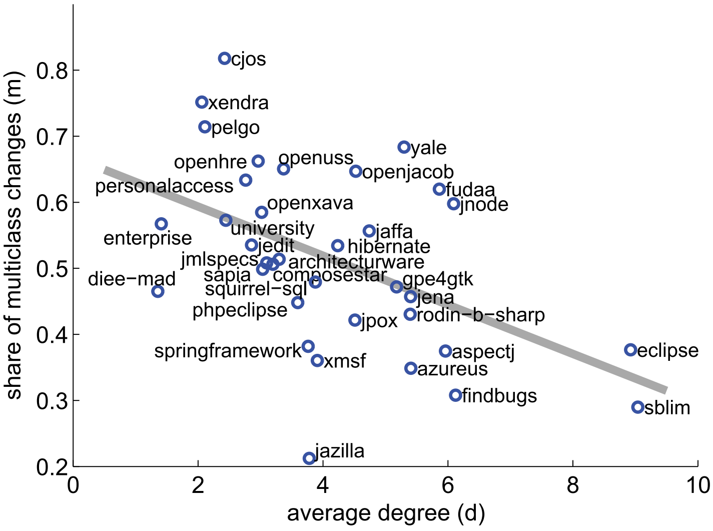

Fig. 8. Link density (d) versus the frequency of multi-class changes (m). See Eqs. (2) and (3) in Sec. 4.3.

data, associated with less multi-class changes. Therefore, in software, the positive effects of dependency seem to outweigh the negative ones, as argued, based on theoretical considerations in Sec. 2. Namely: reuse, while causing more dependency, actually makes small changes with high impact possible. Thus, a modification at a single location (class) is sufficient to add new functionality. See Figs. 2 and 3 in particular.

5.3. *Limitations*

While the theoretical discussion was held as general as possible, the empirical part focused on software architecture. Caution is thus advised when generalizing the results to all possible systems, eligible to the dependency-based view. Software systems have certain idiosyncratic properties: The most salient one is that in a software system no part is repeated. Why should it? If it is needed by more than one other part, we make a module out of it which can be reused as many times as needed. This starkly contrasts with physical systems in which repeated parts abound. The here presented analysis is probably not valid for such physical systems with repeated parts. Systems that share the uniqueness of parts with software systems are, for example, task networks such as the ones described by Hippel [66], division of labor as it is formalized by Marengo and Dosi [43], organizations as described by Rivkin and Siggelkow [52], and Langlois [34]. Particular systems *not* sharing uniqueness of parts with software systems are for example the physical systems discussed by Huang and Kusiak [28], and Ulrich [65].

6. Conclusion

Modularity is a lively field of research, yet scholars focused mainly on modularity and dependency on the micro-level. In this paper, it was argued that once a system is modularized, the dependencies between the modules form a *new* level, the macro-level. Further, it was argued that dependencies play a different role on this level. Whereas on the micro-level they are strong and add complexity to the system; on the macro-level dependencies are weak and a manifestation of functional reuse. This leads to a counterintuitive conclusion: higher dependency goes hand in hand with higher flexibility.

Section 5 presented empirical evidence for this reasoning. First, an analysis of the prestigious Eclipse project showed that high dependency is compatible with or even necessary for good system design and high flexibility. Second, a comparative analysis of 35 more projects showed that more than half of all dependencies between classes are never involved in propagated changes: they are "change neutral". This means that more than half of the dependencies are irrelevant for the project's change dynamics and that methods estimating flexibility and change costs based on the complete set of dependencies consequently will fail. This result also suggests, that the negative role of dependencies is overestimated in management literature. Finally, the analysis revealed, that propagated changes are not more frequent in systems with high interdependency as common wisdom predicts. Quite the contrary: there is a significant negative correlation. Nevertheless, a certain percentage of the dependencies is a vector of change propagation. Blindly adding dependencies to a system will therefore hardly lead to better design. It is essential to add the right dependencies. Guidance in this respect might come from metrics which take further structural aspects into account. For instance the Q metric by Newman [46] has found recent application in the context of software dependency [71, 72]. There are also a number of software specific metrics which might shed more light on the nature of dependencies. An overview of software specific metrics is provided by Gilb [21]. Establishing a link between these metrics and the part of the dependencies which foster change propagation seems a promising line of future research.

As a conclusion, the ambivalent role of dependencies — as a vector of change propagation as well as functional reuse — suggests that we are confronted with a tradeoff between independence and module reuse on the macro-level. A similar tradeoff question concerning not dependency but modularity has already been discussed in the management literature. Ulrich and Tung [64], Welch and Waxman [69], and Brusoni *et al.* [9] pointed out possible negative side effects of modularity and argued for balancing advantages and disadvantages. A similarly balanced view seems also to be appropriate in the case of dependencies.

Acknowledgments

The author acknowledges financial support by the Swiss National Science Foundation (SNSF) (Grant CR12I1_125298).

Appendix A. Projects

Project	Change history	Change events	Nodes	Edges
architecturware	2003-08-20–2008-02-04	25,005	4725	14,637
aspectj	2002-12-16–2008-01-26	18,864	1735	10,342
azureus	2003-07-07–2008-01-29	38,408	3147	17,012
cjos	2000-10-04–2008-01-29	62,244	9030	21,902
composestar	2003-05-30–2006-11-18	8554	2358	7156
diee-mad	2005-07-28–2008-07-03	5780	2594	3517
eclipse	2001-04-28–2007-11-02	371,460	26,444	235,952
enterprise	2002-08-12–2008-02-04	28,970	6702	9467
findbugs	2003-03-24–2008-02-02	23,350	7107	43,511
fudaa	2002-11-25–2008-05-14	55,594	6033	35,364
gpe4gtk	2005-07-05–2007-05-31	19,010	2195	11,365
hibernate	2001-11-27–2006-02-27	44,955	3898	16,512
jaffa	2001-11-07–2008-01-26	12,402	2300	10,901
jazilla	1998-05-06–2004-07-07	8734	2764	10,447
jedit	1999-12-07–2007-04-05	14,375	2973	8489
jena	2000-06-22–2008-01-28	47,391	3931	21,236
jmlspecs	2002-02-18–2008-01-28	14,708	2380	7849
jnode	2003-05-12–2006-06-18	48,285	6677	40,663
jpox	2003-07-21–2007-11-23	45,374	3888	17,539
openhre	2004-10-14–2008-01-31	7720	2288	6777
openjacob	2006-10-16–2008-07-01	7886	2966	13,423
openuss	2000-05-29–2007-03-24	12,238	3074	10,358
openxava	2004-11-02–2008-02-01	32,876	2386	7201
pelgo	2006-04-14–2008-02-19	9012	4837	10,199
personalaccess	2004-09-13–2008-05-22	9972	2349	6490
phpeclipse	2002-07-11–2008-01-06	11,109	2196	7896
rodin-b-sharp	2004-04-28–2008-07-04	31,065	3949	21,309
sapia	2002-11-27–2008-06-21	16,076	2977	9518
sblim	2001-06-20–2008-06-30	36,053	4472	40,423
springframework	2003-02-06–2008-02-01	55,732	4900	18,433
squirrel-sql	2001-06-01–2008-07-03	15,794	2808	10,887
university	2005-01-02–2006-12-11	3623	3548	8677
xendra	2006-05-19–2007-08-01	5636	5010	10,322
xmsf	2003-07-22–2008-07-01	16,745	2516	9836
yale	2002-03-08–2008-01-28	41,836	2197	11,641

References

[1] Alchian, A., Uncertainty, evolution, and economic theory, *J. Polit. Econ.* **58** (1950) 211.

[2] Alexander, C., Ishikawa, S. and Silverstein, M., *A Pattern Language*: *Towns*, *Buildings*, *Construction* (Oxford University Press, 1977).

[3] Alexander, C. W., *Notes on the Synthesis of Form* (Harvard University Press, 1970).

[4] Altenberg, L., NK fitness landscapes, in *Handbook of Evolutionary Computation*, Back, T., Fogel, D. and Michalewicz, Z. (eds.), Chap. B2.7.2 (IOP Publishing Bristol, UK, 1997).

[5] Baldwin, C. and Clark, K., Managing in an age of modularity, *Harvard Bus. Rev.* **75** (1997) 84–93.

[6] Baldwin, C. Y. and Clark, K. B., *Design Rules*: *The Power of Modularity*, Vol. 1 (MIT Press, Cambridge, Massachusetts, USA, 1999).

[7] Bohner, S. and Arnold, R., *Software Change Impact Analysis* (IEEE Computer Society Press, 1996).

[8] Booch, G., *Object-Oriented Analysis and Design with Applications* (Benjamin-Cummings Publishing Co., Inc. Redwood City, CA, USA, 1993).

[9] Brusoni, S., Marengo, L., Prencipe, A. and Valente, M., The value and costs of modularity: A cognitive perspective, *Eur. Manag. Rev.* **4** (2004) 121–132.

[10] Challet, D. and Lombardoni, A., Bug propagation and debugging in asymmetric software structures, *Phys. Rev. E* **70** (2004) 046109.

[11] Clarkson, P., Simons, C. and Eckert, C., Predicting change propagation in complex design, *J. Mech. Des.* **126** (2004) 788.

[12] Cyert, J. G. and March, R., *A Behavioural Theory of the Firm* (Prentice-Hall, Englewood Cliffs, NJ, 1963).

[13] DeMarco, T. and Lister, T. R., *Peopleware* (Dorset House Publishing, NY, USA, 1987).

[14] Eppinger, S. D., Whitney, D. E., Smith, R. P. and Gebala, D. A., A model-based method for organizing tasks in product development, *Res. Eng. Des.* **6** (1994) 1–13.

[15] Ethiraj, S. K. and Levinthal, D., Modularity and innovation in complex systems, *Manag. Sci.* **50** (2004) 159–173.

[16] Frenken, K., A fitness landscape approach to technological complexity, modularity, and vertical disintegration, *Struct. Change Econ. Dyn.* **17** (2006) 288–305.

[17] Gamma, E. and Beck, K., *Contributing to Eclipse*: *Principles*, *Patterns*, *and Plugins*, 1st edn. (Addison Wesley Longman Publishing, Redwood City, CA, USA, 2003).

[18] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns*: *Elements of Reusable Object-Oriented Software* (Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, USA, 1995).

[19] Geipel, M. M., Dynamics of communities and code in open source software, Ph.D. thesis, ETH (2009).

[20] Geipel, M. M. and Schweitzer, F., The link between dependency and co-change: Empirical evidence, *IEEE Trans. Softw. Eng.* **99** (2011).

[21] Gilb, T., *Software Metrics* (Winthrop Publishers, Cambridge, Massachusetts, USA, 1977).

[22] Gorshenev, A. A. and Pis'mak, Y. M., Punctuated equilibrium in software evolution, *Phys. Rev. E* **70** (2004) 067103.

[23] Gosling, J., Joy, B., Steele, G. and Bracha, G., *Java*(*TM*) *Language Specification*, 3rd edn. (Addison-Wesley Professional, 2005).

[24] Guth, R. A., Battling Google, Microsoft changes how it builds software, *Wall Street J.* (2005).

[25] Haefliger, S., von Krogh, G. and Spaeth, S., Code reuse in open source software, *Manag. Sci.* **54** (2008) 180.

[26] Hassan, A. and Holt, R., Predicting change propagation in software systems, in *Proc. 20th IEEE Int. Conf. Software Maintenance* (2004), pp. 284–293.

[27] Horwitz, S., Reps, T. and Binkley, D., Interprocedural slicing using dependence graphs, *ACM Trans. Program. Lang. Syst.* (*TOPLAS*) **12** (1990) 26–60.

[28] Huang, C. and Kusiak, A., Modularity in design of products and systems, *IEEE Trans. Syst. Man Cybern. A* **28** (1998) 66–77.

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

[30] Kahneman, D., Maps of bounded rationality: Psychology for behavioral economics, *Am. Econ. Rev.* **93** (2003) 1449–1475.
[31] Kauffman, S., *The Origins of Order* (Oxford University Press, New York, 1993).
[32] Kauffman, S. A., Lobo, J. and Macready, W. G., Optimal search on a technology landscape, *J. Econ. Behav. Organ.* **43** (2000) 141–166.
[33] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J., Aspect-oriented programming, in *Proc. 11th European Conf. Object-Oriented Programming* (*ECOOP 1997*), eds. Akşit, M. and Matsuoka, S. (Springer-Verlag, 1997), ISBN 3-540-63089-9, pp. 220–242.
[34] Langlois, R. N., Modularity in technology and organization, *J. Econ. Behav. Organ.* **49** (2002) 19–37.
[35] Langlois, R. N. and Robertson, P. L., Networks and innovation in a modular system: Lessons from the microcomputer and stereo component industries, *Res. Policy* **21** (1992) 297–313.
[36] Levinthal, D. A., Adaptation on rugged landscapes, *Manag. Sci.* **43** (1997) 934–950.
[37] Levinthal, D. A. and Warglien, M., Landscape design: Designing for local action in complex worlds, *Organ. Sci.* **10** (1999) 342–357.
[38] Lieberherr, K. J. and Holland, I. M., Assuring good style for object-oriented programs, *IEEE Softw.* **6** (1989) 38–48.
[39] Loyall, J. and Mathisen, S., Using dependence analysis to support the software maintenanceprocess, in *Proc. Conf. Software Maintenance, CSM-93* (IEEE Computer Society Washington, DC, USA, 1993), pp. 282–291.
[40] MacCormack, A., Rusnak, J. and Baldwin, C. Y., Exploring the structure of complex software designs: An empirical study of open source and proprietary code, *Manag. Sci.* **52** (2006) 1015–1030.
[41] March, J., *A Primer on Decision Making*: *How Decisions Happen* (Free Press New York, 1994).
[42] March, J. and Simon, H., *Organizations* (John Wiley & Sons Inc, 1958).
[43] Marengo, L. and Dosi, G., Division of labor, organizational coordination and market mechanisms in collective problem-solving, *J. Econ. Behav. Organ.* **58** (2005) 303–326.
[44] Meyer, B., *Object-Oriented Software Construction* (Prentice Hall PTR, 2000).
[45] Nelson, R. and Winter, S., *An Evolutionary Theory of Economic Change* (Harvard University Press, 1982).
[46] Newman, M. E. J., Modularity and community structure in networks, *Proc. Natl. Acad. Sci.* **103** (2006) 8577.
[47] Orton, J. D. and Weick, K. E., Loosely coupled systems: A reconceptualization, *Acad. Manag. Rev.* **15** (1990) 203–223.
[48] Parnas, D. L., On the criteria to be used in decomposing systems into modules, *Commun. ACM* **15** (1972) 1053–1058.
[49] Parnas, D. L., Designing software for ease of extension and contraction, in *ICSE '78*: *Proc. 3rd Int. Conf. Software Engineering* (IEEE Press, Piscataway, NJ, USA, 1978), pp. 264–277.
[50] Porter, M., *Competitive Advantage*: *Creating and Sustaining Superior Performance*, 1st edn. (Free Press, 1998).
[51] Rajlich, V., A model for change propagation based on graph rewriting, in *Proc. Int. Conf. Software Maintenance* (1997), pp. 84–91.
[52] Rivkin, J. W. and Siggelkow, N., Balancing search and stability: Interdependencies among elements of organizational design, *Manag. Sci.* **49** (2003) 290–311.
[53] Rumbaugh, J. E., Blaha, M. R., Premerlani, W. J., Eddy, F. and Lorensen, W. E., *Object-Oriented Modeling and Design* (Prentice-Hall, 1991).

[54] Ryder, B. G. and Tip, F., Change impact analysis for object-oriented programs, in *PASTE '01*: *Proc. 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (ACM, New York, NY, USA, 2001), ISBN 1-58113-413-4, pp. 46–53, doi:http://doi.acm.org/10.1145/379605.379661.

[55] Sanchez, R. and Mahoney, J. T., Modularity, flexibility, and knowledge management in product and organization design, *Strateg. Manag. J.* **17** (1996) 63–76.

[56] Sangal, N., Jordan, E., Sinha, V. and Jackson, D., Using dependency models to manage complex software architecture, in *OOPSLA '05*: *Proc. 20th Annual ACM SIGPLAN Conf. Object-Oriented Programming*, *Systems*, *Languages*, *and Applications* (ACM, NY, USA, 2005), ISBN 1-59593-031-0, pp. 167–176.

[57] Schumpeter, J., *Theorie der Wirtschaftlichen Entwicklung* (Springer, 1912).

[58] Simon, H., The architecture of complexity, *Proc. Am. Philos. Soc.* **106** (1962) 467–482.

[59] Simon, H. A., A behavioral model of rational choice, *Q. J. Econ.* **69** (1955) 99–118.

[60] Simon, H. A., Near decomposability and the speed of evolution, *Indus. Corp. Change* **11** (2002) 587–599.

[61] Steward, D. V., The design structure system — A method for managing the design of complex systems, *IEEE Trans. Eng. Manag.* **28** (1981) 71–74.

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

[63] Tessone, C. J., Geipel, M. M. and Schweitzer, F., Sustainable growth in complex networks, *Europhys. Lett.* **96** (2011) 58005.

[64] Ulrich, K. and Tung, K., Fundamentals of product modularity, in *Proc. 1991 ASME Winter Annual Meeting Symposium on Issues in Design/Manufacturing Integration*, Vol. 39 (American Society of Civil Engineers, 1991), pp. 73–79.

[65] Ulrich, K. T., The role of product architecture in the manufacturing firm, *Res. Policy* **24** (1995) 419–440.

[66] von Hippel, E., Task partitioning: An innovation process variable, *Res. Policy* **19** (1990) 407–418.

[67] von Krogh, G. and von Hippel, E., The promise of research on open source software, *Manag. Sci.* **52** (2006) 975–983.

[68] Weinberger, E. D., Local properties of Kauffman's N-k model: A tunably rugged energy landscape, *Phys. Rev. A* **44** (1991) 6399–6413.

[69] Welch, J. J. and Waxman, D., Modularity and the cost of complexity, *Evolution* **57** (2003) 1723–1734.

[70] Wright, S., The roles of mutation, inbreeding, crossbreeding and selection in evolution, *Proc. XI Int. Congr. Genetics* **8** (1932) 209–222.

[71] Zanetti, M. S., The co-evolution of socio-technical structures in sustainable software development: Lessons from the open source software communities, in *Proc. ICSE 2012*, *Doctoral Symposium* (2012), pp. 1587–1590.

[72] Zanetti, M. S. and Schweitzer, F., A network perspective on software modularity, in *GI-Edition — Lecture Notes in Informatics* (*LNI*), *Proc. P-200*, *ARCS 2012 Workshops* (2012), pp. 175–186.

[73] Zhang, Y., Gershenson, J. K. and Prasad, G. J., Product modularity: Measures and design methods, *J. Eng. Des.* **15** (2004) 33–51.