Can We Avoid High Coupling?

Conference Paper · July 2011

DOI: 10.1007/978-3-642-22655-7_10 · Source: DBLP

CITATIONS
27

READS
169

3 authors:

Craig Taube-Schock
The University of Waikato
11 PUBLICATIONS **84** CITATIONS
SEE PROFILE

Robert James Walker
The University of Calgary
74 PUBLICATIONS **1,912** CITATIONS
SEE PROFILE

Ian Witten
The University of Waikato
543 PUBLICATIONS **69,849** CITATIONS
SEE PROFILE

Can We Avoid High Coupling?

Craig Taube-Schock[1], Robert J. Walker[2], and Ian H. Witten[1]

[1] University of Waikato, Hamilton, New Zealand
[2] University of Calgary, Calgary, Canada

Abstract. It is considered good software design practice to organize source code into modules and to favour within-module connections (*cohesion*) over between-module connections (*coupling*), leading to the oft-repeated maxim "low coupling/high cohesion". Prior research into network theory and its application to software systems has found evidence that many important properties in real software systems exhibit approximately *scale-free structure*, including coupling; researchers have claimed that such scale-free structures are ubiquitous. This implies that *high coupling must be unavoidable*, statistically speaking, apparently contradicting standard ideas about software structure. We present a model that leads to the simple predictions that approximately scale-free structures ought to arise both for between-module connectivity and overall connectivity, and not as the result of poor design or optimization shortcuts. These predictions are borne out by our large-scale empirical study. Hence we conclude that high coupling is *not* avoidable—and that this is in fact quite reasonable.

1 Introduction

We have long heard the maxim of "high cohesion/low coupling" in software design. It is generally believed that high coupling—that is, high levels of between-module connectivity—particularly signals poor design, as it leads to greater difficulties in modification, comprehension, and parallel development [31,2,28,32,38,17]. Unfortunately, while software can be poorly created with definitely excessive coupling, it is not immediately clear whether high coupling *can* be definitively eliminated in all circumstances.

Analysis of complex networks has revealed the presence of (approximately) *scale-free structure* in networks from a wide variety of fields [27]. A scale-free network is one that has a power-law degree distribution [1,3], characterized by having a majority of nodes involved in few connections and a few nodes involved in many connections. Roughly speaking, an approximately scale-free distribution would manifest itself as a straight line on a log–log plot of the connection degree histogram (i.e., number of connections vs. frequency of nodes). Evidence suggests that other distributions produce similar network characteristics [19,8] and that these distributions exist for various relations in procedural and object-oriented software systems [37,26,24,30,35,4,18,15,17,9,23,6,16,14,36,12]. This set of distributions are known as *heavy-tailed* to differentiate the decay characteristic of their probability mass function from that of typical exponential decay;

a significant probability of occurrence exists even at several standard deviations above the mean [8].

While the application of complex networks to the analysis of software has seen much work, it overlooks a surprising consequence of the observed phenomena:

> *The presence of* any *heavy-tailed distribution describing the degree counts in the connectivity network means that there* must *be nodes that are highly coupled, and even nodes that are very highly coupled, relative to the mean level for each system.*

Given the specific distributions observed empirically, highly coupled nodes are commonplace even in an absolute sense.

This is a paradox. We are all well aware that high coupling is something to be avoided. Yet, the mechanics of heavy-tailed distributions are such that—should they be found to be as universal as claimed—high coupling apparently *must* be present. Possible immediate explanations present themselves: perhaps the systems from which the empirical data were collected were poorly designed and thus the conclusions not indicative of good practice; perhaps the researchers made some serious mistake in the data collection or analysis, such as not actually looking at "real" coupling; perhaps we are witnessing an effect from well-known, practical optimizations; perhaps we have to accept that high coupling is actually *necessary* after all.

To resolve this paradox, we begin (in Section 2) by examining the previous work in network theory, especially as it has been applied to software systems; we find some weaknesses in the generality of the empirical results and a few questionable premises and conclusions, but nothing so serious as to resolve the paradox. Next, we present (in Section 3) a model, based largely on existing ideas, for why (approximate) scale-free structure *should* arise in overall connectivity and thus more specifically in between-module connectivity. These simple predictions are then tested in an empirical study, run against 97 open source software systems (the Qualitas Corpus [33]) written in the Java programming language, across granularities ranging from the statement level to the package level. The design of the study and the experimental apparatus are described in Section 4, while the results are analyzed in Section 5. Remaining issues and observations are discussed in Section 6.

This paper makes three contributions: (1) a demonstration that overall connectivity follows a heavy-tailed distribution across the spectrum of granularity for a large number of open-source systems—regardless of maturity, degree of active support, and level of use; (2) a demonstration that between-module connectivity ubiquitously follows a heavy-tailed distribution—and thus highly-coupled nodes are ubiquitous; and (3) an explanatory model as to why having some areas of high coupling is consistent with good software design practices.

2 Scale-Free Structure and Its Application to Software

There has been considerable interest in the study of the structure of networks [27], due especially to the observation that networks derived from "real"

phenomena (as opposed to phenomena derived from the simulation of mathematical models) have a degree distribution that follows a *power law* [1,3,13]. This is in contrast to networks generated using the algorithmic techniques defined by Erdős and Rényi, which possess a Poisson degree distribution [11].

2.1 Power-law distributions

In networks that possess a power-law degree distribution, the probability that a node x has the degree $\deg(x)$ is proportional to $\deg(x)^{-\alpha}$ where $\alpha > 1$: i.e.,

$$p(\deg(x)) = C\ \deg(x)^{-\alpha}, \tag{1}$$

for some normalization constant C chosen to satisfy $\sum_{y=1}^{\infty} C\, y^{-\alpha} = 1$ (because of the definition of probability mass function). In most power-law distributions encountered in practice, $2 \leq \alpha \leq 3$, but this is not always the case [8]. From such distributions, two key connectivity characteristics emerge:

1. The mean connectivity is low relative to the range because the distribution is left-skewed. This indicates that most of the nodes in the system have low connectivity.
2. The range of connectivity has the potential to be several orders of magnitude greater than the mean, depending on the size of the network. Thus, nodes will be present that exhibit high degrees of connectivity with respect to the mean; these nodes will reside in the *heavy tail* of the distribution.

Networks with a power-law degree distribution are called *scale-free* [1,3], due to the fact that they are self-similar at "all" scales.

Transforming Equation 1 to take the logarithms of each side, we arrive at:

$$\log(p(\deg(x))) = -\alpha\ \log(\deg(x)) + \log(C), \tag{2}$$

which presents itself as a straight line on a log–log plot of $\deg(x)$ versus $p(\deg(x))$ (practically, the frequency observed in empirical data). One is thus tempted to perform a linear regression to the log–log plot to determine the parameters of the model. Strictly speaking, this is not a statistically valid procedure for a variety of reasons [8], not least of which is the fact that data drawn from many different distributions can lead to a roughly straight line on a log–log plot. For our purposes, it is enough to note that any of these *heavy-tailed distributions* lead to an inevitable consequence: the probability is surprisingly large that there exist data points in the heavy tail that are multiple standard deviations away from the mean. The lack of such points would actually invalidate the claim that the data follows a heavy-tailed distribution, as an approximately straight line would not be observed on the log–log plot.

It is well-observed [8] that, for values below some threshold $\deg(x) = d_{\min}$, the power law breaks down because there is some minimum natural scale preventing the behaviour from continuing all the way to 0.

2.2 Empirical findings

There have been several investigations into the structure of software systems that have revealed the presence of power-laws and other heavy-tailed distributions. Wheeldon and Counsell [37] examined power-laws in the class coupling relationships within 3 industrial systems for the purpose of using power-law distributions to predict coupling patterns. They examined 5 different class-coupling relationships (inheritance, interface, aggregation, parameter type, and return type) and concluded that not only does each have a power-law distribution but the relationships are independent of each other. Wheeldon and Counsell do not include coupling as a result of method invocation, and no analysis occurs below the class level.

Myers [26], Marchesi et al. [24], Potanin et al. [30], and Gao et al. [12] observed power-laws in both the in-degree and out-degree distributions of modules in a total of 26 different software systems. Baxter et al. [4] examined 56 systems—many of which are also contained in the Qualitas Corpus [33]—for a large set of measures including some coupling measures, but considered them independently from one another. They observed log-normal out-degree distributions, and some specific coupling measures did not match a heavy-tailed distribution in some instances, perhaps hinting at a lack of universality. Jing et al. [18] found power-laws in the measures weighted methods per class (WMC) and coupling between objects (CBO) for 4 open-source software systems. Concas et al. [9] examined 10 properties of 3 software systems and found those properties to have both Pareto and log-normal distributions. Ichii et al. [16] examined 4 measures (including two variants of WMC) on 6 systems, finding that in-degree follows a power law while out-degree follows some other heavy-tailed distribution. Louridas et al. [23] found power-laws present in the dependencies of software libraries, applications, and system calls in the Linux and FreeBSD operating systems and concluded that power-laws are ubiquitous in software systems.

None of the aforementioned investigations considered software systems at the level of statements and variables, limiting the generality of the findings. Some of the investigations did not explicitly plan to investigate coupling. Myers [26] considered only inheritance and aggregation relationships. Concas et al. [9] focused mostly on size measures, but did include a count of method invocations between classes, which they found to conform to a power-law; however, they did not examine other forms of coupling. Gao et al. [12] considered method–method interaction, thereby excluding other class-level coupling measures.

Hyland-Wood et al. [15] examined coupling relationships at differing levels of granularity (package, class, and method level, but not statement level) for 2 separate open source projects over a 15 month period and concluded that scale-free properties were present at all levels of analysis for each snapshot although they note that these properties were approximate in most cases. While demonstrating the relationship of scale-free structure between differing levels of granularity, this study's lowest level of analysis was that of methods.

Vasa et al. [36] noted that many software metrics have a skewed distribution, which makes the reporting of data using central tendency statistics unreliable.

To address this, they recommend adopting the use of the *Gini coefficient*, which has been used in the field of economics to characterize the relative equality of distributions. They examined 46 systems on a variety of measures, where two of the measures are related to coupling (in-degree count and out-degree count). Their findings appear to mimic the findings of Myers [26] and Gao et al. [12] that in-degrees and out-degrees have differing distributions. However, their findings do not address the structure of software at the source code level.

Some of the investigations had confounding factors, which makes them difficult to directly compare with our investigation. Marchesi et al. [24] examined classes in Smalltalk systems, but issues of dynamic binding prevented precise resolution of between-module interactions. To circumvent these issues, dependency relationships that could only be resolved at runtime were approximated using a weighting function, but it is not clear what effect this transformation may have had. Potanin et al. [30] investigated object graphs, which are not directly comparable to class graphs. For example, collection objects may have large numbers of runtime associations that would not be detectable through static analysis. Similarly, the number of instances of each class could skew the total degree distribution as classes with higher numbers of instances would have greater weight in the analysis. It is not clear that scale-free structure in an object graph translates to scale-free structure in its corresponding class graph.

Valverde et al. [34] and Jenkins and Kirk [17] note that nodes with large numbers of dependencies (termed *hubs*) fall in "the set of bad design practices known as antipatterns" [20]; they fail to identify that the ubiquitous presence of heavy-tailed distributions implies the presence of hubs.

2.3 Process models leading to scale-free structure

To offer an explanation as to how a power-law could develop, Barabási and Albert [3] considered the evolution of complex networks as they increased in size and noted that the *preferential attachment* model caused scale-free structure to emerge. In this model, newly added nodes preferentially attach to nodes that have been in the network the longest time, resulting in a structure where most nodes have limited connectivity and only the oldest have high connectivity.

Several criticisms of the preferential attachment model have been put forth, especially as it applies to software systems. Valverde et al. [34] complain that "no design principle explicitly introduces preferential attachment, nor scaling", offering an alternative model based on optimizing designs to minimize the path length between nodes. Unfortunately, their complaint about design principles is largely irrelevant since known design principles are rules of thumb and incomplete. Furthermore, their evaluation is based on the assumption that the systems they look at possess optimal designs—because they have been under development for a long time. This contradicts Lehman's Law of Declining Quality [21] and the community's general experience.

Myers [26] dismisses preferential attachment because it cannot generate the hierarchical structures present in software; he suggests that scale-free structure arises instead from continuous refactoring. But not all software undergoes non-

trivial refactoring, so either his model is false or we would expect there to exist software systems that do not exhibit scale-free structure—he analyzed only 6 industrial systems that had been under development for prolonged periods and hence could be assumed to have undergone at least some refactoring. Keller [19] points out that many different processes can lead to scale-free structure, and that in fact, the necessary constraints are quite meagre.

Jenkins and Kirk [17] state "preferential attachment relies on newly added nodes having prior knowledge of the rest of the network, which seems implausible, since software is built in pieces from a series of sources using various rules for design patterns which do not apply to the finished software graph"—a clearly untenable assertion, since the developer must have prior knowledge of the network in order to select to which parts of it a newly added node should connect.

Chen et al. [6] added a factor to the preferential attachment model that made it less likely that attachment would happen to a node in another module; they fail to consider how modules themselves are added, deleted, or refactored within a system, and they only validate their conclusions against a single (albeit large and important) system. Li et al. [22] accept the preferential attachment model wholeheartedly without addressing Myers's concern that it fails to explain hierarchical structures; they evaluate their conclusions on two systems.

3 Model

It is generally accepted that dependency between programmatic entities within a software system has a direct impact on that system's ease to be changed, understood, and developed in parallel, and that a key indicator of dependency is connectivity between entities [31,2,10,28,32,7,5]. In Section 3.1, we examine background on the interplay between connectivity and evolvability. We use this background to develop a model, in Section 3.2, for why overall connectivity should be expected to possess an approximately scale-free structure. Adding considerations of practical limitations on module sizes leads us to the conclusion that between-module connectivity should also possess an approximately scale-free structure—and thus, that highly-coupled entities must exist in any sizeable system.

3.1 Connectivity and evolvability

Different theoretical models of the relationship between dependency and evolvability were developed by Simon [31] and Alexander [2] (Alexander used the term *adaption*). Simon focused on the structures common to all complex systems while Alexander focused on the design of systems intended to fit a particular problem. Both researchers viewed complex systems as sets of "components" (we will use the term *entities* or *nodes* to avoid further overloading the term "component") that are organized in a hierarchical structure, which interact in a *non-simple* way. Both viewed the evolvability of complex systems as a probabilistic function based on the interdependency of entities.

In these models, the evolvability of a system is a function of its stability with respect to change propagation. To illustrate this point, consider Figure 1(a). Nodes labelled 1, 2, and 3 share mutual dependency due to their structure of connectivity. Should one of the nodes change, the probability of that change propagating through a connection with dependent nodes is determined by a probability distribution function that could, in principle, be determined empirically. Change propagation may necessitate further change, and so on, thereby increasing the number of structural modifications, which increases the time necessary for the system to stabilize. Such change propagation is often called a *ripple effect* [32,38]. Overall system stability is a function of the probability of propagation p and the number of pathways through which propagation can occur.

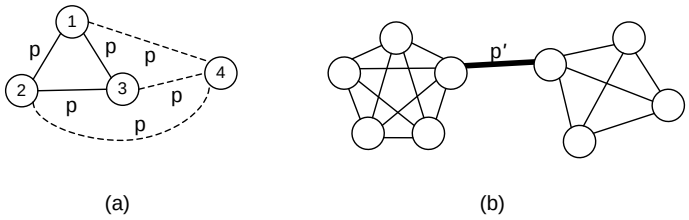

Fig. 1. Evolvability: (a) change propagation; (b) clustering to minimize propagation.

As a system increases in size, the potential for instability caused by ripple effects also increases. In Figure 1(a), the introduction of node 4 introduces three potential new pathways through which change propagation may occur. Since systems require interaction between entities to function, limiting the number of entities, or limiting their ability to interact, may increase a system's stability at the cost of limiting its capabilities. To mitigate the effects of dependency while allowing a system to increase in size requires the use of modularization. Figure 1(b) shows entities grouped into two modules. Entities within each one are free to interact, but interaction between modules is strictly controlled. This structure increases the overall stability of the system by attempting to contain ripple effects within module boundaries.

Whether or not modularization mitigates overall change propagation depends on the probability of propagation between modules (p' in Figure 1(b)). Alexander noted that, in a complex system, the strength of connectivity between entities is not homogeneous [2]. Because of this, he specified that module boundaries be chosen in a way that places entities for which change propagation is high in the same module. This structure minimizes the probability of change propagating between modules.

The principles outlined by Simon and Alexander are echoed in later work by Dijkstra [10] and by Parnas [28,29]. Parnas compared two software systems

that were written to address the same problem. The key difference between the systems was not the selection of individual entities, but rather the criteria used to construct module boundaries. In the first system, modules were constructed by identifying key steps in the overall processing, while for the second they were constructed by using *information hiding* as the primary criterion. The modularizations created in the first system resulted in the sharing of variables between modules. Since the probability of propagation is high between processing statements and variables, shared variables act as a bridge through which change propagation flows between modules. Building module boundaries based upon information hiding, however, encapsulated the interaction between processing statements and variables within modules, thereby constraining the flow of change propagation between modules. Parnas argued that the evolvability (along with the understandability and the ability to construct the modules independently) was higher in the second system due to the structure of its modules.

Recognizing the effects of between-module interaction on evolvability, Stevens et al. [32] coined the term *coupling*. In a modularized system, coupling is defined as "the measure of strength of association established by a connection from one module to another" [32, pp. 233]. In Parnas's example, modularization of the second system exhibited lower coupling than the first and it was therefore deemed to be more evolvable.

3.2 Scale-free structure in overall connectivity

We adopt the preferential attachment model as the starting point for our model, and initially ignore considerations of constraining the maximum allowable connectivity. Barabási and Albert [3] base the notion of preferential attachment on an evolutionary process, in which the probability of attachment increases simply because a node has been in the network longer. We can translate this into more appropriate selection criteria at play in software development that should result in the same overall effect. (1) The probability of attachment will be directly proportional to the usefulness of the functionality provided. The general usefulness of nodes can be expected to vary quite widely; in fact, a node that has proven generally useful in the past is more likely to be generally useful in the future. (2) A developer has to be aware of existing functionality to make use of it. The most commonly used functionality in a system is most likely to be familiar to that developer, his co-workers from whom he is likely to seek help, or any on-line documentation or examples he is likely to encounter. (3) A developer is more likely to use functionality in which he has greater trust, because he or others have used it a lot in the past, or because it has been actively supported for a prolonged time and has acquired a reputation for quality.

Two issues might skew empirically-derived distributions. First, below some minimal scale, insufficient nodes exist for the Law of Large Numbers to hold. Second, developers make errors (contrary to the assumptions of Valverde et al. [34]); a developer might add a spurious connection or fail to add a necessary connection. Whatever the distribution from which these errors would be drawn, their probability is necessarily much lower than that of the correct nodes, so the re-

sult would be a noisy scale-free structure. As a result, we arrive at the following hypothesis:

> **Hypothesis 1***: The overall connectivity network of source code entities for any software system above some minimum size follows an (approximately) scale-free distribution, when no constraints are externally applied to the maximum level of overall connectivity.

Now, we must consider the effect of disallowing any entities to be added to the system with connectivity greater than some value $d_{\max}$. Imagine that, through the standard preferential attachment model, we obtain the first entity e that would normally (in the absence of the constraint) have connectivity of $\deg(e) = d > d_{\max}$. Simply discarding this e is not an option—it was presumably to be added to serve a new purpose within the system. Therefore, we must replace e with some alternative that satisfies the constraint. We can begin by ignoring the constraint and nevertheless insert e into the network, then transform (refactor) the network to again support the constraint.

To replace e, two or more other entities e_i could take its place, each of which (at best) would inherit an independent portion of the connections of e; each of these replacement entities would need at least one connection with another of the replacement entities but as many as one connection with every of the replacement entities. Thus, we have $\deg(e_1) = p_1 d + \hat{p}_1 n$, $\deg(e_2) = p_2 d + \hat{p}_2 n$, $\ldots$, $\deg(e_n) = p_n d + \hat{p}_n n$ where p_i is the fraction of the connections of e inherited by e_i and $\hat{p}_i$ is the fraction of the replacement nodes to be connected to e_i. If any of these replacement entities themselves fail to respect the constraint, the process can recurse. To ensure that this replacement process halts, we can add an additional, simple constraint: that $\deg(e_i) < \deg(e)$ in all cases; thus, progress is made at each iteration and eventually the constraint is satisfied.

To determine specifically to which other entities e_i will be connected, we can return to the original principle of preferential attachment. But preferential attachment is known to result in a scale-free structure in the limit of long time (equivalently, large number of network evolution steps). Thus for any arbitrary $d_{\max}$ and the simple requirement for progress at each replacement step, *a connectivity network with a constraint on its maximum degree will also result in a (different) scale-free structure.* Thus we can revise Hypothesis 1* to eliminate the clause regarding maximum connectivity not being constrained:

> **Hypothesis 1**: The overall connectivity network of source code entities for any software system above some minimum size follows an (approximately) scale-free distribution.

If "high coupling" is defined in terms of number of standard deviations away from the mean, there will thus remain highly coupled entities (ignoring the question of between-module versus within-module connectivity for the moment) after the replacement process—even though the maximum absolute coupling level will have been reduced. However, presumably any arbitrary $d_{\max} \geq 1$ is achievable, and at some point, $d_{\max}$ would be considered "low enough" for practical purposes; thus, "high coupling" would not be universal according to a more absolute

definition. The question then becomes: are there other negative consequences of the replacement process that would tend to prevent an arbitrarily low $d_{\max}$ from being achieved? If so, high coupling would remain, even when defined in absolute terms.

To address this question, consider *inlinks* (inbound connections) and *outlinks* (outbound connections) for all source code entities. In general, outlinking is constrained to be reasonably small. For example, class declarations have few direct superclasses and directly implement few interfaces. Variables are of a single type (or few types in the case of generics) and individual statements tend to be limited to the number of variable access or method invocations due to practical issues, such as style guidelines and the difficulty of reading statements that extend beyond a programmer's screen width. Method declarations have practical limits on the number of return types, the number of parameters, and the number of exceptions that can be thrown by the method. There is, however, no constraint on the number of inlinks that can be made to an entity that has a name within a defined scope. Classes can be used in any number of variable declarations and methods can be invoked from any number of statements. Variables, too, can be used in a variety of different contexts although they will tend to be limited to a stricter scope than that of classes and methods. For these reasons, high connectivity is largely due to inlinks.

A source code entity that exhibits high connectivity is thus likely to do so because of its utilization in multiple contexts. Indeed, use in multiple contexts is a direct side effect of hierarchical structure. Consider a source code entity e such that $\deg(e) > d_{\max}$ and for which the number of connections is largely due to inlinking. To replace e by two or more entities (e_i) in an attempt to satisfy $\deg(e_i) \leq d_{\max}$ would require that all the replacement nodes e_i provide the same utility as e. This suggests the introduction of code clones, which is considered to be poor design. The ability to reuse source code entities suggests that an arbitrarily low $d_{\max}$ cannot be practically achieved.

3.3 Scale-free structure in between-module connectivity

We still have to deal with Myers's concern about preferential attachment being an unsuitable model for software evolution because it does not generate hierarchical structures [26]. While hierarchical structure does not emerge from the preferential attachment model, Myers's analysis does not consider a variation on preferential attachment for which hierarchy is imposed by some external means (such as programming language grammar). As new nodes are added to the network, they will minimally link to a parent node. The programming language syntax and semantics impose constraints on which nodes may act as an acceptable parent based on the type of the node being added. For example, the Java programming language only allows method declarations to be placed within the source code graph as children of a class declaration. This constrains node linkages in a preferential manner, although the preferential probability function differs from that defined in the preferential attachment model [6].

A final question remains: is it reasonable to consider that scale-free structure for overall connectivity necessarily leads to high coupling? Consider the models of Alexander [2] and Simon [31], discussed in Section 3.1. A primary function of modularization is to minimize propagation of change between modules, and to this end, propagation of change within a module can be ignored if the module stabilizes quickly. Module stabilization time is largely affected by limiting module size. Having fewer entities within a module reduces the number of pathways through which change propagation can occur, thereby resulting in pressure to limit the size of modules.

For any source code entity that exhibits high connectivity its links will resolve to other entities contained either in the same module or a different module. If they are resolved within a module, this implies that there are enough entities within the module with which resolution can occur, and that suggests a large module if connectivity is high. Since there is pressure to limit the size of modules, this suggests that high connectivity of source code components is resolved between modules, which represents a form of high coupling. Thus we arrive at:

> **Hypothesis 2***: The between-module connectivity network of source code entities follows a heavy-tailed distribution.

But the same replacement process can be applied to between-module connectivity as was for overall connectivity, with the same constraints. Thus, while between-module connectivity can be reduced, it cannot be practically reduced beyond some minimum level. We can therefore adjust Hypothesis 2*:

> **Hypothesis 2**: The between-module connectivity network of source code entities follows a heavy-tailed distribution, and the degree of left skewness has some maximum level.

Hypothesis 2 (if supported) implies that highly-coupled entities *must* exist for a sizeable system, even when considered in absolute rather than relative terms.

4 Empirical Study

This study comprises an empirical investigation of source code connectivity as observed in practice. The empirical data comes from the Qualitas Corpus [33], a collection of 100 independent open-source software systems written in the Java programming language. The corpus contains at least one version of each independent system, and for some systems multiple versions are present. Since different versions of the same software are not independent, our study only includes one version of each system, specifically the latest one within the corpus, resulting in 100 systems available for study. For three of the systems (eclipse_SDK-3.3.2-win32, myfaces_core-1.2.0, and jre-1.5.0_14-linux-i586), source code was absent from the corpus and thus discarded from the examination set, leaving 97 systems for investigation.

Table 1 provides a truncated view of the systems examined. For each one, counts are reported for: source code entities, connections between entities, modules, classes, methods, statements, and variables. Source code entities include

modules, classes, method declarations, blocks, statements, and variables; each is modelled as a node within a directed graph. Connections between source code entities are modelled as links between nodes; these include parent/child relationships, method invocations, superclass/subclass relationships, superinterfaces, type usage, variable usage, and polymorphic relationships. The systems shown in Table 1 are sorted in descending order by node count; only the top and bottom six systems are presented.

#	Name/Version	Nod	Cnx	Mod	Cls	Mth	Blk	Sta	Var
1	derby-10.1.1.0	318831	809952	135	1805	25067	56357	160555	74910
2	gt2-2.2-rc3	256838	651522	219	3453	26347	52556	106738	67523
3	weka-3.5.8	248704	682151	91	2019	19169	47561	124152	55710
4	jtopen-4.9	230394	593240	18	1940	20559	42206	112259	53410
5	tomcat-5.5.17	177249	433523	149	1777	17152	36247	80214	41708
6	compiere-250d	155379	388859	43	1260	18128	25458	73472	37016
92	jmoney-0.4.4	6310	17618	6	193	713	996	2989	1411
93	nekohtml-0.9.5	6606	17153	7	54	422	1453	2887	1781
94	jchempaint-2.0.12	5757	15844	8	125	419	1146	2696	1361
95	jasml-0.10	5482	15419	8	53	256	895	3011	1257
96	fitjava-1.1	3862	10296	5	96	462	786	1564	947
97	picocontainer-1.3	3771	9117	5	99	540	842	1155	1128

Table 1. Structural measures of the systems that were examined. "Nod" = nodes; "Cnx" = connections; "Mod" = modules; "Cls" = classes; "Mth" = methods; "Blk" = blocks; "Sta" = statements; "Var" = variables.

4.1 Graph-based source code representation

The basis of our analysis is a directed graph representation of source code, where nodes represent source code entities (packages, classes, methods, blocks, statements, and variables), and links (directed arcs) represent connections between entities (hierarchical containment, method invocation, superclass, implementation, type, variable usage, and method overriding). Figure 2 shows the metamodel used in this investigation, similar to that of Mens and Lanza [25].

Our model differs mostly in terms of the level of details provided (Mens and Lanza's metamodel is language-independent and was simplified for readability); however, there are three key structural differences. (1) Our model explicitly defines package and block entities, which are implicit in Mens and Lanza's model. Our reasoning for inclusion of these structural features is that they are important means of structuring in practice, and they could have a significant effect on the connectivity network. (2) Our model is more explicit about containment and hierarchical structure. For example, classes can contain other classes and statements can contain other statements or blocks (such as the code to be executed as part of a loop). This kind of containment definition is particularly relevant

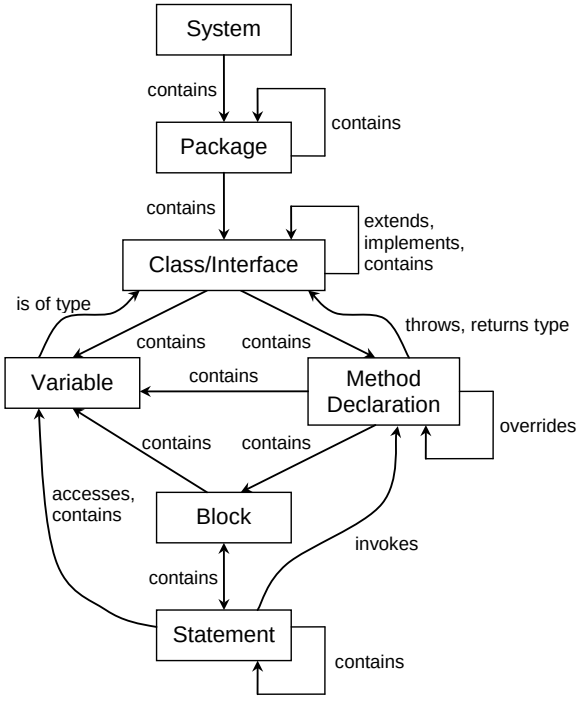

Fig. 2. Metamodel applied in our analysis.

in terms of variable declarations. Our model allows for explicit containment of variables within classes/interfaces (as class and instance variables), method declarations (as parameters), blocks (as scope-limited variables), and statements (e.g., a for-loop counter), where the Mens and Lanza model appears to be focused exclusively on instance variables. (3) Our model supports relationships that do not exist in Mens and Lanza's model. For example, variables have a type (which in our model is represented as a relationship between the variable declaration entity and the associated type entity) and methods have a return type and can specify exceptions that can be thrown from within the method's body. Method invocation is represented as a relationship between a statement and the target method declaration, whereas the Mens and Lanza model represents invocation as an entity contained within a method.

To illustrate the use of this metamodel, we provide a Java source example (Figure 3) and the resulting directed graph (Figure 4). In Figure 4, different node categories are illustrated as different shapes and link categories are shown using different colour and line styles. To improve readability, some relationships shown in the metamodel are excluded from the example (specifically "extends", "implements", "throws", and "overrides"). The example source code contains two class definitions (X and Y), which are represented as Type nodes in the graph, and each class is located within separate packages, p1, and p2. The code con-

tains several references to int, a primitive data type in Java. The code contains three method declarations—m1(), getVar2(), and getValue()—the first two being defined within class X and the last within class Y. To simplify this example, the implementation is provided for only two of the methods.

```
package p1;
public class X {
    private Y var1;
    private int var2;
    public int m1() {
        int temp;
        temp = var1.getValue() + getVar2();
        return temp;
    }
    private int getVar2() {
        return var2;
    }
}

package p2;
public class Y {
    public int getValue() { ... }
}
```

Fig. 3. Sample Java source code listing.

The hierarchical structure[1] of the source code is maintained through Parent links. Child nodes connect to their parent source code entities, and each node can only have one hierarchical parent. For example, class X is in package p1, and method m1() is defined within class X. Specification of type (as is seen in variable declaration and method return type specifications) is represented as a link from the specifying node to the type declaration node. In the example, instance variable var1 (contained in class X) has a reference to class Y, creating a link between node var1 and node Y. Similarly, each of the methods are declared to have an int return type, so there is a link from each method to the node representing the int type. There are two method invocations, which are represented as links between the calling statement node and the called method declaration node. Finally, uses of variables (var1, var2 and temp) are represented as links between the using statement and the used variable declaration node.

This structure is constructed through the standard parser of the Eclipse integrated development environment.[2] The Eclipse parser is robust to all versions

[1] Note that, in this context, "hierarchical" refers to the syntactic hierarchy, and should not be confused with the type hierarchy.

[2] `http://www.eclipse.org`

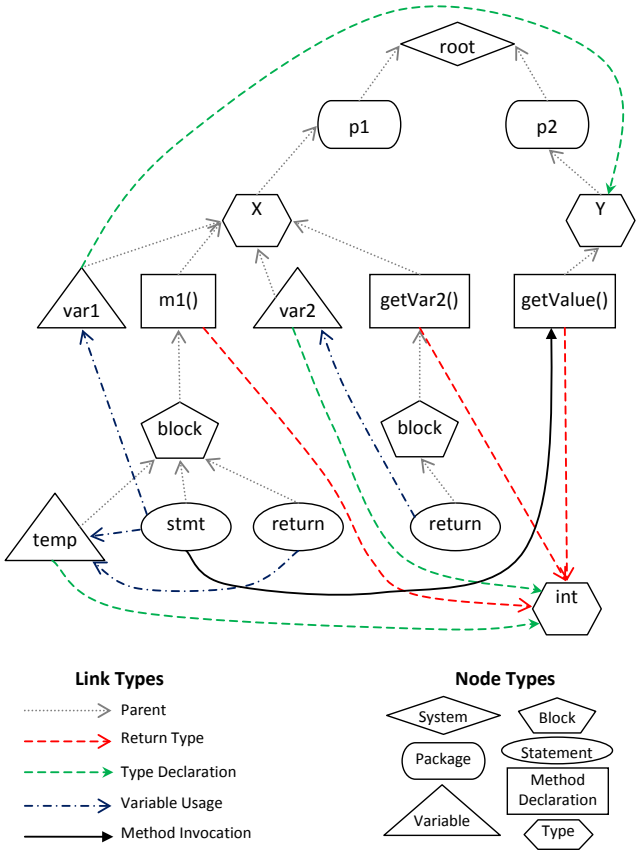

Fig. 4. Directed graph of the sample source code.

of the Java programming language and because it is used in both industrial-strength toolsets and research environments, it provides a reasonable basis upon which we conduct this investigation. The abstract syntax tree (AST) for each compilation unit in a given system is constructed using the parser. The hierarchical relationships in the resulting directed graph are derived from the hierarchical structure of the AST. Each compilation unit is inserted into the hierarchy, according to package definition, under a root node that represents the whole system. The remaining links are derived from the semantic information contained in the AST. In the case of type relationships, the Eclipse parser provides the fully qualified name of all resolved types, which is used to resolve the target node within the directed graph (the source node is implicit to the context of the type relationship). In the case of method invocations, special care is needed to correctly resolve overloaded methods as the fully-qualified names for overloaded methods are the same. To resolve this ambiguity, we extract the fully qualified

name as well as the declared method's signature for each method declaration (which are guaranteed to be unique within the containing class). This signature is used in conjunction with the method's fully qualified name to resolve to the correct method declaration node. Finally, the Eclipse parser cannot provide a fully qualified name for variables that are embedded within blocks because blocks do not provide a namespace for contained variables. However, because the hierarchical structure of the source code is preserved in our directed-graph representation, the scope of any node can be computed. In the case of variable accesses for which the parser has not provided a fully qualified name, the scope of the accessing node is computed and the variable in question is resolved within that scope by unqualified name.

We consider package declarations, class and interface declarations, method declarations, variable declarations, blocks, and statements as the base units of analysis. The form of other substructures, such as expressions and subexpressions, are highly constrained by the syntax of the language, rather than necessarily taking on a form that arises more naturally; thus, we ignore them for the sake of this analysis. To eliminate subexpressions from the data structure, all links for each subexpression are collapsed into the nearest non-expression ancestor node within the structure. For example, the statement `return x * y;` is represented by four nodes in the AST: the return statement, the multiplication expression, and the two variable references. Nodes `x` and `y` have links to the associated variable declaration nodes and the node representing the multiplication has no links (other than hierarchical containment ones). The relationships between `x` and `y` and their respective variable declarations are collapsed into the return statement. Once all subexpression relationships are resolved and collapsed, subexpression nodes are removed from the data structure.

Embedded within the source code structure are polymorphic relationships that are not explicitly identified by the compiler. Specifically, polymorphic method invocation is resolved at runtime: it is a dependency relationship that is implicit within the inheritance structure defined by superclass and subclass relationships. To make this relationship explicit, all overriding method declarations are identified and a link is added between each declaration and all ancestor method declarations in the class and interface hierarchy that have the identical signature. The Java programming language allows for single inheritance of classes but implementation of interfaces, which is supported by our toolset.

Virtually all software systems contain references to externally defined entities (e.g., libraries and programming language types). Since external entities are not part of the system under investigation, they are not considered in the analysis. However, proxy nodes that represent external entities are included in the analytic structure to act as placeholders, thereby allowing consideration of the connections between internal and external entities. For example, variables of type `int` possess a link to an `int` type declaration node even though the `int` type is external to the software system.

4.2 Identification of within-module and between-module links

In object-oriented programming, the key module in a software design is the object, and objects are represented in source code by their classification (class). For the purposes of this analysis, we consider class to be the defining aspect of modular boundaries; alternatives are both possible and desirable targets for analysis, which we discuss further in Section 6. Identification of within-module and between module links is a matter of identifying links that cross class boundaries.

Encapsulated within the analytic structure are the hierarchical relationships for each entity in a software system. Within-module links are those for which both its associated source and target nodes share a common class in their hierarchical ancestry. Figure 5 illustrates examples of within-module and between-module links.

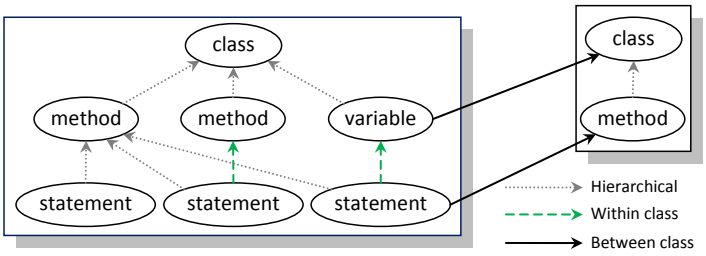

Fig. 5. Identifying within-module and between-module dependency.

Hierarchical relationships introduce a confounding factor into our analysis. These links do not define relationships of direct interaction and because they are overwhelmingly within-module, their inclusion as part of the analysis will skew any comparison in that direction. For this reason, hierarchical links are used to identify structural boundaries that are relevant to the analysis but are not included as part of the computation of degree distributions.

Heavy-tailed distributions are then identified and fit to a power-law model via the informal procedure described in Section 2.1.

5 Analysis

To test Hypothesis 1, the degree distribution for all systems is computed and, in accordance with Section 4.2, hierarchical links are eliminated from the analysis. To test Hypothesis 2, we compute the degree distribution for all systems excluding hierarchical and within-module links. Between-module links are identified using the approach outlined in Section 4.2: links are between-module if their source node and target node do not share the same class in their hierarchical ancestry.

All nodes that have a degree of zero (such as those whose sole dependency is through hierarchical relationships) are removed from the analysis. The resulting distributions are plotted using a log–log scale; the full set is available elsewhere.[3]

5.1 Overall connectivity

For the purposes of discussion, three example plots are chosen based on system size (total node count). The example systems are *derby-10.1.1.0*, *jung-1.7.6*, and *picocontainer-1.3*, which represent the largest, median, and smallest systems, respectively. Figure 6 shows these distributions on a single plot. The similarity in shape is striking: we observe a positive slope between the first two data points, followed by a linear negative trend. Note that each of the distributions is noisy at the right end of the distribution, which is expected because they are produced from discrete data points and naturally have fewer points exhibiting high values.

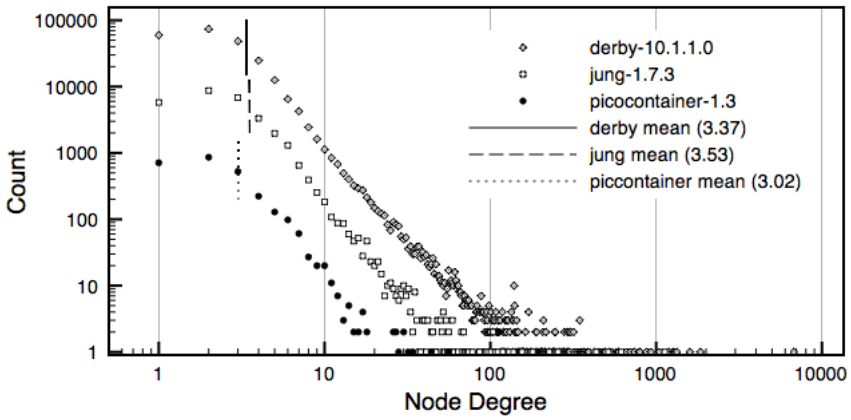

Fig. 6. Distribution of overall connectivity for the example systems.

All the plots exhibit characteristics of heavy-tailed distributions. They are left skewed and have a total range that is at least an order of magnitude larger than the mean. The mean degree for each example system is also shown on Figure 6. To demonstrate this for all systems, the mean and maximum degrees of each system are computed and plotted with a logarithmically-scaled y-axis in Figure 7, and the systems are sorted by descending order of maximum degree. The mean degree over all systems remains relatively constant, while the maximum is roughly between 10 and 1000 times the mean for all systems.

Further observation of the distributions in Figure 6 reveals clear differentiation of the three systems except to the right where the distributions are noisy.

[3] `http://hdl.handle.net/10289/5307`

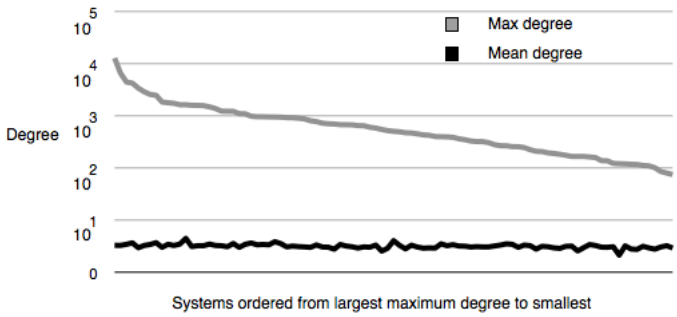

Fig. 7. Mean degree versus maximum degree.

This is consistent with the expectations of a power-law distribution (Equation 1). The probability of a node with a high degree decreases proportionally to the degree; therefore, given a fixed α, the number of nodes with higher degrees increases for systems that have more nodes. Based on these observations, we conclude that Hypothesis 1 is satisfied: overall connectivity for source code entities follows a heavy-tailed distribution for all systems within the corpus.

5.2 Between-module connectivity

The between-module distributions for the three example systems are shown in Figure 8. The between-module distributions show the overall coupling present in each system.

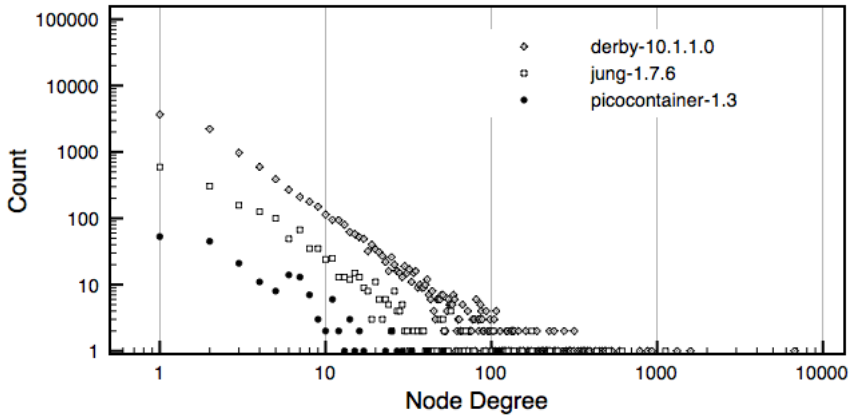

Fig. 8. Degree distributions for the sample systems (between-module only).

Figure 8 demonstrates that the between-module connectivity distributions are similar in shape to those computed to show overall connectivity (Figure 6). The between-module connectivity distributions are less well defined and this is due to the avoidance of between module interaction; less interaction equates with fewer data points, thereby producing noisier distributions. All the between-modules connectivity distributions have similar shape, including a heavy tail.

Figure 9 shows the overall and between-module connectivity distributions plotted together for each of the target systems. For each of the three systems we observe an overlap in the heavy tail. If the links for the nodes that exhibited high overall connectivity were primarily resolved within-module, then we would observe a migration of data points towards the left in the within-module distributions, which would therefore not exhibit a heavy tail. However, this migration is not observed. Instead, we observe heavy tails in both distributions, which overlap when we plot them on the same graph. This demonstrates that the nodes that appear in the heavy tail of the overall connectivity distributions are the same nodes that appear in the heavy tail of the between-module connectivity distributions, and are responsible for the presence of high-coupling.

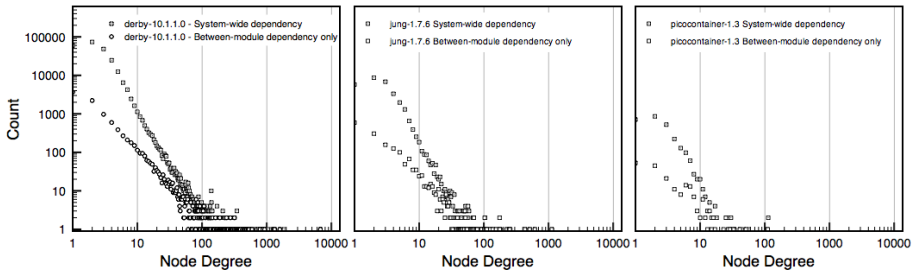

Fig. 9. Comparison of overall and between-module connectivity distributions for the example systems.

In Figure 9, we observe that there is a difference in slope of the linear portions of the distributions. Because the overall connectivity distributions have more data points in the left side of the distribution, the slope in the overall connectivity distributions are steeper than the slope for the corresponding within-module connectivity distributions. Using the process outlined in Section 2.1, we estimate α for all distributions. We use $d_{\min} = 1$ for all between-module distributions and $d_{\min} = 2$ for all overall connectivity distributions; data below the threshold are ignored. Comparison of estimated α between overall and within-module connectivity distributions for all systems (sorted in descending order by largest α estimate for overall connectivity) is shown on Figure 10. Estimated α for between-module connectivity distributions is lower than the overall connec-

tivity distribution for the same system and this is true for all systems. Based on the above analysis, we conclude that Hypothesis 2 is satisfied.

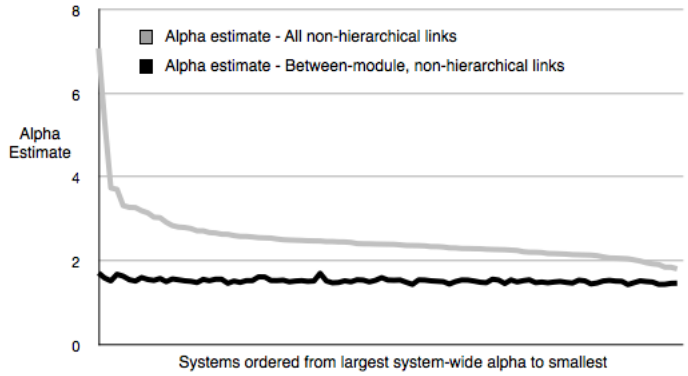

Fig. 10. Comparison of α estimates.

6 Discussion

Here we discuss remaining issues and avenues for further research.

6.1 Threats to validity

Internal validity. It is an important characteristic of these findings that scale-free structure was found to be ubiquitous in the data set. It is important because we do not have *a priori* knowledge about the quality of design of the systems in the corpus and because of this, we have no ground truth from which to argue about the relationship between scale-free structure, high coupling, and the design of software systems. The results demonstrate, however, that high coupling is present in every system in our data set. If we are to accept high coupling as a definitive indicator of poor design, then we would have to conclude that all 97 systems under investigation suffer from poor design. This conclusion, however, seems implausible given the number of systems and the long modification and usage history and maturity of some of the systems. It is likely that at least some of the systems in the corpus are well designed despite the fact that they contain areas of high coupling. To be clear, we do not argue that all the systems in the corpus are well designed, but we argue against the notion that they are all poorly designed and that at least for the systems in the corpus, the presence of high coupling does not distinguish good design from poor. From these findings, we conclude that the presence of high coupling can be consistent with good design.

Construct validity. In Section 3.2, we presented a model of software evolution that is based on preferential attachment. Our model assumes hierarchical structure imposed as a constraint and utilizes a preferential probability function based on node functionality, module structure, and scoping rules. We are quick to point out that our finding of scale-free structure at the source code level does not necessarily imply our model in action. Keller notes that the mere presence of a particular distribution does not imply particular underlying or generational process [19]. However, our model does show that preferential attachment can be modified to be consistent with the evolution of software systems, thereby providing the possibility that our findings may translate to other programming languages and paradigms. If scale-free structure is common at the overall connectivity level, then high coupling is difficult to avoid.

External validity. While this study demonstrates that the presence of high coupling can be consistent with good design practice, we caution against extrapolating the specific structures identified in the examined systems to all software systems. All the systems in this investigation were open source and written using the Java programming language. Based on the structure of our investigation, it may be that the programming paradigm or open source nature of the systems confound our results. Different programming paradigms may produce dependency structures that are different from those generated using an object-oriented paradigm. Similarly, open source software may introduce greater levels of dependency through the desire to appeal to a broad base of users. Although our investigation did not identify any systems that did not contain areas of high coupling, we cannot conclude that a system with such structure does not exist.

The differing functionality, size, maturity, and modification histories of the investigated systems supports some generalizability of these findings. None of the systems under investigation were immune to the hypothesized effects, thereby suggesting that the presence of scale-free structure is independent of these properties.

6.2 Near-constant α for between-module connectivity

From our model, we believe that α for between-module connectivity is obtained through balancing two opposing goals: manageable module sizes and low coupling. While we expected to see distributions that were not extremely left-skewed (as stated in Hypothesis 2), we were surprised at the constancy of α. This may be a sign of an optimal balance that developers are able to achieve. One must recall that this is an exponent, however, and so even small variations can have large effects on the actual data. Even so, the possibility that this is more than coincidence is intriguing, and demands further investigation.

6.3 Varieties of coupling

It has been argued that not all types of coupling are the same. Indeed, Wheeldon and Counsell [37] studied 5 different class-coupling relationships and found them

to be independent. However, they also concluded that all 5 relationships followed a power-law, which suggests that high-coupling exists across those types. One avenue of future work suggests expanding the analysis performed here to account for different types of coupling.

There are cases where the quality of design is reduced as a tradeoff to a more desirable goal, such as performance optimizations. It is possible that some of the coupling detected by our analysis is the result of performance optimization; however, we consider it unlikely that all 97 systems have been subjected to this kind of optimization.

7 Conclusion

We have long heard the maxim of "high cohesion/low coupling" as a basis for good design. Abstract models of evolvability demonstrate why modularization and minimization of between-module connectivity (coupling) are essential to building complex systems: change propagation that would otherwise destabilize an unconstrained network can be contained. We build from the preferential attachment model and standard ideas of modularity to theorize as to why highly-coupled nodes should be expected in real software systems. In our model, we propose a preferential probability function based on entity utility and we argue that the probability of attachment to utility-providing nodes is not uniform because nodes will provide functionality of differing utility.

Using classes to define modules, we studied connectivity in 97 open source software systems using a graph-based analytic framework. Regardless of maturity, size, modification history, and the size of the user community, all these systems exhibit a similar scale-free dependency structure in both the structure of overall connectivity and between-module connectivity (coupling). Our analysis also demonstrated a relationship between highly-connected source code entities and high coupling: entities that exhibited high connectivity were the same entities that participated in areas of high coupling, as these nodes made up the heavy tail of both distributions. The links of highly connected source code entities were not generally resolved within-module, and our model indicates that this is due to practical limits on module sizes. From this, we conclude that scale-free structure in the source code network translates directly to high coupling.

Thus, we conclude that high coupling is impracticable to eliminate entirely from software design. The maxim of "high cohesion/low coupling" is interpreted by some to mean that all occurrences of high coupling necessarily represent poor design. In contrast, our findings suggest that some high coupling is necessary for good design.

Acknowledgments

We would like to thank Robert Akscyn for feedback on the research and paper, the members of the Laboratory for Software Modification Research at the University of Calgary (especially Rylan Cottrell) for their devoted assistance

over the years, and the anonymous reviewers for their diligence and professionalism. This work was funded by a Discovery Grant and a Postgraduate Scholarship (PGS D) from the Natural Sciences and Engineering Research Council of Canada. High-performance computing was provided by the Symphony High-Performance Computing Cluster at the University of Waikato.

References

1. R. Albert, H. Jeong, and A.-L. Barabási. Diameter of the World Wide Web. *Nature*, 401:130–131, 1999.
2. C. Alexander. *Notes on the Synthesis of Form*. Harvard University Press, 1964.
3. A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
4. G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of Java software. In *Proc. ACM Conf. Obj.-Oriented Progr. Syst. Lang. Appl.*, pages 397–412, 2006.
5. L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Sofw. Eng.*, 25(1):91–121, 1999.
6. T. Chen, Q. Gu, S. Wang, X. Chen, and D. Chen. Module-based large-scale software evolution based on complex networks. In *Proc. IEEE Int. Conf. Comp. Info. Technol.*, pages 798–803, 2008.
7. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.
8. A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Rev.*, 51(4):661–703, 2009.
9. G. Concas, M. Marchesi, S. Pinna, and N. Serra. Power-laws in a large object-oriented software system. *IEEE Trans. Softw. Eng.*, 33(10):687–708, 2007.
10. E. W. Dijkstra. Structured programming. In J. N. Buxton and B. Randell, editors, *Software Engineering Techniques*, pages 84–87. NATO Scientific Affairs Division, Brussels, 1970.
11. P. Erdős and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.*, 5:17–61, 1960.
12. Y. Gao, G. Xu, Y. Yang, X. Niu, and S. Guo. Empirical analysis of software coupling networks in object-oriented software systems. In *Proc. IEEE Int. Conf. Softw. Eng. Service Sci.*, pages 178–181, 2010.
13. K.-I. Goh, E. Oh, H. Jeong, B. Kahng, and D. Kim. Classification of scale-free networks. *Proc. Nat. Acad. Sci.*, 99(20):12583–12588, 2002.
14. L. Hatton. Power-law distributions of component size in general software systems. *IEEE Trans. Softw. Eng.*, 35(4):566–572, 2009.
15. D. Hyland-Wood, D. Carrington, and S. Kaplan. Scale-free nature of Java software package, class and method collaboration graphs. Technical Report TR-MS1286, University of Maryland, College Park, 2006.
16. M. Ichii, M. Matsushita, and K. Inoue. An exploration of power-law in use-relation of Java software systems. In *Proc. Australian Conf. Softw. Eng.*, pages 422–431, 2008.
17. S. Jenkins and S. R. Kirk. Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution. *Info. Sci.*, 177:2587–2601, 2007.

18. L. Jing, H. Keqing, M. Yutao, and P. Rong. Scale free in software metrics. In *Proc. Int. Comp. Softw. Appl. Conf.*, 2006.
19. E. F. Keller. Revisiting "scale-free" networks. *BioEssays*, 27(10):1060–1068, 2005.
20. A. Koenig. Patterns and antipatterns. *J. Obj.-Oriented Progr.*, 8(1):46–48, 1995.
21. M. M. Lehman. Laws of software evolution revisited. In *Europ. Wkshp. Softw. Process Technol.*, volume 1149 of *Lect. Notes Comp. Sci.*, 1996.
22. D. Li, Y. Han, and J. Hu. Complex network thinking in software engineering. In *Proc. Int. Conf. Comp. Sci. Softw. Eng.*, pages 264–268, 2008.
23. P. Louridas, D. Spinellis, and V. Vlachos. Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 18(1):2/1–2/26, 2008.
24. M. Marchesi, S. Pinna, N. Serra, and S. Tuveri. Power laws in Smalltalk. In *Proc. Europ. Smalltalk User Group Joint Event*, 2004.
25. T. Mens and M. Lanza. A graph-based metamodel for object-oriented software metrics. *Electr. Notes Theoret. Comp. Sci.*, 72(2), 2002.
26. C. R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Phys. Rev. E*, 68, 046116, 2003.
27. M. Newman, A.-L. Barabási, and D. J. Watts. *The Structure and Dynamics of Networks*. Princeton University Press, 2006.
28. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.
29. D. L. Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng.*, 2(1):1–9, 1976.
30. A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in object-oriented programs. *Commun. ACM*, 48(5):99–103, 2005.
31. H. A. Simon. The architecture of complexity. *Proc. Amer. Phil. Soc.*, 106(6):467–482, 1962.
32. W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Syst. J.*, 13(2):231–256, 1974.
33. E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Proc. Asia-Pacific Softw. Eng. Conf.*, 2010.
34. S. Valverde, R. Ferrer Cancho, and R. V. Solé. Scale-free networks from optimal design. *Europhys. Lett.*, 60(4):512–517, 2002.
35. S. Valverde and R. V. Solé. Logarithmic growth dynamics in software networks. *Europhys. Lett.*, 72(5):858–864, 2005.
36. R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz. Comparative analysis of evolving software systems using the Gini coefficient. In *Proc. IEEE Int. Conf. Softw. Maint.*, pages 179–188, 2009.
37. R. Wheeldon and S. Counsell. Power law distributions in class relationships. In *Proc. IEEE Int. Wkshp. Source Code Analys. Manipul.*, pages 45–54, 2001.
38. F. G. Wilkie and B. A. Kitchenham. Coupling measures and change ripples in C++ application software. *J. Syst. Softw.*, 52(2–3):157–164, 2000.