An Empirical Study of Software Architectures' Effect on Product Quality

Klaus Marius Hansen*,

Department of Computer Science, University of Copenhagen, Njalsgade 128, bygn. 24, 5. sal, 2300 Kobenhavn S, Denmark

Kristjan Jonasson, Helmut Neukirchen

Department of Computer Science, University of Iceland, Saemundargotu 2, 101 Reykjavik, Iceland

Abstract

Software architecture is concerned with the structure of software systems and is generally agreed to influence software quality. Even so, little empirical research has been performed on the relationship between software architecture and software quality. Based on 1,141 open source Java projects, we calculate three software architecture metrics (measuring classes per package, normalized distance, and a new metric introduced by us concerning the excess of coupling degree) and analyze to which extent these metrics are related to product metrics (defect ratio, download rate, methods per class, and method complexity). We conclude that there are a number of significant relationships between product metrics and architecture metrics. In particular, the number of open defects depends significantly on all our architecture measures.

Keywords: Software architecture, metrics, product quality, empirical study

1. Introduction

It is often claimed that software architecture enables (or inhibits) software quality (cf. e.g., Perry and Wolf (1992); Garlan and Shaw (1993)). An example would be that an architectural choice of a specific, relational database for an application implies quality constraints on performance, modifiability etc. However, this claim has not been extensively validated empirically. While much work has focused on measuring software quality, little has focused on measuring software architecture. In the work described here, we investigated the software architecture of open source software projects, defined metrics for software architecture, and analyzed to which extent they correlated with software quality metrics. Specifically, the data that we collected was meta-data on 21,904 projects and source code from 1,570 of these. All projects are Java projects and hosted on the SourceForge[1] repository. Based on the meta-data and source code, we computed and analyzed the results of various metrics. Our objective was to compare software architectures and product quality according to various perspectives on software quality.

Our main contributions are a new metric for modeling of coupling and the actual empirical study of software architectures' effect on product quality including the analysis of relationship between individual metrics using uni- and multi-variate models.

The rest of this article is structured as follows: first, we present some foundations in Section 2. Section 3 presents and discusses metrics on software quality and on software architecture. Next, Section 4 presents our operationalization process, in particular our study method, including how data was gathered and how metrics were calculated. Our analysis is presented in Section 5 and Section 6 summarizes and and concludes our work.

2. Background

Our view on software quality originates in the work of Garvin (1984). Garvin defined a set of views on quality which are also applicable to software (Kitchenham and Pfleeger, 1996). The characteristics of quality in these views are:

- In the *transcendental* view, quality can be recognized but not defined. This is the view that is espoused by Christopher Alexander in his patterns work (Alexander, 1979) and to a certain extent in the software patterns literature (Gamma et al., 1995)
- In the *user view*, a system has high quality if it fulfills the needs of its users. This view is highly related to usability and is in line with "quality in use" as defined in the ISO 9126 standard (ISO/IEC, 2001) as shown in Figure 1
- In the *manufacturing* view, a product is seen as being of high quality if its development conforms to specifications and defined processes. This view is to a certain extent part of CMM(I) (CMMI Product Team, 2006) or SPICE (ISO/IEC, 2004) and to the "process quality" concept briefly mentioned in ISO 9126 as shown in Figure 1. In the sense of conformance to specifications, aspects of

*Corresponding author

Email addresses: klausmh@diku.dk (Klaus Marius Hansen*), jonasson@hi.is (Kristjan Jonasson), helmut@hi.is (Helmut Neukirchen)

[1] http://www.sourceforge.net

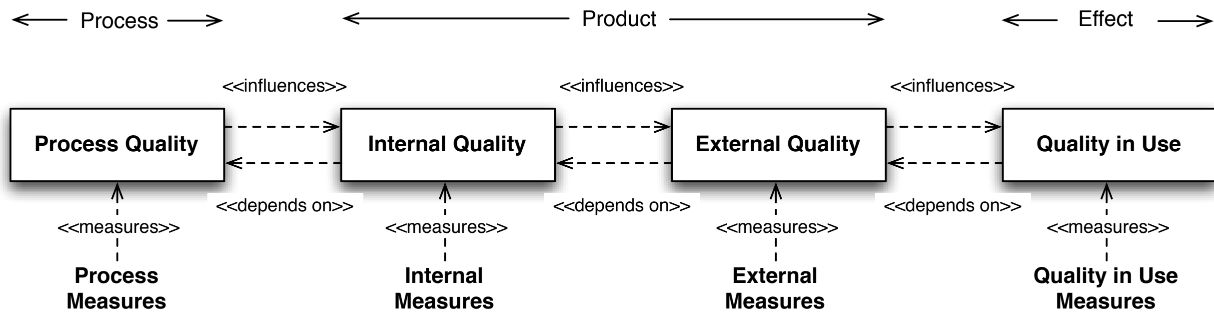

Figure 1: ISO 9126 quality views. (Adapted from ISO/IEC (2001))

"external" quality related to faults is also related to this view

- The *value-based* view equates quality to the amount a customer is willing to pay for a product
- In the *product view*, quality is tied to properties of the product being developed. This is the primary view of "internal" and "external" quality in ISO 9126 as shown in Figure 1

Turning to software architecture, there are many definitions of software architecture[2]. An influential and representative definition by Bass et al. (2003) states that:

> The software architecture of a computing system is the structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them

In other words, software architecture is concerned with structures (which can, e.g., be development or runtime structures) and abstracts away the internals of elements of structures by only considering externally visible properties.

Recently, focus has also been on decisions made when defining system structures (Tyree and Akerman, 2005; Jansen and Bosch, 2005). This leads to definitions such as:

> A software system's architecture is the set of principal design decisions made about the system (Taylor et al., 2009).

We are here concerned with a large set of open source projects and thus necessarily have to rely on (semi-)automated analyzes. Thus we take the definition of Bass et al. as our basis for a definition of software architecture.

Concerning metrics to measure software quality, a huge set of metrics to chose from exists as metrics are widely practiced and researched (Kan, 2002). However, only few of these metrics are suitable to measure the quality of software architectural.

In principle, software architecture quality can be seen in any of the views of Garvin. As an example, Grady Booch is applying a value-based view in his selection of software architecture for the Handbook of Software Architecture[3].

[2]http://www.sei.cmu.edu/architecture/start/definitions.cfm

[3]http://www.handbookofsoftwarearchitecture.com

However, prevailing software architecture analysis methods (Dobrica and Niemela, 2002) tend to take a user-based or manufacturing-based view on software architecture quality. The Architecture Trade-off Analysis Method (ATAM; Kazman et al. (2000)), e.g., aims at finding trade-offs and risks in a software architecture compared to stakeholder requirement. ATAM's focus on stakeholders gives it to a large extent a user-based quality view, but a manufacturing-based view is also included (e.g., in determining whether a specific trade-off is a potential risk). Architecture analysis methods do not often, however, include specific metrics on software architecture; rather they focus on the software architecture-specific parts of analyzes. Clements et al. (2002), e.g., describe metrics for complexity only (e.g., "Number of component clusters" and "Depth of inheritance tree" to predict modifiability and sources of faults).

Very little has been written specifically on metrics for software architecture. We looked systematically at papers from architecture-related conferences that contain metrics (Hansen et al., 2009). None of these metrics were appropriate for our purposes nor did we find any other publication that investigates empirically the effect of software architectures on product quality.

Applying statistical models and linear regression in the way we do, to investigate relationships between different metrics, is also novel in the software engineering literature, although it is common in some other disciplines.

3. Metrics

We divide the metrics that we consider into "product metrics" which are metrics related to software quality that are not architectural in nature and "architecture metrics" which are architectural in nature. Section 3.1 presents and discusses product metrics, Section 3.2 presents and discusses architecture metrics, while Section 3.3 summarizes our choice of metrics for this work.

3.1. Product metrics

We are concerned with metrics that can measure quality from any of the five views described in Section 2. With our data, we can measure quality (to some extent) from three of the views.

Metrics related to the manufacturing view

Here we can use defect count as a direct measure of quality to extent that defects are introduced during manufacturing.

Definition 1 (Open Defect Ratio (ODR)). *The Open Defect Ratio (ODR) for a project p is the ratio of the number of open defects (plus 1) to the total number of open and closed defects (plus 1).*

Metrics related to the value-based view

The value users put on an open source software project could be quantified indirectly in a number of ways: number of downloads of a project, usage count, communication about the project. Our data contains usage count, and we can use usage rate as a direct measure of quality:

Definition 2 (Rate Of Usage (ROU)). *The Rate Of Usage (ROU) of a project p is the ratio of total number of downloads to the project age (in days)*

We explicitly exclude payment since the projects we are concerned with can all be used without paying for the use.

Metrics related to the product view

In the product view, quality is not measured directly, but rather through measuring internal characteristics of the product. Basili et al. (1996) validated a set of design metrics originally proposed Chidamber and Kemerer (1994) as being useful in predicting fault-prone classes. To limit the analysis, we consider one metric here that was found to significantly predict fault proneness[4]:

Definition 3 (Weighted Methods per Class (WMC)). *The number of methods defined in a class multiplied by a weight for each method*

We do not have fault data for specific classes in our data so we do not apply this class level metric directly, but rather average WMC over all classes. Furthermore, as Basili et al. we set the weight of each method to 1:

Definition 4 (Average Methods per Class (AMC)). *The average number of methods defined in classes in a project*

Other product metrics include McCabe's cyclomatic complexity metric (McCabe, 1976) and lines of code. While there has been considerable controversy surrounding these and other metrics (cf. e.g. Shepperd (1988)), the metrics are readily calculated and may be used together to provide a metric of "complexity density" (Gill and Kemerer, 1991). The cyclomatic complexity of a program corresponds to the number of independent, linear paths through the control graph of the program.

Definition 5 (Average Complexity Density (ACD)). *ACD for a project is the sum of the cyclomatic complexities for all methods in classes in the project, divided by the total number of methods.*

[4]The validation was done using C++, not Java as in our case

3.2. Software architecture metrics

"High-level design" metrics or object-oriented design metrics can to a certain extent also be used for software architecture even though they often work on a detailed level (e.g., on specific classes and their methods and fields). As an example for such a translation of high-level design metrics to software architecture, we may, e.g. following Basili et al. (1996) again, define (analogous to WMC):

Definition 6 (Average Classes per Package (ACP)). *The Average number of Classes per Package for a project is the total number of classes divided by the total number of packages*

3.2.1. Architecture metrics based on Martin

Martin[5] defines a set of principles and metrics related to (package) architectures. One of his principles is the following:

Definition 7 (The Dependency Inversion Principle (DIP)). *Depend on abstractions. Do not depend upon concretions.*

To measure adherence to this principle, Martin proposes three metrics:

Definition 8 (INStability (INS)). *The number of outgoing dependencies (from classes) for a package divided by the sum of the number of outgoing and incoming dependencies of the package*

Definition 9 (ABStractness (ABS)). *The number of abstract classes in a package divided by the sum of the number of abstract and concrete classes in the package*

Definition 10 (NOrmalized Distance (NOD)). *The sum of instability and abstractness for a package, normalized to be in the range of 0 to 1, i.e., for a package p, NOD is* $|\mathrm{INS}(p) + \mathrm{ABS}(p) - 1|$

A value of NOD close to zero indicates that if a package has many outgoing dependencies (INS is high) then it is not abstract (ABS is low) or vice versa. Martin states that if NOD is close to zero then "the package is abstract in proportion to its outgoing dependencies and concrete in proportion to its incoming dependencies".

Assume that for a package, p, NOD(p) is close to zero. If p has a high degree of incoming dependencies in relation to outgoing dependencies (i.e., INS is close to zero), then p is highly abstract (ABS(p) close to 1). On the other hand, if p is highly concrete (ABS(p) close to 0), then p has a high degree of outgoing dependencies in relation to incoming dependencies. Thus a low NODs for a project can be said to indicate that the project follows the DIP.

In our case, we include interfaces in "abstract classes" in the ABS metric. Furthermore, for INS, we consider only dependencies expressed on a package level through "import" statements (realizing that this estimate may be slightly off).

Again, to get a project-level metric, we average the normalized distance over all packages in a project and get:

[5]`http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf`

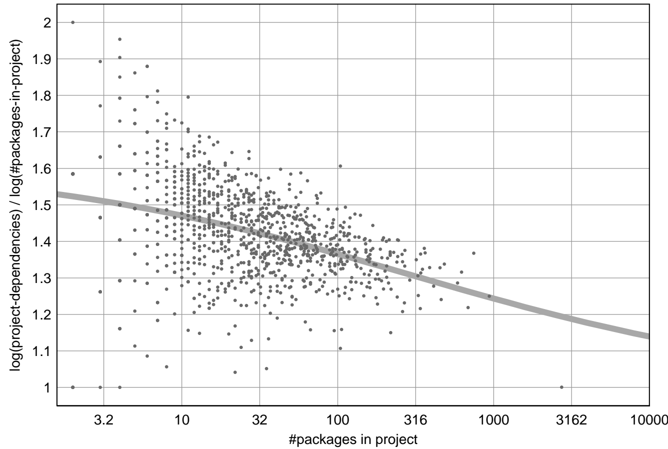

Figure 2: Scatter plot of the relationship between project size, n, and coupling exponent, k, for 1,141 studied projects. The gray line is the model (1) for the parameters estimated at the beginning of Section 5. The explained variance R^2 for the displayed model is 15.8%

.

Definition 11 (Average Normalized Distance (AND)). *AND for a project is the sum of NOD for all packages divided by the number of packages*

3.2.2. Further architecture metrics

We hypothesize that the more coupled an architecture is, the harder it is to maintain. The dependency graph of the packages of a project has a directed edge connecting two packages if a class from the first package imports the second package (or a class from that package). Thus a graph on n packages will have at least n edges and at most n^2 edges. It therefore seems natural to assume the graph having $E = n^k$ edges. We define the *coupling exponent* of a project as the exponent k, and attempt to find a model that describes how k depends on n. We note that

$$k = \frac{\log E}{\log n} \qquad \text{with } 1 \leq k \leq 2$$

and that it is realistic to assume that k tends to 1 with increasing n (if this does not hold then the average number of imports per package will grow without limit with project size). This means that k is dependent on the size of projects and not directly usable as a metric (across projects of differing sizes). This is illustrated in Figure 2 for the 1,141 projects we studied.

We are thus faced with determining a model form which offers enough flexibility to describe available data, and which gives $k \approx 1$ for large n with the added requirement that the function has a finite limit when n goes to 0. One of the simplest functions to fulfill these requirement would be a rational function on the form $1 + a/(1 + bn)$. To allow a little more flexibility, we add an exponent, c, on n together with an error term giving the model:

$$k = 1 + \frac{a}{1 + bn^c} + \frac{1}{\log n} \cdot \varepsilon \qquad (1)$$

where ε is an $N(0, \sigma^2)$-distributed error term. Notice that the error term tends to 0 with increasing n.

Maximum likelihood estimation can now be used to determine the parameters a, b, c, and σ^2; we return to this in the results section (Section 5). Based on this, we now define:

Definition 12 (Coupling Excess (CEX)). *Given a set of projects, S, where package dependencies are modeled using (2), the Coupling Excess, CEX, of a project $p \in S$ with n packages and E dependencies among these packages is the model residual:*

$$CEX(p) = \varepsilon = \log E - (1 + \frac{a}{1 + bn^c}) \log n$$

where a, b, and c have been estimated with maximum likelihood according to (1)

3.3. Choice of metrics

We summarize the product and architecture metrics that we have chosen to analyze further in Table 1 and Table 2 respectively.

4. Gathering and processing of data

Our material is projects on SourceForge. We focus on Java projects since this makes metrics calculation uniform and we hypothesize that statistical correlations are more likely to hold within similar projects. It has been observed that many projects on SourceForge have little activity (Herraiz et al., 2008; Beecher et al., 2007). In our analysis, we use projects where there is activity in terms of download and furthermore if a project has no activity it may still have a software architecture that is of interest to investigate.

Our method can be divided into three steps:

1. Gathering data on projects, which involved
   (a) Gathering meta-data on projects
   (b) Filtering projects based on meta-data
   (c) Gathering source code for projects
   (d) Filtering projects based on source code
2. Measuring filtered projects by applying selected metrics
3. Statistically analyzing measurements

We describe step 1 (“Data Gathering”) and step 2 (“Project Measurement”) next. Step 3 (analysis and results) is described in detail in Chapter 5.

4.1. Data gathering

4.1.1. Meta-data gathering

We first collected meta-data on the 21,094 most highly ranked Java projects on 2009-03-17 from SourceForge for which it was possible to get such data. Here “Java projects” were defined as projects belonging to “trove” 198 at SourceForge and “rank” was the SourceForge ranking of projects. The data consisted of characteristics such as number of bugs, time of latest file upload, number of developers, number of open bugs, and SourceForge “rank”.

Below is an example record for the most highly ranked Java project, “Sweet Home 3D” showing the characteristics that were used in our analysis.

Metric	Full Name	Explanation
ODR	Open Defect Ratio	The ratio of open defects to the total number of defects
ROU	Rate Of Usage	The number of downloads per month the project has existed
AMC	Average Methods per Class	The total number of methods divided by the total number of classes
ACD	Average Complexity Density	The sum of cyclomatic complexities for all methods divided by the number methods

Table 1: Product metrics

Metric	Full Name	Explanation
ACP	Average Classes per Package	The total number of classes divided by the total number of packages
AND	Average Normalized Distance	A measure of how abstract (ratio of abstract classes/interfaces to concrete classes) and instable (ratio of outgoing dependencies to all dependencies) packages are on average
CEX	Coupling Excess	A measure to which degree the coupling of packages to other packages exceeds our coupling model

Table 2: Architecture metrics

name	sweethome3d
url	`http://sourceforge.net/projects/sweethome3d`
bugs_closed	124
bugs_open	21
development_status	5
downloads	2441636
latest_file	2009-03-13
no_developers	8
registered	2005-11-07
repository_modules	["SweetHome3D"]
repository_type	cvs

Figures 3 to 6 show the distribution of the number of developers, development status, project age, and download characteristics for these projects

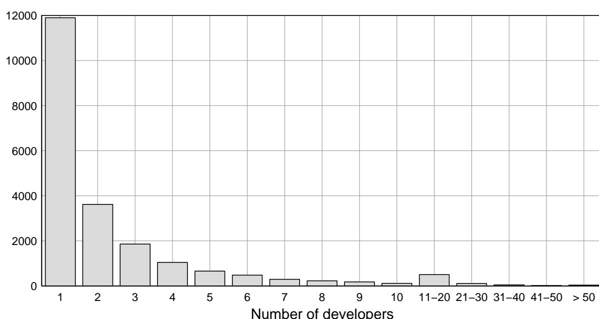

Figure 3: Number of developers per project for all projects

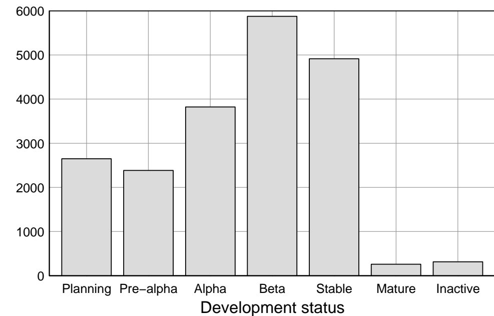

Figure 4: Development status of projects for all projects

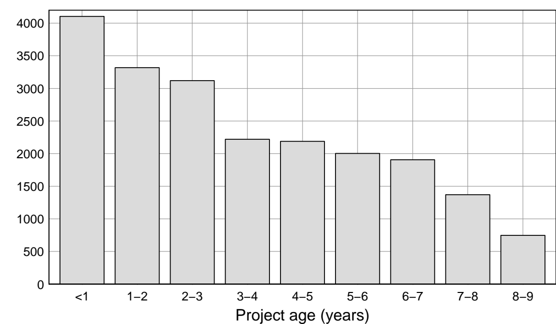

Figure 5: Project age for all projects

4.1.2. *Filtering based on meta-data*

Based on the meta-data, we defined a set of *relevant* projects, i.e., projects amenable to our analyses. To be relevant, a project had to:

- *Keep track of bugs* using SourceForge. We defined this as

	Relevance filtering ("All projects")	*Classification filtering ("Mature projects")*
Number of bugs reported	≥ 1	≥ 1
Download rate (downloads per day)	≥ 2	≥ 7
Number of developers	≥ 2	≥ 4
Development status	4, 5, 6	5, 6
Project age (days)	≥ 180	≥ 180
SLOC[7]	≥ 2000	≥ 2000
Total number of projects	*1,141*	*282*

Table 3: Filtering and classification summary

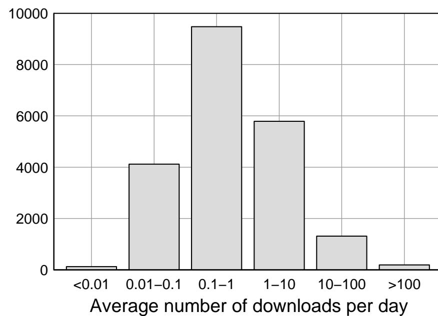

Figure 6: Download rate for all projects

`bugs_closed` + `bugs_open` being greater than zero. For Sweet Home 3D, the sum is 145 and 12,743 projects did not use SourceForge to keep track of bugs

- Have a reasonable *download rate*. We defined this to be downloaded at least 2 times a days over the project history. For Sweet Home 3D, the download rate estimated on 2009-03-17 was, e.g., approximately 2,441,636/(2009-03-17 - 2005-11-07 days) = 1,191 downloads per day. 16,159 projects did not fulfill this criterion
- Have a sufficient *number of developers* to warrant a focus on software architecture in the project. We defined this as `no_developers` being at least 2. Sweet Home 3D, e.g., had 8 developers. 11,950 projects had less than two developers
- Have sufficiently *advanced development status*. We defined this as having a SourceForge `development_status` of at least 4 which is "beta" status. The status of Sweet Home 3D, is "5" which is "stable". 10,048 projects did not fulfill this
- Have a *development history*. We defined this as `registered` being at least 180 days ago at the time our analysis was made. On 2009-03-17, Sweet Home 3D was, e.g., 1,226 days old. 2,412 projects were too young

We used the five criteria to filter the projects studied. The two first criteria are in addition not only used for filtering but also as quality measures for the projects. The result of this filtering was 1,570 Java projects.

4.1.3. Source code gathering

We attempted to download source code for the filtered projects on 2009-03-30 or revisions with date stamp 2009-03-30[6]. For projects that used CVS as configuration management tool, we downloaded all current CVS modules. For projects that used SVN as configuration management tool, we assumed that the project used the recommended "Trunk" repository layout (Collins-Sussman et al., 2009). Furthermore, we did not follow external SVN references. This means that we either i) downloaded all the top level "trunk" directory if there was one or ii) attempted to download all "dir/trunk" directories (where "dir" is a top level directory) if there was no trunk top level directory. No distinction between monolithic and plugin-based application has been made: if plugins and the core application were different projects on SourceForge, each counts as a project on its own.

After source code download, we deleted all non-Java files since that data is irrelevant for our analysis. In total, 3.3 GB of data and 550,198 Java files were downloaded.

Since our analysis requires source code, we further filtered based on the available number of lines of code. We set 2,000 SLOC[7] as the limit; research by Zhang et al. (2009) indicates that for open source Java projects, the average SLOC per class is around 100 yielding 20 classes as the limit in our case. We also removed three projects for which our metrics could not be calculated. This further reduced the number of relevant projects by 429 leaving 1,141 projects.

4.1.4. Classification filtering

We next classified a subset of the relevant projects as mature by further requiring that development status should be at least "stable", that there should be at least four developers, and there should be more than 7 downloads per day.

Table 3 summarizes our filtering of "mature" projects from all "relevant" projects.

4.2. Metrics calculation

We use four techniques to gather facts from project source code:

- We use Python and regular expressions on the contents of Java files to populate an (SQLite) database with data on public classes, packages, and "import"s. We only detect package level imports that are due to "import" statements
- We use SLOCCount[8] to calculate the physical source lines of code of projects. This data is also put into a database
- We use JavaNCSS[9] to calculate cyclomatic complexity and method count of projects. This data is exported to Rigi Standard Format (Wong, 1996)

[6]This was done through a "-D2009-03-30" argument for CVS and a "-r2009-03-30" argument for Subversion

[7]SLOC: physical source lines of code, which is the total number of non-blank, non-comment lines in the code

[8]`http://www.dwheeler.com/sloccount/`

[9]`http://javancss.codehaus.org/`

- Finally, we use a Java parser (built upon the Java grammar included in JavaCC[10]) to extract data on inheritance (and implementation), on classes (and interfaces), and on methods. We do a simple semantic analyzes that only uses the current project as classpath to qualify references. Furthermore, we do not take enums or generics into account

Thus, effectively, we have two types of data sets: i) relational data and ii) Rigi data. We initially worked with relational data, but found out (in line with Beyer et al. (2005)) that relational queries were inefficient in handling our data and thus also worked with data in Rigi format.

With the relational data, we use simple relational queries, e.g., to calculate the number of dependencies between distinct packages in a project. With the Rigi data, we use Crocopat (Beyer et al., 2005) to, e.g., calculate ABS, INS, and NOD. The end result is in both cases metrics and numbers that can be used directly in our statistical analyzes.

5. Results

We now turn to our analysis of the gathered metrics data. We first estimate the parameters of our coupling model (Section 5.1), then in Section 5.3, and 5.4 we construct and discuss several linear regression models involving the metrics. Finally in Section 5.5, we discuss some limitations of our analysis.

5.1. Modeling of coupling

Using (1) with the data for all the 1,141 projects used in the study, maximum likelihood estimation gives $a = 0.614$, $b = 0.136$, c=0.804 and $\sigma^2 = 0.0185$ giving the model

$$k = 1 + \frac{0.614}{1 + 0.136 n^{0.804}} + \frac{1}{\log n} \cdot \varepsilon \quad (2)$$

where ε is $N(0, \sigma^2 = 0.0185)$.

We have already shown this model in Figure 2 in Section 3.2.2. It is also instructive to see directly how $\log E$ depends on $\log n$. This relationship is depicted in Figure 7.

5.2. Variable transformation

Turning to the distribution of calculated metrics, Figure 8 shows histograms of of the product and architecture measures. The raw values of four of the seven metrics have highly positively skewed distributions. For two of these, AMC and ACP, it sufficed to take logarithms to produce approximately normal distributions (meaning that AMC and ACP are approximately log-normally distributed), but for ACD and ROU the distribution was still quite skewed even after taking logarithms. For these we removed the skew using a Box-Cox power-transformation (Box and Cox, 1964). The transformation is

$$y = \frac{(x + \alpha)^\lambda}{\lambda}$$

where x is the raw variable and y is the transformed variable. The parameters α and λ were estimated by maximizing the normal likelihood over all 1,141 projects, giving $\alpha = -0.038$, $\lambda = -1.94$ for ROU and $\alpha = -2.34$, $\lambda = 2.52$ for ACD. Finally, we decided not to try to normalize ODR. We note that it is a metric that depends highly on each project's culture of defect reporting.

[10] `https://javacc.dev.java.net/`

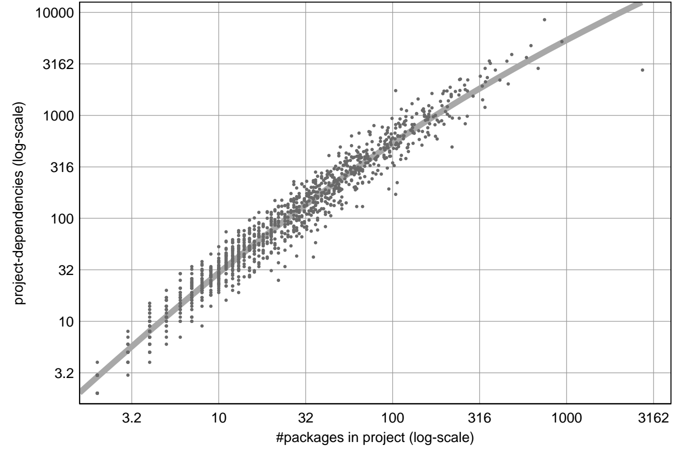

Figure 7: Scatter plot of the relationship between number of packages in project, n, and number of edges, E, in the package dependency graph for 1,141 studied projects. The gray line is the model (2) times $\log n$

The histograms in Figure 8 indicate that after transformation all the variables are essentially skew-free, and all except ODR are approximately normally distributed. To investigate normality further, the number of projects out of the total of 1,141 with a variable that is more extreme than 2 and 3 standard deviations from the mean have been counted. These counts are shown in Table 4 together with the counts that a pure normal distribution would give.

	$< -3\sigma$	$< -2\sigma$	$> 2\sigma$	$> 3\sigma$
Normal	*1.5*	*26*	*26*	*1.5*
box-cox(ROU)	0	32	30	4
log(AMC)	5	19	32	8
box-cox(ACD)	4	23	26	5
log(ACP)	0	12	35	11
AND	0	48	27	2
CEX	10	30	15	1

Table 4: Counts of Extreme Values of Metrics, All Projects

Note that ACP has a little heavy right tail, and CEX has a heavy left tail, but apart from that the normality assumption holds reasonably well. This indicates (approximately) that AND and CEX are normally distributed, that AMC and ACP are log-normally distributed, and that ROU and ACD have a truncated power-normal distribution.

5.3. Pairwise regression and model significance

For each of the 12 pairs of architecture and product metrics we have investigated both a straight line and a parabolic linear regression model taking the architecture metric as an independent variable. a 5% significance level, all pairs gave a model significantly different from a constant model, according to a standard t-test. Figure 9 shows a scatter plot of the metric pairs together with the models. The model coefficients and p-values for each model are also displayed in the figure. In cases where the second order term was not significantly different from 0, a straight line model is given.

From the top left graph of Figure 9 we observe that projects with ACP around 15–20 (minimum is at 14.7 for all projects, and at 18.9 for most mature ones) have significantly fewer open defects than the projects where ACP is either low or high. In addition we see that the

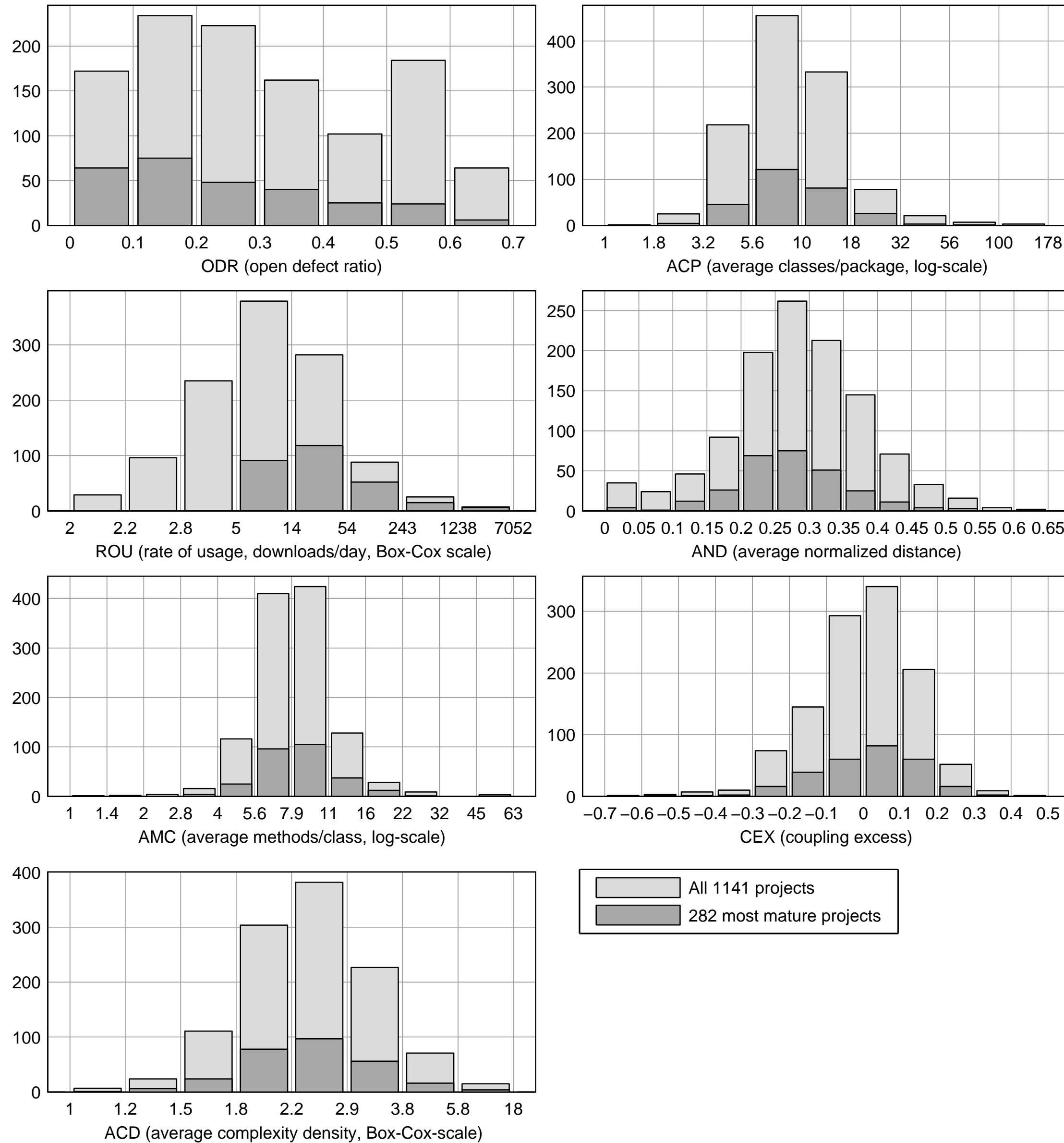

Figure 8: Distribution of measurements for all and mature projects

most mature projects have a lower defect ratio. The defect ratio also depends on AND in a similar way, for AND around 0.3 (minimum at 0.27 for all, 0.33 for mature) the defect ratio is on average significantly lower than for low and high AND values. It is in particular interesting to see that low AND values seem to give more defects, in view of the principle put forward in Martin, that good architecture should have normalized distance close to zero. Regarding the pair CEX-ODR we see that for all projects the relationship is weak, but for the most mature projects, it seems again to be advantageous to have close to average coupling, rather than low or high (minimum is at 0.08 for mature).

Turning attention to ROU, we observe that the most downloaded projects among the mature ones have a tendency to have high ACP. The effect of AND on ROU matches its effect on ODR: For average AND there are significantly more downloads than when AND is low or high (the maxima occur at at 0.27 for all, 0.24 for mature projects). The CEX-ROU graph indicates that projects with low coupling tend to have fewer downloads. For the ACP and CEX relationships one might have expected the opposite effect, i.e. that fewer classes per package and low coupling might be beneficial.

The remaining two dependent variables are the internal product metrics AMC and ACD. For high quality their values should be low. For mature projects the effect of ACP on these metrics matches the effect of ACP on defect ratio: average ACP is beneficial (minima at 6.7 and 7.7 respectively). For all projects the effect is less pronounced. Regarding the effect of AND and CEX on ACD, we again find the effect to be as expected, and similar to the effect of these metrics on ODR (minima at 0.35 and −0.04 for all projects; 0.42 and 0.01 for mature ones). The dependence of AMC on AND appears minimal, but its dependence on CEX is however as one might have expected: both low AND and low CEX should be favorable.

5.4. Multiple regression models

In addition to the models shown in Figure 9, we have constructed multiple regression models, explaining architecture metrics in terms of

Dependent variable	Project set	Model variables, model coefficients, and p-values										R^2
		Constant	CEX	CEX^2	p	log(ACP)	$\log(ACP)^2$	p	AND	AND^2	p	
ODR	All	0.57	-0.03	-0.15	0.61	-0.27	0.11	8×10^{-3}	-1.02	1.87	3×10^{-8}	4.3%
	Mature	0.57	-0.02	0.51	0.05	-0.30	0.12	0.16	-1.12	1.72	0.05	7.4%
box-cox(ROU)	All	1.18	0.83		0.02	0.11		0.59	4.47	-8.44	0.01	1.5%
	Mature	2.45	1.20		0.02	0.72		0.02	0.40	-2.16	0.54	5.7%
log(AMC)	All	0.88	0.15		3×10^{-5}	0.08	-0.02	0.04	-0.14		4×10^{-3}	3.3%
	Mature	1.22	0.36		1×10^{-7}	-0.60	0.32	9×10^{-3}	-0.17		0.13	12.9%
box-cox(ACD)	All	0.39				0.001	0.000	0.82	-0.01	0.01	2×10^{-7}	2.8%
	Mature	0.40				-0.005	0.003	0.04	-0.01	0.01	0.02	5.6%

Table 5: Multivariate regression models, explaining architecture metrics in terms of product metrics. Blanks in the table indicate that the corresponding term was non-significant for both project sets. When both a linear term and a squared term are present, the p-value is joint for both terms. It measures whether the addition of these two terms improves the model significantly.

product metrics, using step-wise regression (see e.g. Lindgren (1976)).

The resulting models are shown in Table 5. As an example, the the complete ODR model for all projects is

$$\begin{aligned} ODR = 0.57 - 0.03 \cdot CEX - 0.15 \cdot CEX^2 \\ - 0.27 \cdot \log(ACP) + 0.11 \cdot \log(ACP)^2 \\ - 1.02 \cdot AND + 1.87 \cdot AND^2 \end{aligned}$$

Blanks in the table indicate that the corresponding term turned out to be non-significant for both project sets. When both a linear term and a squared term are present, the p-value is joint for both terms. It measures whether the addition of these two terms improves the model significantly.

To facilitate the comparison between projects sets, a term is included if it is significant for either projects set. Another convention that is followed is to include a linear term whenever the squared term is included (regardless of whether the linear term is significant).

The column headed R^2 shows the proportion of the total variance of the dependent variable which is explained by the models.

It is interesting to note that except for the ACD-model, all the architecture metrics are significant components of the models. This means that their combined effect on the corresponding product metric is larger than their effect in the simple regression models shown in Figure 9. However it must be admitted that the R^2-values are not very impressive. Even though the p-values are highly significant (since many projects have been analyzed), there is a large amount of spread in our data. The models can be used to predict average product metrics with confidence, but they would not be very useful to predict metrics for single projects.

5.5. Limitations

There are a number of important limitations to our study that may influence its validity. Threats to validity are sometimes categorized as construct, internal and external (Carver et al., 2004). Concerning construct validity,

- Our analysis across projects is based on average values. Concas et al. (2007) argue that system properties (such as WMC) often follow a power law or a log-normal distribution. Thus it may be problematic to work with the mean (or standard deviation) of these properties to characterize whole systems or projects. While working with means (or standard deviations) may thus be representative of a known distribution, we did not assume specific distributions. Further research could look also at specific distributions of these metrics for the projects investigated.
- The analysis is automated. This means that we did not check, e.g., if the downloaded source code could compile or if bug reporting was consistent across projects. The large set of projects is meant to counter the effects of this. A related problem is the step to filter the gathered data: this is partially based on information imposed by the project developers such as number of developers or development status. In particular the latter is highly subjective (and typically just monotonically ascending). A more complete analysis of the project status, could be performed by mining the FLOSSMole database. (Howison et al., 2006).
- We have analyzed a limited number of metrics. In particular, the range of available architecture metrics appears limited and further research would be needed in that area. In relation to this, there is a current interest in software architecture research in non-product aspects of software architecture design, e.g., in design decisions (Jansen and Bosch, 2005) and organizations (Clements et al., 2007)
- The CEX metric is dependent on the set of changeable parameters, a, b, and c that are estimated based on a particular set of projects. It would be worthwhile to try to simplify the coupling model to obtain some sort of measure that would be reasonably size-independent, but simpler than the current CEX definition, and hopefully more likely to apply to other project sets
- A possible problem of the ODR metrics is that failure reporting is not uniform across projects which implies that the data about defects may not be accurate or timely. This is, however, an important metric of quality and given that we analyze a large number of projects, the inaccuracies may even out. Furthermore, we exclude project that do not report defects using SourceForge in our analysis

Internal validity deals with the question of whether cause and effect relationships discovered in a study are real.

- One reason for failure to meet internal validity is that statistical tests may fail. Either one finds relationships that are really nonexistent (type I error) or real relationships are wrongly deemed non-significant (type II error). With the standard statistical tests and 5% significance level we use, the probability of making a type I error is 5%, however, the probability of type II errors is difficult to estimate.

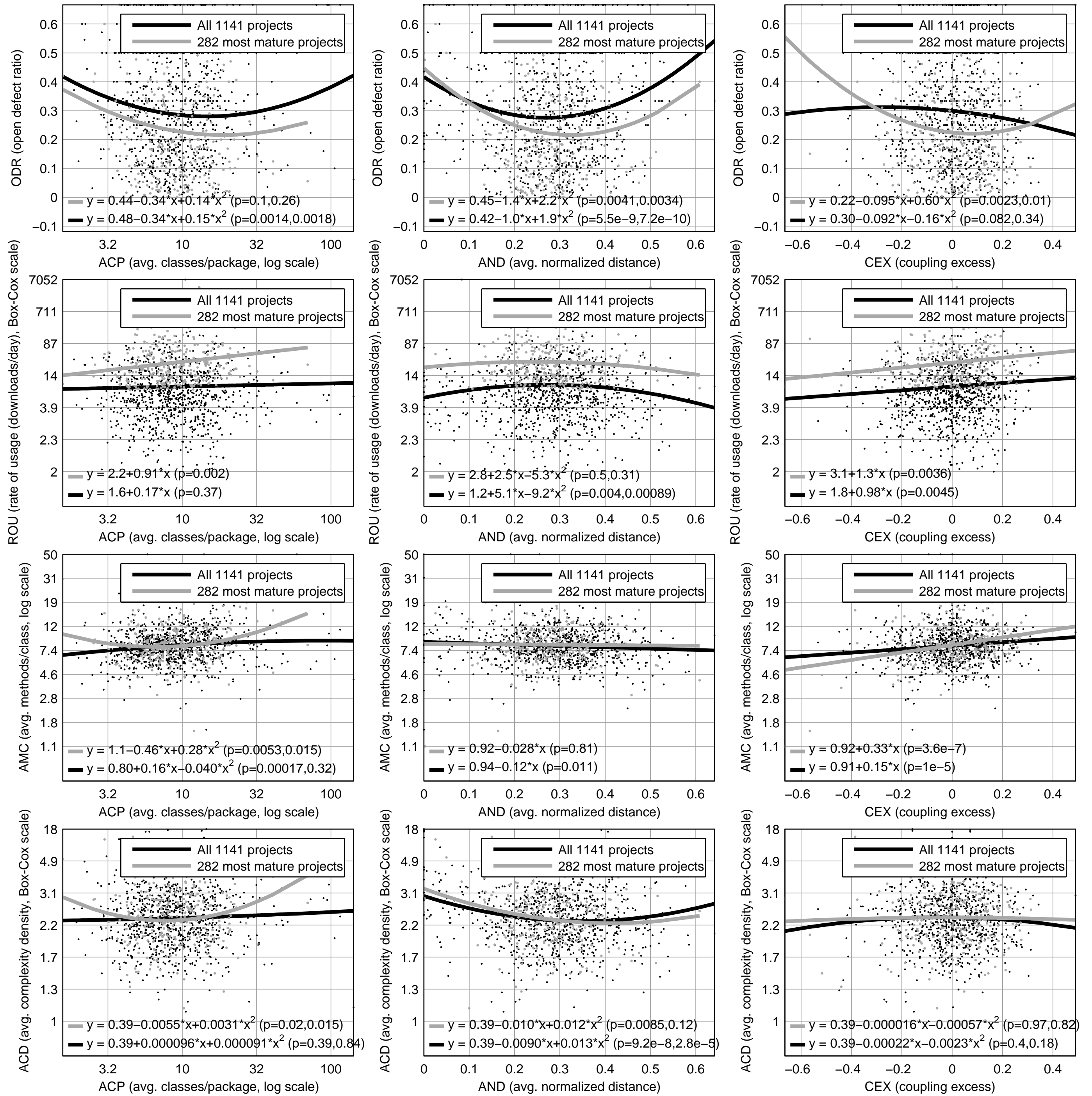

Figure 9: Analysis of relationships between product and architecture metrics (When two *p*-values are given the first applies to the x coefficient and the second applies to the x^2 coefficient)

- Maybe the most severe limitation of our study is that we are attempting to explain software quality by software architecture, but have only limited description of the architecture. We are forced to make do with metrics that describe the architecture at best partially. Our results, that several of the software quality measures are significantly related with the architecture metrics, may well be real cause-and-effect relationships, but it is quite possible that, in some cases, both the quality and the architecture metric are the result of some unknown cause. There is not really an easy way around this limitation.
- The CEX metric is dependent on the set of changeable parameters, a, b, and c that are estimated based on a particular set of projects. It would be worthwhile to try to simplify the coupling model to obtain some sort of measure that would be reasonably size-independent, but simpler than the current CEX definition, and hopefully more likely to apply to other project sets

Finally regarding external validity:

- The projects that we surveyed are all open source projects and moreover open source projects that are hosted on SourceForge. While the results may not be generalizable to closed source projects, this points to an area of further research

6. Conclusions

We carried out a review of some metrics on software design and quality that have been proposed in the scientific literature, especially as applied to (large) Java projects. We classified these metrics into architecture metrics, which try to measure the high-level design of software, and product metrics, which try to measure software implementation. Seven metrics were computed for a large body of open source Java projects, and subsequently analyzed statistically. To our knowledge this is the first study of this type.

An important issue in software architecture is that of package coupling, i.e. the degree to which the packages of a project depend on one another. We hypothesize that the dependency graph becomes sparser and sparser with project size. We have modeled the effect as $E = n^k$, where n is the number of packages and E is the number of edges in the graph, and find that for small projects k is around 1.5 and for the largest projects that we analyzed it is around 1.25. Our model then assumes that k tends to 1 with increasing n. One of our architecture metrics, CEX, is based on this model, but the others (classes per package, ACP, and normalized distance, AND) are based on previously proposed metrics. As product metrics we computed open defect ratio, ODR, rate of usage, ROU, methods per class, AMC, and cyclomatic complexity, ACD, but all of these are (or have been proposed to be) measures of software quality. For six of these metrics (all but ODR) we established an approximate probability distribution, valid for our data set.

The analyzed projects consist of 1,141 open source software projects selected from the SourceForge repository. Criteria for inclusion in the study included that the projects were pure Java projects and not brand new, used SourceForge to keep track of bugs, had at least 2000 source lines of code, had at least two developers, had been downloaded at least twice daily on average, and had reached development status beta. In addition we selected a subset of 282 "mature" projects, which had at least four developers, had been downloaded at least seven times daily, and had reached development status stable.

For both sets of projects (i.e., all 1,141, and the 282 mature ones) we constructed regression models for all 12 pairs of product-architecture metrics as well as multiple regression models for all three architecture metrics. In all cases statistically significant relationships were discovered. The relationships are in general stronger for the mature set. For this set and ODR all three architecture metrics give rise to convex parabolic relationships, meaning that when these metrics give medium values, less error prone software results than when the metric values are extreme, whether low or high. ODR as predicted by the models ranges from a minimum of around 0.2 to a maximum of around 0.4. The relationship is similar for both ACP and AND in the larger project set.

There is also a significant relationship between the architecture metrics and the other product metrics ROU, AMC and ACD. For the mature set, medium values of ACP go together with low values of AMC and ACD (both pointing to high quality), and for both project sets medium values of AND give high ROU and low ACD (again pointing to high quality). In other cases the effect is not as conclusive, and in a few cases it is even counterintuitive (in particular for the pairs CEX-ROU and CEX-AMC).

In general, the effect of the architecture metrics on the product metrics agrees with what has been proposed in the literature. The most notable exception is AND. It was formulated as ideally being 0, but our results indicate that it is better to strive for a "compromise" on average when designing architecture, e.g., a value around 0.3. A similar tentative conclusion can be reached for ACP: to produce quality software one should aim for about 10 classes per package on average. The effect of CEX on quality is more inconclusive.

In summary, we have presented evidence of an effect of architecture quality on product quality in a set of 1,141 open source Java projects. Further research is needed to address the limiting threats to validity and to be able to make predictions on a per-project basis, but the effect we have found is quite significant statistically, and may be relied on to draw conclusions about expected software quality given a set of projects.

References

Alexander, C., 1979. The timeless way of building. Oxford University Press, USA.

Basili, V., Briand, L., Melo, W., Oct 1996. A validation of object-oriented design metrics as quality indicators. Software Engineering, IEEE Transactions on 22 (10), 751–761.

Bass, L., Clements, P., Kazman, R., 2003. Software architecture in practice, 2nd Edition. Addison-Wesley Professional.

Beecher, K., Boldyreff, C., Capiluppi, A., Rank, S., 2007. Evolutionary success of open source software: an investigation into exogenous drivers. In: Proceedings of the Third International ERCIM Symposium on Software Evolution (Software Evolution 2007). pp. 124–136.
URL ftp://ftp.umh.ac.be/pub/ftp_infofs/2007/ERCIM-Evol2007.pdf

Beyer, D., Noack, A., Lewerentz, C., 2005. Efficient relational calculation for software analysis. IEEE Transactions on Software Engineering 31 (2), 137–149.

Box, G., Cox, D., 1964. An analysis of transformations. Journal of the Royal Statistical Society. Series B 26, 211–252.

Carver, J., VanVoorhis, J., Basili, V., August 2004. Understanding the impact of assumptions on experimental validity. In: Empirical Software Engineering, 2004. ISESE '04. Proceedings. 2004 International Symposium on. pp. 251 – 260.

Chidamber, S., Kemerer, C., Jun 1994. A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20 (6), 476–493.

Clements, P., Kazman, R., Klein, M., 2002. Evaluating software architectures: methods and case studies. Addison-Wesley Professional.

Clements, P., Kazman, R., Klein, M., Devesh, D., Reddy, S., Verma, P., Jan. 2007. The duties, skills, and knowledge of software architects. In: Proceedings of The Sixth Working International IEEE/IFIP Conference on Software Architecture (WICSA 2007). pp. 20–23.

CMMI Product Team, 2006. CMMI for Development, Version 1.2. Tech. Rep. CMU/SEI-2006-TR-008, Software Engineering Institute, Carnegie Mellon University.

Collins-Sussman, B., Fitzpatrick, B. W., Pilato, C. M., 2009. Version Control with Subversion. For Subversion 1.4. Red-Bean online version. Compiled from r2866.

Concas, G., Marchesi, M., Pinna, S., Serra, N., 2007. Power-laws in a large object-oriented software system. IEEE Transactions on Software Engineering 33 (10), 687–708.

Dobrica, L., Niemela, E., 2002. A survey on software architecture analysis methods. IEEE Transactions on software Engineering 28 (7), 638–653.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design patterns: elements of reusable object-oriented software. Addison-wesley Reading, MA.

Garlan, D., Shaw, M., 1993. An introduction to software architecture. Advances in Software Engineering and Knowledge Engineering 1, 1–40.

Garvin, D. A., 1984. What does "product quality" really mean? Sloan Management Review 26 (1), 25–43.

Gill, G., Kemerer, C., Dec 1991. Cyclomatic complexity density and software maintenance productivity. Software Engineering, IEEE Transactions on 17 (12), 1284–1288.

Hansen, K. M., Jónasson, K., Neukirchen, H., July 2009. An empirical study of open source software architectures' effect on product quality. Tech. Rep.

VHI-01-2009, Engineering Research Institute, University of Iceland, http://www.hi.is/~kmh/doc/vhi-01-2009.pdf.
Herraiz, I., González-Barahona, J. M., Robles, G., 2008. Determinism and evolution. In: Hassan, A. E., Lanza, M., Godfrey, M. W. (Eds.), Fifth International Workshop on Mining Software Repositories, MSR 2008 (ICSE Workshop), Leipzig, Germany, May 10-11, 2008, Proceedings. ACM, pp. 1–10.
Howison, J., Conklin, M., Crowston, K., 2006. FLOSSmole: A collaborative repository for FLOSS research data and analyses. International Journal of Information Technology and Web Engineering 1 (3), 17–26.
ISO/IEC, 2001. Software engineering – Product quality – Part 1: Quality model. ISO/IEC 9126-1:2001.
ISO/IEC, 2004. Information technology – Process assessment – Part 1: Concepts and vocabulary. ISO/IEC 15504-1:2004.
Jansen, A., Bosch, J., 2005. Software architecture as a set of architectural design decisions. In: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005). pp. 109–120.
Kan, S., 2002. Metrics and models in software quality engineering. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.
Kazman, R., Klein, M., Clements, P., August 2000. Atam: Method for architecture evaluation. Tech. rep., CMU/SEI.
URL http://www.sei.cmu.edu/publications/documents/00.reports/00tr004.html
Kitchenham, B., Pfleeger, S. L., Jan. 1996. Software quality: The elusive target. IEEE Software, 12–21.
Lindgren, B. W., 1976. Statistcal Theory, 3rd Edition. McMillan, New York.
McCabe, T. J., 1976. A Complexity Measure. IEEE Transactions on Software Engineering 2 (4), 308–320.
Perry, D., Wolf, A., 1992. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17 (4), 40.
Shepperd, M., Mar 1988. A critique of cyclomatic complexity as a software metric. Software Engineering Journal 3 (2), 30–36.
Taylor, R. N., Medvidovic, N., Dashofy, E. M., 2009. Software Architecture: Foundations, Theory, and Practice. Wiley.
Tyree, J., Akerman, A., 2005. Architecture decisions: Demystifying architecture. IEEE software, 19–27.
Wong, K., July 1996. Rigi User's Manual. Department of Computer Science, University of Victoria, http://www.rigi.cs.uvic.ca/downloads/rigi/doc/user.html.
Zhang, H., Tan, H. B. K., Marchesi, M., 2009. The distribution of program sizes and its implications: An eclipse case study. CoRR abs/0905.2288.

Klaus Marius Hansen is a professor of Software Development at the University of Copenhagen. He received a Ph.D. degree in Computer Science from Aarhus University in 2002 and focuses on software architecture research in particular in relation to pervasive and dependable computing.

Kristjan Jonasson is a professor and head of department at the Department of Computer Science of the University of Iceland. He received a Ph.D. degree in Numerical Analysis from the University of Dundee in Scotland in 1985. His research interests are in scientific computing, numerical optimization and applied statistics.

Helmut Neukirchen is associate professor for Computer Science at the University of Iceland. He received a Ph.D. degree in Computer Science from the University of Göttingen in 2004 and a Computer Science diploma degree from the RWTH Aachen University in 1999. His research interest are in the domains of software quality, distributed systems, and agile software development.