Software metrics for policy-driven software development life cycle automation

Leonid Borodaev
Faculty of mathematics and natural sciences, University of Groningen
Groningen, The Netherlands
Leo.borodaev@gmail.com

Rein Smedinga, Alex Telea
Faculty of mathematics and natural sciences, University of Groningen
Rix Groenboom,
Parasoft,
Groningen, The Netherlands

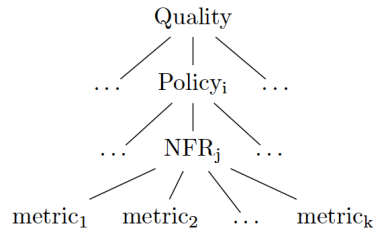
***Abstract*—Automation of SDLC requires continuous verification of compliance of the software product under construction to a set of expectations about its quality. We define a policy as an expectation about some aspects of software quality that is expressed as a collection of non-functional requirements (NFRs), compliance to which can be potentially measured. The results of such measurements can be used to verify whether the product meets the expectation set about it. In this paper, we discuss existing NFR taxonomies and propose mapping of software metrics to twelve NFRs. We then propose a model for reliability prediction using publicly available quality metrics for several open source projects.**

Keywords—NFR, software quality, continuous delivery, policy

I. Introduction

The key idea behind policy-driven development is that high-level expectations about software quality can be represented by a set of non-functional requirements (NFRs) in the early phases of the development process, and then be continuously verified during the product development [1]. Verifying in an automated fashion that the product adheres to the policy typically implies performing a measurement procedure and setting a threshold on the resulting measured values (see Figure 1). Several problems arise in this context, as follows: all stakeholders must agree on the way (language) to express the high-level expectations; NFRs that represent the policy need to be measurable; and suitable thresholds need to be set for the results of such measurements, so that an automated decision about the software can be made.

To facilitate the maturing of policy driven development, we propose a framework for mapping NFRs to software metrics and demonstrate how this mapping can be used to ensure that the software meets expectations about it. For this, we use well-accepted quality metrics from software engineering, whose implementation is publicly available [10]. We proceed as follows. We begin by overviewing related work on software quality attributes (Sec. II) and related software quality metrics (Sec. III). In section IV, we propose mappings for the following quality attributes: security, reliability, maturity, maintainability, modularity, reusability, analysability, modifiability, testability, portability, resilience, documentation. In section V, we demonstrate one way to derive measurable thresholds to quantify one high-level NFR (reliability) by using a machine learning approach. Finally, Sec. VI concludes the paper.



$$Q = f(P_1, P_2, \ldots P_n) = g(NFR_1, NFR_2, \ldots NFR_m)$$

$$NFR_k = h(m_1, m_2, \ldots m_i)$$

Figure 1: relation of quality, policies, NFRs, and software metrics.

II. Software Quality Attributes

In general, NFRs are defined as being requirements that describe (a) quality aspects of the source code or (b) pertain to the run-time characteristics of the program. In the following, we consider only NFRs related to software quality attributes (b), often referred to as quality NFRs. Extensive listings of NFRs and software quality attributes can be found in [2][3][4]. NFRs can be organized via taxonomies, which are typically either flat or hierarchical. Hierarchical taxonomies imply that higher-level NFRs can be expressed via lower-level NFRs. While useful in understanding what NFRs mean and how they are related to each other, such taxonomies do not provide clarification on whether their NFRs are measurable or not and, for measurable NFRs, how these can be estimated in practice.

Most software quality concepts are, by nature, rooted in the characteristics of source code only, see e.g. modularity, complexity, documentation, and all other metrics described in e.g. [11]. Other quality concepts combine aspects of the source code with those of the development (software evolution) process and run-time program behaviour [13][14]. Such quality concepts are quantified in practice by software metrics, described next.

III. SOFTWARE METRICS

In practice, hundreds of different software metrics are used to quantify the software quality aspects introduced in Sec. II. We next outline several widely used software metrics, from which we will select the ones we will use next in our work for quantifying NFRs.

A. Direct metrics

- Weighted methods per class (WMC)[12]: The sum of complexities of a class' methods.
- Depth of inheritance tree (DIT)[12]: The length of the longest path-to-root in the inheritance tree of a system.
- Number of children (NOC)[12]: The number of immediate children (subclasses) of a class in a class hierarchy.
- Coupling between objects (CBO)[12]: The number of couplings (dependencies) between two classes.
- Response for class (RFC)[12]: The number of methods that can potentially be executed in response to a message received by an object of a given class.
- Lack of cohesion in methods (LCOM)[12]: Two methods of a class are related if they use at least one common field. Let Q and P be the number of pairs of methods in class C that are, respectively are not, related. Then, LCOM is defined as $\max(P - Q, 0)$.
- Static analysis violations (S): The number of violations of chosen rules by a chosen static analysis engine. Rules are typically defined as patterns searched over the annotated syntax tree (AST) of a program [15][16].
- Reported defects (D): The number of defects reported through a bug tracking system (BTS) or mailing list.
- Number of classes (NC): The number of (abstract) classes comprising an object-oriented system.
- Number of parameters (NP): The number of parameters that a method takes when called.
- Cyclomatic complexity (CC): The number of linearly independent paths through the control flow graph of a program.
- Lines of code (LOC): The size of the program in lines of code text. If not mentioned otherwise, LOC is usually assumed to exclude comment lines.
- Number of packages (NOP): The number of packages comprising a system.
- Afferent couplings (Ca): For a package P, Ca is the number of classes in other packages that use classes in P [6].

  Efferent Coupling (Ce): For a package P, Ce is the number of classes in other packages that classes in P use [6].
- Number of developers (DEV): The number of developers involved with the process of creating and maintaining the software [13][17].
- Test coverage (C): The number of lines covered by a test suite when executed.
- Failed test ratio (F): The number of failed tests, in an unit or similar test, to the total number of executed tests.
- Languages (L): The number of different programming languages present in the analysed source code.

B. Indirect or derived metrics

- Comment density (CD):. The ratio, or percentage, of comment lines to the total number of lines in source code.
- Defect density (DD): The ratio of reported defects through a BTS (D) to the number of non-comment lines (LOC).
- Defect arrival rate (DAR): The number of defects reported through a BTS (D) per unit time.
- Abstractness (A): The ratio of the number of abstract classes (and interfaces) in a package to the total number of classes in the package [6].
- Instability (I): The ratio of efferent couplings to total couplings, i.e., $I = Ce/(Ce+Ca)$ [6].
- (S/DEV): The ratio of static analysis violations to the number of developers. Indicative of developer experience.
- (D/DEV): The ratio of reported defects to the number of developers.
- (NOM/DEV): The ratio of number of methods to the number of developers. High values can indicate low maintainability of the source code.
- (LOC/DEV) indicates the amount of code a developer is responsible for, in LOC.
- (NC/DEV): The amount of classes a developer is responsible for.
- (CC/LOC): The density of decision making in source code. For a detailed discussion of this and the following three metrics, see [11]
- (LOC/NOM): Average method size, in LOC [11].
- (NOM/NC): Average size of class, in methods [11].
- (NC/NOP): Average package size, in classes [11].
- (S/KLOC): Density of static analysis violation occurrences.

C. Metric extraction tools

The above-mentioned metrics can be computed by many metric extraction tools that offer various trade-offs between

number of covered metrics, scalability, genericity, and ease of use [15][16][18][19]. In our work, we use for this task the SonarSource toolsuite [10] to demonstrate the feasibility of quantifying the proposed mappings. When not provided, we compute indirect metrics from their direct counterparts following Sec. IIIB.

IV. Proposed mapping

In this section, we propose a mapping of twelve high-level software quality attributes to software metrics. Figure 2 shows the proposed mapping of quality attributes (rows) to metrics (columns), which we discuss in detail next.

	Class						Package				Other										
	WMC	DIT	LCOM	NOC	CBO	RFC	Ca	Ce	A[1]	I[2]	NC[3]	LOC	CC[4]	DD[5]	DAR[6]	CD[7]	NOM[8]	S[9]	NP[10]	L[11]
Security		*	*		*	*	*	*									*	*	*	*
Reliability												*	*	*	*		*	*		*
Maturity														*	*					
Maintainability			*		*	*	*	*		*	*					*				*
Modularity					*	*	*	*		*										
Reusability					*	*		*		*			*	*					*	*
Analysability		*	*		*	*		*	*		*					*	*		*	*
Modifiability			*	*	*	*	*			*										
Testability					*				*				*				*		*	
Portability											*	*								*
Documentation																*				
Resilience					*	*	*	*						*	*		*	*	*	*

Figure 2: Mapping of quality attributes (rows) to metrics (columns).

A. Security

Security can be defined as the level of data protection [3]. It is apparent that there is no way to measure security directly. We identify the following factors that can be used in predicting security: the complexity of method call hierarchy and the length of data flow paths through the program. The longer the data path, the more chance there is that either unwanted data can be injected by an attacker, or that protected data will leak. We map these to the following metrics.

A large DIT negatively affects the security of a class, because the longer the inheritance path, the more convoluted become connections between overridden and inherited methods, and the easier it is to make a mistake.

LCOM negatively affects the security of an application, due to the greater exposure of fields to methods.

CBO negatively affects security due to the greater number of classes involved in information exchange.

Higher RFC undermines security by increasing the length of the data path. The same holds for Ca and Ce, but with respect to the number of dependencies.

Higher values of NOM and NP increase the attack scope by increasing the number of parameter passes and, thus, the amount of possibly tainted data.

Finally, S is indicative of the experience of developer who has written the analyzed unit. In turn, low-experience developers are more prone to generate low-security code.

B. Reliability

We define reliability as the degree of confidence that the software will work in an expected manner. We argue that reliability depends on the number of reported defects, their density, and their arrival rate. The number of found static analysis errors (S) is an important predictor for reliability, since a high number of discovered static errors increases the probability of software failure.

Cyclomatic complexity (CC) is a second predictor for reliability, since it is easier to introduce an error into complex code. The same reasoning applies to the number of methods (NOM).

C. Maturity

We define maturity as the degree to which a software product has grown to meet its expected behaviour. As software matures, defects present in its early versions are removed, while new defects can be injected. Thus, maturity is related to the defect density (DD) and defect arrival rate (DAR).

D. Maintainability

Maintainability can be defined as the ease with which the program can be extended, updated, modified, and its errors removed [3][2]. Hence, LCOM, CBO, RFC are important metrics for quantifying maintainability since they correlate with the complexity of a system, and it is well known that more complex systems are more difficult to maintain. The same reasoning applies to Ca, Ce and I metrics, but at the package level. Conversely, NC negatively correlates with maintainability. Finally, CD determines understandability and, thus, maintainability of the system.

E. Modularity

We understand by modularity a measure of the level of independence of the constituents (components) of a system. A component in a highly modular system has little to no impact on, and from, other components of the system. Modularity is decreasing with higher CBO and RFC, due to a larger number of objects involved in message exchange. Ca and Ce negatively affect modularity on the package level, for the same reason. Packages with higher instability I are less modular because they depend on more classes in other packages.

F. Reusability

We define reusability as the degree to which a program (or parts thereof) can be reused in other applications [2]. Reusability of a class depends on its CBO and RFC, since they represent the number of dependencies on other classes. Reusability of a package is reflected in Ce. Separately, reusability of a system is lower if it contains a lot of defects. Therefore, DD is important for quantifying reusability. Finally, more complex code (higher CC values) is less reusable due to lower understandability. Methods with high NP are harder to reuse, for the same understandability reason.

G. Analysability

The notion of analysability lies close to understandability. Classes with higher CBO and RFC are harder to analyse due to greater number of couplings with other methods and objects. Abstract packages (higher A) and classes with high DIT are harder to analyse because of a larger number of (abstract) classes that need to be kept in mind while reading the source code. Systems with higher NC and NOM are less analysable for the same reason. Packages with higher Ce are less analysable due to a greater number of dependencies. Higher CD improves analysability.

H. Documentation

The high-level quality concept of documentation can be directly mapped to the source code metrics of documentation.

I. Modifiability

Following [3], we define modifiability as the degree to which a program can be changed without introducing errors. Intuitively, modifiability depends on modularity and coupling. Thus, LCOM, CBO, RFC are important predictors for modifiability. Classes that serve as a superclass for a larger number of other classes (higher NOC) are more difficult to modify without breaking their children classes. Packages with lower instability I and higher Ca are less modifiable, due to a greater number of dependencies.

J. Testability

According to [2], testability is the effort required to test a program. A method with fewer parameters NP is easier to test than a method with a greater number of parameters. Packages with a lot of abstract classes (higher A) are harder to test thoroughly. Methods with higher CC are less testable because of the large number of execution paths through them. Classes with high NOM are harder to test due to a large number of required unit tests. Finally, classes with high CBO need more testing effort to eliminate dependencies and write test stubs.

K. Portability

Portability is the degree to with which a program can be transferred from one environment to another [3][2]. Larger projects are more likely to contain dependencies that might prevent portability, so LOC and NC can be used to predict portability. Some languages provide a unified run-time for all platforms (e.g. Java), while others target specific platforms (e.g. .NET). Separately, some languages have run-times available on many platforms while others do not. Therefore, L is an important predictor for portability.

L. Resilience

We define resilience to be the degree to which the software can be expected to operate in unexpected environments. Modular software that is written in a language that handles exceptions and is based on resilient run-time environment can be expected to be more resilient. Modularity indicators such as CBO, RFC, Ca, and Ce can be used to indicate resilient software. Software with a high defect density (DD) and/or defects arriving at a high rate (DAR) is assumed to be less resilient. Greater NOM and NP values may indicate non-resilient software due to a larger data boundary surface. A high number of static analysis violations (S) shows that the software is not resilient to improper input. Finally, some languages allow better error handling than others. Thus, L can be a predictor for resilience.

V. Implementation for Reliability

In the following, we demonstrate the mapping of quality attributes to metrics proposed in Sec. IV. Given space limitations, we do this for a single quality attribute: reliability. Mappings of other quality attributes (to their corresponding metrics) can be designed and implemented analogously. We demonstrate how a high-level expectation about the product under construction's reliability can be transformed into a set of requirements for the source code, compliance to which, in principle, can be checked in an automated fashion.

A. Estimation and relevant metrics

Reliability is likely one of the most often mentioned software quality attributes [20]. Several efforts to quantify software reliability have been made early on within NASA and AT&T [8][9]. The field of software reliability borrows some models from the field of system reliability and hardware reliability [9]. When some characteristics cannot be directly measured, but an estimation (value) thereof is required, an operational or 'proxy' measure is often used. Known operational measures for reliability are the number of reported defects, the mean time between failure, and the defect density. Measuring reliability of a working system is traditionally called reliability estimation. Establishing future reliability of a running software system by using sources other than the running program itself is referred to as reliability prediction.

We use defect density as an operational measure of reliability, and use LVQ1 learning algorithm to create prototype vectors for reliable and unreliable software, in the metrics space (metrics extracted from the source code, the information about the development process, the information extracted from the repository.) This model can further be used to classify the system under construction as reliable or unreliable, We identify the following high-level factors, and their related metrics, that affect reliability:

1. *Developer experience*, represented indirectly by CC/LOC, LOC/NOM, NOM/NC, NC/NOP;

2. *Program complexity*, measured indirectly by S/DEV, D/DEV, NOM/DEV, LOC/DEV, NC/DEV;

3. *Testing effort*, measured indirectly by test coverage C and the proportion of the failed tests F;

4. The *density of errors* S/LOC found in the source code by a static analysis tool. Any tool or number of tools can be used, provided that the set of tools is not changed during the model creation. At the simplest level, compiler warnings can be used to estimate S. More complex static analysis tools, such as discussed in Sec. IIIC, provide more comprehensive measurements for S.

To assess our proposal, we use the toolset in [10] to compute the above-mentioned quality factors from their respective metrics for the following open-source software projects: nginx, Checkstyle, rocksdb, MySQL Server, AngularJS, jQuery, cMake, Notepad++, jUnit. Defect density information is

extracted separately using bug tracking services for the above-mentioned software projects.

Due to limitations related to data availability (more specifically, limitations of the used static analysis tooling for providing the required data), we exclude from our reliability model the metrics that are related to classes and packages. We also leave out metrics related to testing, since there is no test coverage data available for some of the projects in our considered set.

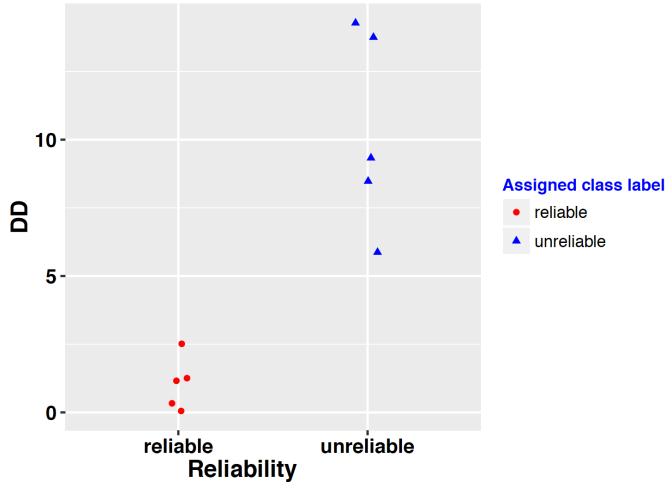


Figure 3: Reliability as a function of the defect density (DD)

B. Learning a reliability model

With the metrics extracted as indicated in Sec. VA, we next aim to construct, or learn, a predictive model – that is, a model that, given such measured values on a software system, can infer whether the system's reliability meets the expressed expectations. For this we use a machine learning approach, as follows.

Learning vector quantization 1 (LVQ1) is a supervised learning algorithm that is aimed at creating a set of prototype vectors for each class present in its input data [7]. Putting it simply, LVQ1 maps multi-dimensional input data (vectors) to a nominal, or categorical, scale (class labels). As input vectors, we use the measured quantitative (real-valued) metrics outlined in Sec. VA. Based on these vectors, and a labelling of the training set, LVQ1 computes a set of so-called prototypes, i.e., points in the multi-dimensional metric space that best approximate surrounding clusters of labelled observations (software systems) having the same class label. Using these prototypes, LVQ1 finally assigns a label to untrained samples (software systems for which we do not know the reliability) using a k-nearest-neighbours (kNN) approach. Compared to other classification approaches, LVQ1 has advantages when manually finding correlations between specific subsets of features (metrics) and feature values in the input data, and class labels (from the training data), is hard to do due to high dimensionality of the input data.

We simplify the modelling of reliability by considering a two-class problem – that is, we aim to classify our software systems into either reliable or unreliable. While this is, clearly, an oversimplification of the real world where reliability is better modelled as a quantitative (or ordinal) variable, this simplification allows us to easily create labelled data as well as train and test the LVQ1 classifier based on a small number of sample points (software systems). If thousands of sample points, including their metric values and label values, are available, precisely the same training-and-testing approach described here can be used to learn more complex reliability models.

C. Dataset

Table I shows the input dataset for our classifier construction, containing nine sample points and their respective six measured software metrics. In the context of policy driven development, the team can use reliability data from the previous releases of the product under development, or from similar products either within the company or open-source. The class label (rightmost column) has the binary values reliable (R) or unreliable (U). We label the systems with DD < 5 (D/KLOC) as reliable, and those with DD > 5 as unreliable. When implemented in practice, this labelling must be performed by the members of the team in collaboration with the stakeholders and reflect their common understanding of the reliability of the product(s) that is(are) used as a benchmark. During this process, we also use scatterplots to examine how class labels correlate with the measured metrics. Figure 3 shows such an example. Here, the vertical axis maps defect density (DD) and the horizontal axis maps the assigned class label (R or U). In this example, we clearly see how the examined systems are grouped into two clusters based on the DD values.

TABLE I. SOFTWARE METRICS FOR TRAINING AND TESTING

Name	KLOC	NOM	CC	Devs	D	S	Class
AngularJS	119.0	4917	17126	1504	696	2022	U
Checkstyle	31.5	2571	6770	107	10	6	R
cMake	182	4529	26341	520	1700	440	U
jQuery	6	578	2201	291	51	14	U
jUnit	9.5	1321	2433	155	136	46	U
MySQL server	2787	27874	204670	1368	3534	4700	R
Nginx	122	1304	17575	46	9	70	R
Notepad++	78	1722	15468	81	1070	162	U
rocksdb	197	7141	19638	284	228	78	R

To train the model, we exclude two random rows from Table I and perform LVQ1 training for the remaining rows, using 2000 training iterations. We use two prototypes, one modelling the R label and one for the U label, respectively. Table II shows the coordinates (i.e., the metric values) of these two trained prototypes. Finally, we use the excluded two prototypes to validate the trained model.

TABLE II. PROTOTYPE VECTORS FOR THE TWO LEARNED CLASSES

Class	CC/LOC	NOM/DEV	KLOC/DEV	S/KLO	D/DEV	LOC/NOM	S/DEV
Reliable	0.08	20	2.07	1.88	2.65	121	4.19
Unreliable	0.14	2.85	0.07	17.17	0.69	24.5	1.38

VI. DISCUSSION AND CONCLUSIONS

We next discuss our most important findings, as follows.

One interesting finding is that the amount of code per developer (KLOC/DEV, NOM/DEV) is positively correlated with defect density in all indirect metrics, in both small and large projects. Both large and small projects are less defect-prone if there are only a few developers that work each on a large chunk of code. Our belief is that this might only hold true for open source software (OSS) and stem from the way in which OSS is maintained and how the OSS community functions. Separately, we see that the number of raised static analysis warnings is strongly correlated with defect density. This can be looked at from two possible standpoints: First, it may indicate a sloppy developer attitude that manifests itself by ignoring the compiler and static analysis tools' warnings. Separately, this can indicate that higher error density in a codebase leads to a higher defect density in the resulting software – a correlation which seems very likely. Another interesting finding is that the average size of a method (LOC/NOM) is negatively correlated with defect density in our model. This is in line with many earlier studies where understandability and thus maintainability was inversely correlated with the average method size.

Apart from these interesting findings, we however have to mention a number of threats to validity for our study. First and foremost, we used a quite small sample set (9 software projects). While most papers in software quality literature that we are aware of use datasets of similar size, this is more problematic in our case, where we use a machine learning approach to learn a model for reliability. Such approaches typically need hundreds of labelled samples to arrive at a good balance between generalization and overfitting. Secondly, we considered only a subset of all existing software quality metrics that analysis tools can deliver. Adding new metrics may offer different insights. Doing this is, conceptually, easy, but it requires the availability of easy-to-use and generic metric tools that cover a rich palette of metrics, programming languages, and platforms – a desiderate not yet met by the state-of-the-art in software metric tooling. Finally, while generalizing our approach to the other NFRs listed in Sec. IV is, conceptually, straightforward, doing this in practice and assessing the quality of obtained predictions is necessary to further strengthen the practical added-value of our proposal.

REFERENCES

[1] Ariola, W. and Dunlop, C. [2015] Continuous testing for IT leaders. CreateSpace Independent Publishing

[2] McCall, J.A., Richards, P.K. and Walters, G.F. [1977] Factors in software quality, RADC TR-77-369, vols I-III, US Rome Air Development Center Reports

[3] ISO/IEC [2010] ISO/IEC 25010 system and software quality models. Technical report.

[4] Glinz, M. [2007] On non-functional requirements, Proc. 15th IEEE International Requirements Engineering Conference, 21–26

[5] IEEE [2011]. Systems and software engineering – life cycle processes – requirements engineering, ISO/IEC/IEEE 29148:2011(E)

[6] Robin, M.C. and Micah, M.[2006]. Agile Principles, Patterns, and Practices. Prentice Hall.

[7] Kohonen, T. [1995]. Self-Organizing Maps. Springer.

[8] Musa, J., Iannino, A. and Okumoto, K. [1990] Software Reliability. McGraw-Hill.

[9] Pham, H. et al[2003] Handbook of Reliability Engineering, Springer.

[10] SonarSource SA, https://sonarcloud.io/, SonarSource SA. Available: https://sonarcloud.io/organizations/default/projects?sort=-size. [Accessed 12 06 2016]

[11] Lanza, M. and Marinescu, R. [2006] Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer.

[12] Chidamber, S. and Kemerer, C. [1994] A metrics suite for object oriented design, IEEE Trans Soft Eng 20(6), 476-493

[13] Voinea, L. and Telea, A. [2007] Visual data mining and analysis of software repositories. Computers & Graphics 31(3), 410-428

[14] Mens T. and Demeyer, S. [2001] Future trends in software evolution metrics. Proc. IEEE IWPSE, 83-86

[15] Telea, A. and Voinea. L [2007] An interactive reverse engineering environment for large-scale C++ code. Proc. ACM SOFTVIS, 67-76

[16] Ferenc, R., Siket, I., and Gyimothy, T. [2004] Extracting facts from open source software. In Proc. IEEE ICSM, 342-350

[17] Mens, T. and Demeyer, S. [2008] Software Evolution. Springer

[18] SciTools, Inc. [2018] Understand reverse-engineerin tool. http://www.scitools.com

[19] Bullseye, Inc. [2018] Code coverage analyzer. http://www.bullseye.com

[20] I. Gorton [2011] Essential Software Architecture (Ch. 3: Software Quality Attributes). Springer