An Empirical Investigation of Modularity Metrics for Indicating Architectural Technical Debt

Zengyang Li
Department of Mathematics and Computer Science
University of Groningen
Nijenborgh 9, 9747 AG
Groningen, The Netherlands
(+31) 50 363 7127
zengyang.li@rug.nl

Peng Liang
State Key Lab of Software Engineering, School of Computer
Wuhan University
Luojiasha 430072
Wuhan, China
(+86) 27 6877 6137
liangp@whu.edu.cn

Paris Avgeriou
Department of Mathematics and Computer Science
University of Groningen
Nijenborgh 9, 9747 AG
Groningen, The Netherlands
(+31) 50 363 7057
paris@cs.rug.nl

Nicolas Guelfi
Computer Science and Communications Research Unit
University of Luxembourg
6, rue Richard Coudenhove-Kalergi
L-1359 Luxembourg, Luxembourg
(+352) 46 66 44 5251
nicolas.guelfi@uni.lu

Apostolos Ampatzoglou
Department of Mathematics and Computer Science
University of Groningen
Nijenborgh 9, 9747 AG
Groningen, The Netherlands
(+31) 50 363 5181
a.ampatzoglou@rug.nl

ABSTRACT

Architectural technical debt (ATD) is incurred by design decisions that consciously or unconsciously compromise system-wide quality attributes, particularly maintainability and evolvability. ATD needs to be identified and measured, so that it can be monitored and eventually repaid, when appropriate. In practice, ATD is difficult to identify and measure, since ATD does not yield observable behaviors to end users. One indicator of ATD, is the average number of modified components per commit (ANMCC): a higher ANMCC indicates more ATD in a software system. However, it is difficult and sometimes impossible to calculate ANMCC, because the data (i.e., the log of commits) are not always available. In this work, we propose to use software modularity metrics, which can be directly calculated based on source code, as a substitute of ANMCC to indicate ATD. We validate the correlation between ANMCC and modularity metrics through a holistic multiple case study on thirteen open source software projects. The results of this study suggest that two modularity metrics, namely Index of Package Changing Impact (IPCI) and Index of Package Goal Focus (IPGF), have significant correlation with ANMCC, and therefore can be used as alternative ATD indicators.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics- *Product metrics*; D.2.11 [**Software Engineering**]: Software Architectures - *languages*

General Terms

Measurement, Experimentation

Keywords

Architectural technical debt; modularity metric; commit; software architecture

1. INTRODUCTION

Technical debt has been increasingly gaining attention from researchers in the software engineering domain and from practitioners in the software industry in the past years [3; 9; 16]. The concept of technical debt was coined by Ward Cunningham to describe immature work in coding that can yield short-term benefit (e.g., fast delivery), but will lead to high maintenance and evolution cost in the long term [4]. Technical debt can span all phases of the software development lifecycle, including requirements analysis, architecture design, detailed design, testing etc. [8]. More generally, technical debt refers to immature work in a software system that takes compromises in one dimension to meet urgent needs in some other dimension [3]. In this work, we focus on the technical debt at architecture level [11], i.e., architectural technical debt (ATD).

ATD is caused by design decisions that consciously or unconsciously compromise system-wide quality attributes (QAs), especially maintainability and evolvability [8; 10]. Typical ATD includes violations of best architecture practices and breakages of the consistency and integrity of software architectures. An example of ATD is the creation of architecture dependencies that violate the strict layered architectural pattern, i.e., a higher layer having direct dependencies to layers other than the one directly below it. ATD may also include the adoption of immature architecture techniques. Another ATD example is the use of an immature web application framework, which might require significant modifications, and therefore extra effort, to adapt in the developed web application.

By taking into account the negative impact on the long-term health of a software system, ATD needs to be effectively managed to keep its amount under a reasonable level. Management of ATD entails identifying and measuring it, so that it can be monitored and eventually repaid [10]. However, in practice ATD is difficult to identify and measure, since ATD does not yield observable behaviors to end users [3; 10; 16]. One solution is to define ATD *indicators* that denote the presence and relative amount of ATD. One such indicator is the average number of modified components per commit (further referred as ANMCC). A commit, also called a revision, is a unit of modification to the source code of a software system. ANMCC indicates the presence of ATD as follows:

- ANMCC reflects the complexity and difficulty of making changes to a software system. A high ANMCC means that in a specific revision, and in order to perform a maintenance task (e.g., debugging or implementing a new feature) many components had to be modified. This fact indicates a difficulty in performing maintenance activities, due to high coupling between components, and intensive ripple effects.
- The increase of the complexity and difficulty of making changes to a software system is the consequence of accumulated ATD. If not repaid, ATD will continuously accumulate interest, which makes the system more complex and difficult to implement changes later on [3; 4].
- A higher ANMCC entails an increase in the complexity and difficulty to change, thus implying potential increase in ATD.

However, it is hard and sometimes even impossible to calculate ANMCC, because the commit records (i.e., history data of source code changes) are not always available. For instance, a legacy system may not have commit history data; or a system that is built based on reused components from different projects has no complete commit history data. To address this issue, we try to find a substitute for ANMCC indicator that can be calculated based on source code; such a substitute should be accurate (ground truth representation of the system) and available.

In order to identify such a substitute we look into *modularity metrics*. According to ISO/IEC 25010 standard [7], modularity is one of the sub-characteristics of maintainability, which is one of the QAs compromised by ATD. Modularity is defined as the "*degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components* [7]". A snapshot of the source code of a project is the result of previous changes (commits). The modularity metrics of a snapshot of the project source code can, to a certain extent, reflect the development difficulty of changes to this project in the near future [19]. Specifically, as the modularity of a software system increases, the ANMCC of this software system is expected to decrease. Consequently, system modularity, to a certain degree, can also substitute ANMCC in terms of indicating ATD.

In this work, we empirically investigate the ability of existing modularity metrics to substitute ANMCC as ATD indicators, through a holistic multiple case study on thirteen open source projects. The main contribution of this work is the empirical evidence supporting that two of the investigated modularity metrics, i.e., Index of Package Changing Impact (IPCI) and Index of Package Goal Focus (IPGF), have significant negative correlations with ANMCC – the ATD indicator. Thus, the two software modularity metrics (IPCI and IPGF) can be used as indicators of ATD of a system. The merit of using IPCI and IPGF as ATD indicators is that they can be automatically calculated using a single version of source code, while the calculation of ANMCC requires commit history information of a project which is not always available.

The remainder of this paper is organized as follows: we discuss related work on technical debt measurement, especially ATD measurement in Section 2. The case study design is illustrated in Section 3. Section 4 describes the results of the case study and Section 5 discusses the study results and their implications. The threats to the validity of the case study are identified in Section 6. We conclude this work with future work directions in Section 7.

2. RELATED WORK

Technical debt measurement is considered as an important step in the technical debt management process [18]. Although technical debt is not easy to measure [3], there have already been some attempts trying to estimate it at various levels (e.g., code level, architecture level) and from different perspectives.

In [3], Brown et al. proposed to aggregate individual technical measures of technical debt in three aspects similarly to financial debt: principal, interest probability, and interest amount. The total technical debt is the sum of the principal, and the product of interest probability and interest amount. Seaman and Guo measured these three aspects of a technical debt item by assigning them values of high, medium, or low [18]. They suggested that these coarse-grained estimates should be sufficient for tracking technical debt items and making preliminary decisions on technical debt management. When more required information (e.g., historical effort data) is available, fine-grained estimations can be made upon that information for refined management decision-making.

Curtis et al. estimated technical debt by calculating the cost of fixing different types of violations (e.g., code smells) that were identified through automatic static analysis of source code against rules of good architecture and coding practice [5]. They analyzed millions of lines of source code of business applications collected from various companies in different application domains. These collected applications were written in 28 programming languages. The principal of technical debt can be calculated through the following formula [5]:

Principal =
((Σ high-severity violations) × (percentage to be fixed) × (average hours needed to fix) × ($ per hour)) +
((Σ medium-severity violations) × (percentage to be fixed) × (average hours needed to fix) × ($ per hour)) +
((Σ low-severity violations) × (percentage to be fixed) × (average hours needed to fix) × ($ per hour)).

When the percentages of high-, medium-, and low-severity violations to be fixed are 50%, 25%, and 10%, respectively, fixing each violation takes one hour and the labor cost is 75 US dollar per hour, the average estimated technical debt principal is 3.61 US dollar per line of code in the aforementioned collected source code. The technical debt principal per line of source code differs among programming languages. However, there are some issues with estimating the technical debt of a concrete software system with fixed value (i.e., 3.61 US dollar per line of code). Usually, architectural violations are much more difficult to fix compared with the design-level and code-level violations. In addition, the cost of fixing the same type of violations differs largely in different contexts of various software projects.

Marinescu proposed an approach to identify and measure technical debt of object-oriented software systems by detecting and assessing specific types of design flaws through object-oriented metrics [12]. The approach is composed of four steps: (1) choose a set of concerned design flaws, (2) define rules for detecting the selected design flaws, (3) measure the negative impact of each instance of the design flaws, and, finally, (4) calculate an overall score based on all detected design flaws to indicate the design quality of a system. The accuracy of the technical debt measurement in this approach depends on the ability of the design flaws detection. This approach can only identify and measure technical debt at detailed design level, while our investigation focuses on technical debt at architecture level.

Nord et al. defined a metric for managing ATD [15]. The value of this metric, calculated for each release, is the total cost of the implementation of new architectural elements introduced in this release, and the rework of pre-existing elements in previous releases. They considered architectural rework as the necessary adaption work for adding new architectural elements to the existing architecture of a software system. The rework cost is calculated based on the analysis of the changing dependencies from existing adapted architectural elements to the new introduced elements. This metric can be used to calculate the relative amount of ATD incurred in different software evolution paths, i.e., release plans. Suppose that there are two release plans RP1 and RP2, in which the same features are implemented, i.e., they generate the same amount of business value. The relative amount of ATD is the difference between the values of metric calculated on RP1 and RP2. Thus, this metric can facilitate architecture decision-making in ATD management. The main limitation of this approach is the accuracy of the estimation of implementing new features and rework, especially the latter. Each software evolution path involves several releases, which implies that the estimation of rework and new implementations of later releases is based on the estimation of the earlier releases. This may pose a significant threat to the accuracy of ATD estimation.

In our work, we consider that the estimation of ATD should be calculated on real data (i.e., source code), and the estimation makes more sense within a relative short term, e.g., between two releases. That is, the estimation of the next release is based on the real data of this implemented release.

3. CASE STUDY DESIGN

In order to investigate the ability of modularity metrics to substitute the average number of modified components per commit (ANMCC), and thus act as alternative indicators of system's ATD, we performed a holistic multiple case study on thirteen C# open source software (OSS) projects provided by GitHub[1]. The main reason for conducting a case study is that, through using OSS projects, and more specifically their source code and commit information, we examine the phenomenon in its real-life context, since both factors, i.e., modularity and ANMCC, cannot be monitored in isolation, and their environment cannot be controlled. In this section we describe the case study, which was designed and reported according to the guidelines proposed by Runeson and Host [17].

[1] https://github.com/

3.1 Objective and Research Questions

The goal of this study, described using the Goal-Question-Metric (GQM) approach [2], is: *to analyze modularity metrics for the purpose of evaluation with respect to their ability to act as substitutes of ANMCC, for indicating ATD, from the point of view of software architects in the context of OSS evolution*.

Based on the abovementioned goal, we have extracted two research questions (RQs):

RQ1: Are there modularity metrics that correlate with ANMCC?

RQ2: Which modularity metrics have the most accurate correlation with ANMCC?

3.2 Case and Unit Analysis

According to [17], case studies can be characterized based on the way they define their cases and units of analysis. This study is a holistic multiple case study, in the sense that we investigate multiple OSS projects, i.e., cases, and from each case we extract a single unit of analysis. In this study, as unit of analysis we use the pair of two selected releases of an OSS project.

3.3 Case Selection

In this study, we only investigate C# OSS projects, since we make use of the functionalities provided by Microsoft Visual Studio 2012 (MS VS2012). Specifically, the functionality of code map generation can create detailed and complete reports on the structure of a software system, and the reports on the software structure can be used to calculate modularity metrics.

For selecting cases (OSS projects) included in our study, we apply the following criteria:

1. Each selected project should have at least 6 releases, so that we can choose two neighboring releases that meet the release selection rules that are defined in the later part of this section. If a project has only two releases, it may still be in very early development stages, leading to tremendous changes between two neighboring releases. Thus, any estimation for the second release based on the first release is not likely to be reliable.
2. Each release of a selected project should have at least 5 components. With this criterion, the modularity concept for a software system makes sense, in the sense that a system with less than five components is either not modular, or small in terms of size.
3. The full list of commit records of the selected project can be automatically extracted using the TortoiseSVN[2] client. The TortoiseSVN client is a user-friendly tool that can automatically export the complete commit records of a project from most SVN servers.

The source code of the selected project should be successfully compiled, which is the prerequisite of generating code maps using MS VS2012. The code maps generated by VS2012 are used as input for the modularity metrics calculation.

Since a unit of analysis is the pair of two selected releases of the OSS, we need to set rules to ensure that the selected releases are reasonable. The rules are defined as follows:

1. The difference between the numbers of components of the two selected releases is relatively small (<=2).

[2] http://tortoisesvn.tigris.org

2. The difference between the numbers of types (e.g., classes, interfaces) of the two selected releases should not be too small (>=10);
3. The number of commits between the two selected releases should not be too small (>=15).

The first two rules ensure that the OSS project did not dramatically change, but still changed significantly, and the third rule helps to reduce the unevenness of changes over commits.

3.4 Data Collection

3.4.1 Dataset

For each unit of analysis, we have recorded seven variables, six modularity metrics (V1-V6) and the ANMCC value, i.e., the ATD indicator (V7), as follows:

V1. **Index of Inter-Package Usage (IIPU)** is the ratio of the number of *Use* dependencies between classes within a local package against the total number of *Use* dependencies between classes of the whole software system [1].

V2. **Index of Inter-Package Extending (IIPE)** is the ratio of the number of *Extend* dependencies between classes within a local package against the total number of *Extend* dependencies between classes of the whole software system. The *Extend* dependency here can be the inheritance relationship between two classes or the implementation relationship between a class and an interface [1].

V3. **Index of Package Changing Impact (IPCI)** is the percentage of the number of the non-dependency package pairs against the total number of all possible package pairs. This metric measures the strength of the independency of packages [1].

V4. **Index of Inter-Package Usage Diversion (IIPUD)** is the average extent of how diverse the classes used by a specific package distribute in different packages [1].

V5. **Index of Inter-Package Extending Diversion (IIPED)** is the average extent of how diverse the classes extended by a specific package distribute in different packages [1].

V6. **Index of Package Goal Focus (IPGF)** is the average extent of the overlap between the different service sets provided by the same component to other different components in a software system [1]. IPGF indicates the average extent that the services of a specific package serve for the same goal.

V7. **Average Number of Modified Components per Commit (ANMCC)** is the average number of components that are modified during each commit (i.e., revision) in the studied period.

The value of each modularity metrics falls in the range [0, 1]. A greater value of a modularity metric indicates that the software system is better modularized. Finally, in order to mitigate the influence of project size on the ANMCC value, for data analysis, we have used the normalized value of ANMCC. We normalize the ANMCC value by dividing ANMCC with the number of the components (as a representation of project size) of the early release in the two selected releases. All the modularity metrics are calculated by the *ModularityCalculator* tool, while the ANMCC is calculated by the *CommitAnalyer* tool. Both tools are developed by the authors and publicly available[3].

Note that in this study we define a component as an assembly[4] in C# software projects.

3.4.2 Data collection method

Figure 1 shows the data collection method of this case study. More specifically, for each selected C# OSS project, we need to collect its full list of commit records and the source code of a set of releases. The former is used to calculate the ANMCC, and the latter is used to calculate the modularity metrics of releases. A commit record is the log information of changes to the source code repository during this commit.

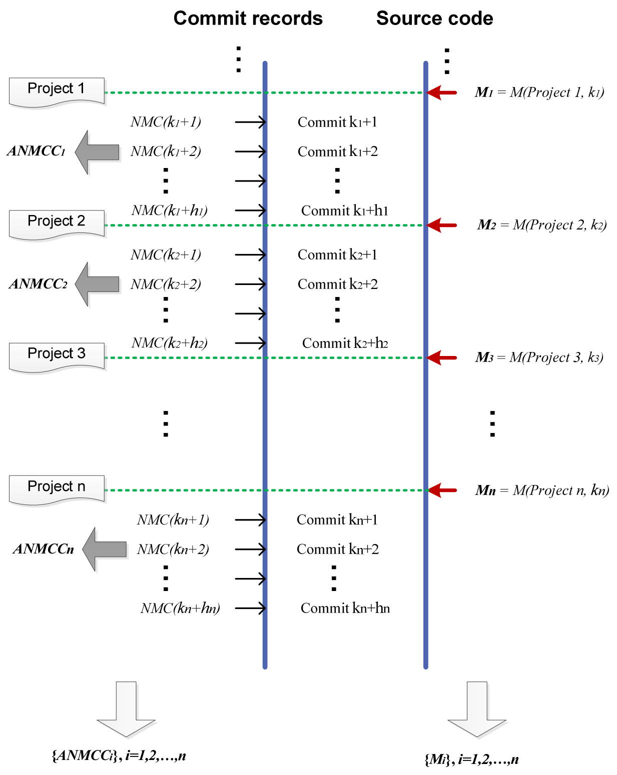

Figure 1. The method to collect the data for calculating ANMCC and modularity metric M.

Suppose that there are n OSS projects. For project i, the two selected releases are releases i_1 and i_2, k_i is the total number of commits of the first i_1 releases, and release i_2 has h_i commit records. In Figure 1, $NMC(k_i+j)$ denotes the number of modified components in commit k_i+j of project i, and $ANMCC_i$ denotes the value of ANMCC during release i_2 (i.e., between releases i_1 and i_2) of project i. Thus, we use formula (1) to calculate $ANMCC_i$:

$$ANMCC_i = \left(\sum_{j=1}^{h_i} NMC(k_i + j)\right)/h_i \qquad (1)$$

M_i denotes the value of the modularity metric M of release i_1 of project i. M_i is calculated based on the source code of release i, i.e., the commit k_i.

[3] http://www.cs.rug.nl/search/uploads/Resources/ATDAnalysisTools.zip

[4] In C# (or generally in .NET) "*an assembly is a collection of types and resources that forms a logical unit of functionality [13].*"

3.4.3 Data collection procedure for modularity metrics

The task of data collection for modularity metrics calculation is performed in four steps that are described below:

(1) **Source code download**. The source code of each release of a selected OSS project can be downloaded and stored locally using the TortoiseSVN client for further analysis.

(2) **Code map generation**. The goal of this step is to get the structure data of the selected OSS projects. The code map of a version of the source code of a C# OSS project is an XML file that contains the structure data of all the software elements (e.g., assembly, class, and method) and links between them. We generate the code maps for all releases of the selected C# OSS projects using VS2012. For an OSS project, there are two types of code that should be filtered out when generating the code maps: 1) test code (e.g., unit tests, integration tests) and 2) code of examples that show how to use the functions and APIs provided by the functional part. The reasons for excluding these two types of code are that: test code will not be delivered to users, and code of examples is not related to the internal quality of the OSS. But both types of code are tightly coupled with the functional code and can seriously reduce the modularity of software systems, and consequently should be removed from modularity metrics calculation.

(3) **Code map parsing**. Since the code maps generated by VS2012 are too complicated to understand and use, we use our tool *CodeMapParser* to parse the generated code map into a simplified and clean format that is easier to handle than the original format. This *CodeMapParser* tool is available together with the other two tools used in this work[5].

(4) **Modularity metrics calculation**. We use the tool *ModularityCalculator* to calculate the modularity metrics (V1-V6) presented in Section 3.4.1 based on the simplified form of the code map data generated in the previous step. For each selected OSS project, this tool can generate a report in Microsoft Excel format, which contains the modularity metrics of all releases of the project.

3.4.4 Data collection procedure for ANMCC

The goal of this task is to calculate the average number of the components that are modified in each commit (i.e., ANMCC, the ATD indicator) in the selected projects. For each project, we need to extract all the commit records and to identify the component that each modified source code file belongs to in every commit. The detailed steps of the data collection procedure for ANMCC are described as follows:

(1) **Commit records download**. The commit records of the selected OSS projects can be downloaded using the TortoiseSVN client, which can automatically extract a complete list of commit records of a project. With the TortoiseSVN client, we can extract commit records from standard SVN servers and any code repositories supporting Subversion, such as GitHub.

(2) **Commit records parsing**. We need to parse the commit records to extract needed data items for ANMCC calculation. This step can be performed using our developed tool *CommitAnalyzer*. The extracted data items include the start and end commit numbers of each release and the list of files modified in each commit.

(3) **Commit records filtering**. Some data in commit records are invalid for the ANMCC calculation and therefore need to be filtered out. First, the data on the test code files should be removed, and second, the data on the code files of examples should also be removed for the same reasons we presented in Section 3.4.3. The tool *CommitAnalyzer* can semi-automate the commit records filtering with human intervention to confirm which source code directories contain the invalid data.

(4) **ANMCC calculation**. In order to calculate the ANMCC, we need to identify the component that a modified source code file belongs to in every commit, and the release that each commit record belongs to. The tool *CommitAnalyzer* also provides the functionality for calculating ANMCC.

3.5 Data Analysis

In order to explore the research questions, set in section 3.1, we will investigate the correlations between the modularity metrics and ANMCC. Intuitively, we expect that there are negative correlations between modularity metrics and ANMCC. There are two candidate correlation tests, i.e., the Pearson correlation coefficient and Spearman's rho [6]. Pearson correlation coefficient is a parametric test, used to measure the strength of a linear association between two variables. Spearman's rho is a non-parametric test used to measure the strength of monotonic association between two variables. The values of both Pearson correlation coefficient and Spearman's rho range in [-1, 1], where the value 1 means a perfect positive correlation, and the value -1 means a perfect negative correlation. Using the Pearson correlation coefficient requires that two variables for the correlation calculation follow normal distributions, while using the Spearman' rho does not have such a requirement. To select the appropriate correlation calculation method, we need to check the normality of the variables (i.e.,V1-V7), through a Shapiro-Wilk's test [6].

Concerning RQ1, we choose the appropriate correlation test according to the results of the Shapiro-Wilk's tests. We use the correlation coefficient value of the selected correlation test and the level of statistical significance, for each correlation. Next, concerning RQ2, we use the Hoteling-Williams test [6], in order to test possible differences among the predictive ability of different modularity metrics, which appear to be significantly correlated to ANMCC, in RQ1.

All statistical tests will be performed with Matlab by one author, and will be validated with SPSS by another author.

4. CASE STUDY RESULTS

We analyzed thirteen OSS projects by following the case study design presented in Section 3. The list of the selected OSS projects along with demographic information is shown in Table 1, where: "#Release" is the number of all the releases of the project, "#Component" is the number of the components of the latest release of the project, "#Type" is the number of the types of the latest release of the project, and "#Commit" is the number of all the commits of the project. The data of the aforementioned four columns describe the sizes and change frequency of the selected OSS projects.

The rest of this section presents the collected dataset and the results of the correlation tests between the modularity metrics and ANMCC.

4.1 Collected Dataset

The selected releases and their demographic information of the thirteen selected OSS projects are shown in Table 2, where: the

[5]http://www.cs.rug.nl/search/uploads/Resources/ATDAnalysis Tools.zip

column “Release 1” is the early release of the corresponding project; the column “Release 2” is the later release; the columns “#Component” and “#Type” are the number of components and the number of Types of Release 1, respectively; the column “#Commit” is the number of commits during the period between Release 1 and Release 2; and the “Δ(#Component)” and “Δ(#Type)” are the difference of the numbers of components and types between Release 1 and Release 2, respectively.

As shown in Table 3, the dataset has thirteen data rows, and each data row is collected from a different C# OSS project. A data row in Table 3 includes two parts: the modularity metrics and (normalized) ANMCC. The former is calculated based on the source code of an early release (i.e., release 1 in Table 2), and the latter is calculated based on the commit records that occurred during the period between the early release and later one (i.e., release 2 in Table 2).

Table 1. Selected OSS projects in the case study

#	Name	#Release	#Component	#Type [a]	#Commit	Duration	URL
1	Cassette	8	13	398	2022	1.5 years	github.com/andrewdavey/cassette
2	CastleCore	12	6	569	6744	9.0 years	github.com/castleproject/Core
3	CCNET	28	14	1093	6359	10.5 years	github.com/ccnet/CruiseControl.NET
4	ILSpy	7	14	2641	1706	6.0 years	github.com/icsharpcode/ILSpy
5	MassTransit	20	17	960	4165	6.0 years	github.com/phatboyg/MassTransit
6	Nancy	25	20	493	3471	4.5 years	github.com/NancyFx/Nancy
7	NSpec	38	5	162	644	2.5 years	github.com/mattflo/NSpec
8	NUnit	20	20	861	3723	10.0 years	github.com/nunit/nunitv2
9	Rebus	87	18	304	1257	2.0 years	github.com/rebus-org/Rebus
10	Scriptcs	9	5	120	842	0.5 year	github.com/scriptcs/scriptcs
11	SignalR	23	18	598	18978	2.5 years	github.com/SignalR/SignalR
12	SimpleData	21	9	307	774	3.0 years	github.com/markrendle/Simple.Data
13	SolrNet	11	9	301	1782	6.0 years	github.com/mausch/SolrNet

[a] A Type in C# can be a Class, Interface, Enum, Delegate, or Struct

Table 2. Selected releases and their demographic information

#	Project	Release 1	#Component	#Type	#Commit	Release 2	Δ(#Component)	Δ(#Type)
1	Cassette	v2.0.0	12	327	134	v2.1.0	1	71
2	CastleCore	v3.0.0	6	558	44	v3.1.0	0	10
3	CCNET	v1.3.0	8	547	239	v1.4.1	0	52
4	ILSpy	v1.0.0-M3	8	1772	179	v1.0.0-Beta	1	68
5	MassTransit	v1.x.eol	18	564	107	v2.0b1	-1	70
6	Nancy	v0.7.1	12	241	155	v0.8.1	0	32
7	NSpec	v0.9.61	5	150	76	v0.9.64	0	10
8	NUnit	v2.5.9	22	767	194	v2.6.0	0	28
9	Rebus	v0.27.0	16	232	20	v0.28.0	0	36
10	Scriptcs	v0.7.0	5	109	55	v0.8.0	0	10
11	SignalR	v1.0.0a2	17	377	423	v1.1.0beta	2	25
12	SimpleData	V1.0.0-beta3	9	285	68	v0.18.1	0	22
13	SolrNet	v0.2.3	6	166	191	v0.3.0b1	1	62

Table 3. Dataset of modularity metrics and ANMCC.

#	Project	IIPU	IIPE	IPCI	IIPUD	IIPED	IPGF	ANMCC	Normalized ANMCC
1	Cassette	0.7444	0.7128	0.8939	0.8676	0.9444	0.9379	1.8284	0.1524
2	CastleCore	0.9837	0.9612	0.8667	0.9063	1.0000	0.9343	1.6136	0.2689

3	CCNET	0.8032	0.9419	0.8214	0.763	0.9028	0.8473	1.1297	0.1412
4	ILSpy	0.9017	0.9733	0.7500	0.7334	0.8311	0.7516	2.6983	0.3373
5	MassTransit	0.7930	0.8991	0.9118	0.8333	0.9259	0.9527	3.7757	0.2098
6	Nancy	0.7367	0.7755	0.9167	1.0000	1.0000	0.9355	1.6387	0.1366
7	NSpec	0.4937	0.5923	0.8500	1.0000	1.0000	0.8952	1.4737	0.2947
8	NUnit	0.5143	0.7593	0.9113	0.6640	0.8526	0.8563	2.6804	0.1218
9	Rebus	0.7943	0.7213	0.9333	0.9346	1.0000	0.9501	1.7500	0.1094
10	Scriptcs	0.3936	0.5882	0.6000	0.6493	0.8933	0.7804	2.1636	0.4327
11	SignalR	0.8702	0.8015	0.9265	0.7658	0.8822	0.9093	2.0047	0.1179
12	SimpleData	0.8043	0.7368	0.8333	0.7382	0.9306	0.8494	2.3529	0.2614
13	SolrNet	0.7691	1.0000	0.8333	1.0000	1.0000	0.8927	1.8063	0.3011

Table 4. Results of Shapiro-Wilk Test

	IIPU	IIPE	IPCI	IIPUD	IIPED	IPGF	Normalized ANMCC
W	0.899	0.925	0.803	0.917	0.880	0.891	0.903
p-value	0.096	0.296	0.007	0.231	0.072	0.101	0.145

4.2 Correlation Coefficient Results

As described in Section 3.5, we performed Shapiro-Wilk's tests on the modularity metrics and the normalized ANMCC to check their normality. The results of the Shapiro-Wilk's tests are shown in Table 4, where only the IPCI does not follow a normal distribution, with p-value <0.05 (the corresponding column is marked with gray background); the normalized ANMCC and other modularity metrics (i.e., IIPU, IIPE, IIPUD, IIPUE, and IPGF) follow normal distributions. Thus, we cannot use Pearson correlation test to calculate the correlation between the IPCI and normalized ANMCC. However, since we need to run the Hotelling-Williams' test on the correlation coefficients between the modularity metrics and normalized ANMCC, the correlation coefficients should be calculated by the same test. In order to use a uniform test for all correlations, we selected to use the Spearman's correlation test. As presented in the Introduction section, an increase of modularity indicates a decrease of ANMCC. This is a directional hypothesis for the correlation tests between the modularity metrics and normalized ANMCC, thus we use one-tailed test. In this Section, we answer the research questions stated in Section 3.1.

RQ1*: Are there modularity metrics that correlate with ANMCC?*

The results of Spearman's correlation tests between the six modularity metrics and the normalized ANMCC are shown in Table 5. The second and third columns present the resulting correlation coefficient using Spearman's rho test (shortly, *rho*) and its *p-value*, respectively.

Table 5. Correlation coefficients between modularity metrics and normalized ANMCC.

	rho	p-value
IIPU	-0.099	0.3741
IIPE	-0.104	0.3671
IPCI	**-0.828**	**0.0001**
IIPUD	-0.138	0.3261
IIPED	-0.028	0.4631
IPGF	**-0.522**	**0.0341**

As shown in Table 5, concerning IPCI the Spearman's rho is -0.828 with *p-value* 0.0001 < α=0.05, which means the IPCI has a significant negative correlation with the normalized ANMCC. In addition, the IPGF also has a significant negative correlation with the normalized ANMCC, because the Spearman's rho is -0.522, and its *p-value* is 0.0341 (less than 0.05).

The modularity metrics IIPU, IIPE, IIPUD, and IIPED, do not significantly correlate with the normalized ANMCC, since the value of the Spearman's rho of each modularity metric is close to zero and the *p-value* is way bigger than 0.05.

RQ2*: Which modularity metrics have the most accurate correlation with ANMCC?*

We used the Hotelling-Williams test to explore the possible difference in the predictive ability of IPCI and IPGF. The test result shows that IPCI and ANMCC are more highly correlated than IPGF and ANMCC. To obtain this result, we first calculated the rho between IPCI and IPGF and the resulting rho is 0.831 with *p-value*=0.0001 < α=0.05. Then, with the three *rhos* (i.e., the rho between IPCI and IPGF, rho between IPCI and ANMCC, and rho between IPGF and ANMCC), we conducted the Hotelling-Williams test, which is used to investigate if there is significant difference between two dependent correlations. We got t = -3.4838, *p-value* = 0.0059, i.e., $|t| > 1.771$ (α=0.05) => *p-value* < 0.05. Thus, we can reject the null hypothesis, i.e., equality between two dependent correlations, which means that there is significant difference between the rho values of IPCI and IPGF. In addition, the rho value of IPCI is greater than the rho value of IPGF, therefore, IPCI has a significantly stronger correlation with ANMCC than IPGF. That means IPCI is more accurate than IPGF as an alternative indicator of ANMCC.

5. DISCUSSION

In this section, we interpret the case study results and discuss their implications for researchers and practitioners in this section.

5.1 Explanation of Obtained Results

The results of the correlations between modularity metrics and ANMCC show that the modularity metrics IPCI and IPGF have a significant negative correlation with the normalized ANMCC, while the other modularity metrics (i.e., IIPU, IIPE, IIPUD, and IIPED) do not. Although the main objective of this work is not to investigate the casual relationship between modularity metrics and ANMCC, we still try to explore the potential reasons for the aforementioned correlation results.

To understand the potential reasons for the significant negative correlation between IPCI, IPGF and ANMCC, we examined the definitions of IPCI and IPGF. First, according to [1], ICPI is defined as the percentage of the number of non-dependency component pairs against the number of all possible component pairs. This metric measures to what extent other components will **not** be impacted by changes to a specific component. Intuitively, a higher ICPI indicates a smaller change propagation influence. In other words, a higher ICPI indicates that a smaller number of components will be modified in each commit (which directly links to ANMCC). Second, IPGF is defined as the extent of the overlap between the different service sets provided by the same component to other different components in the software system. IPGF indicates to what extent the services of a specific component serve the same goal. A larger value of IPGF of a software system indicates that services of components focus more on the logical goals provided by the components. Thus, to a certain degree, each component is more stable and provides services to relatively fewer client components. Therefore, the components will undergo relatively fewer modifications, and the value of ANMCC of the software system will decrease.

The results of the correlation analysis have also shown that the other four modularity metrics (i.e., IIPU, IIPE, IIPUD, and IIPED) do not have significant correlations with the ANMCC value. The potential reason for these insignificant correlations is that the calculation of the four modularity metrics does not take into account both *Use* and *Extend* dependencies at the same time. Thus, some of the dependencies are ignored in the calculation of these four modularity metrics. In these four modularity metrics, IIPU and IIPUD are defined based on the *Use* dependencies among classes, while IIPE and IIPED are defined based on the *Extend* dependencies (i.e., implementing an interface or inheriting from a class) among classes. In contrast, both *Use* and *Extend* dependencies are used in the calculation of the modularity metrics IPCI and IPGF which are in significant negative correlations with ANMCC. The ANMCC value of a software system is calculated based on all the commits occurring during the later release in the two selected releases, i.e., all the changes made during this release, and these changes can involve any one of the *Use* and *Extend* dependencies between classes. In this sense, the exclusion of either the *Use* or *Extend* dependencies in the calculation of a modularity metric can lead to a weak and insignificant correlation between this metric and ANMCC.

The result of the Hotelling-Williams test has shown that the modularity metric IPCI has a stronger correlation with the normalized ANMCC than the modularity metric IPGF. The potential reason leading to this fact is: the calculation of the ICPI metric takes into consideration the influence of all the types (e.g., interfaces) acting as services to the client components, while the calculation of the IPGF metric does not. A change to any service of a specific component may lead to the change(s) of its client component(s). When calculating the ICPI metric of a software system, the influence of all the services (e.g., interfaces) in every component on its client components has been taken into account. The IPGF metric calculates the average percentage of the overlap between the service (e.g., interface) sets that each component provides to its client components. Thus, the IPGF metric emphasizes the influence of part of the services in a component out of all services provided to its client components, i.e., the intersection of the service sets that the component provides to its client components. However, the changes of the rest services can also lead to the modifications of their client components, which is not taken into consideration in the calculation of the IPGF metric. Thus, the IPGF metric may lose some ability of correlating to the ANMCC, compared to the IPCI metric, i.e., the IPGF metric is less accurate than the IPCI metric in terms of substituting the ANMCC.

5.2 Implications for Researchers

The results of this work imply that the modularity metrics defined purely based either on the *Use* dependencies or on the *Extend* dependencies among classes may not effectively reflect the complexity and difficulty of making changes to a software system (and thus potentially ATD). We should take into account both the *Use* and *Extend* dependencies (i.e., all kinds of dependencies in a software system) when considering modularity metrics in relation to ATD.

The architecture of a software system is in a higher level and more abstract than the source code of the system, and consequently the architecture quality is harder to measure than source code. A feasible way to measure architecture quality is to relate architecture quality to software metrics based on source code; the architecture quality can then be estimated if the source code-based software metrics have a significant correlation with the architecture quality. In our case, modularity metrics are calculated based on the source code, but some of these metrics (e.g., IPCI and IPGF) can still indicate architecture-level phenomena such as ATD.

5.3 Implications for Practitioners

Based on the results of this study, we can conclude to a number of implications for practitioners. First, the modularity metrics IPCI and IPGF can be used to indicate ATD. We have provided evidence about the significant negative correlation between IPCI, IPGF and ANMCC, which means that a greater IPCI or IPGF is linked to a smaller ANMCC (indicator of the amount of ATD). Like ANMCC, IPCI and IPGF are also not absolute quantifiable measures of ATD, but they can be used to relatively suggest whether one version of a software system has more or less ATD than another version [10]. This way, architects and project managers can get informed about the potential ATD of the software system. Consequently, IPCI and IPGF can be considered as ATD indicators. A higher IPCI or IPGF indicate less ATD.

Second, IPCI and IPGF can be used to estimate the needed effort for software development in the near future (e.g., next release). The ANMCC reflects the degree of the difficulty and complexity to maintain and evolve a project, thus, it can facilitate the estimation of the needed effort of software development in the near future. Due to the significant negative correlations between IPCI, IPGF and ANMCC, the values of IPCI and IPGF can also be used to estimate effort needed of software development. Furthermore, as presented in Section 4.2, IPCI has a significantly stronger negative correlation with ANMCC than IPGF, thus, IPCI is preferable than IPGF when both metrics can be calculated with similar effort.

Third, modularity metrics can be calculated based on source code. Therefore, it is an opportunity for Integrated Development Environment (IDE) vendors to integrate such kind of ATD indicators (e.g., IPCI and IPGF) into IDE tools based on source code, which is directly available in IDE tools, for practical use. This can facilitate the ATD management in the daily work of architects and project managers as well as provide ATD indication information to developers, since they can measure and monitor ATD easily and take appropriate actions timely to prevent the situation when too much ATD is accumulated.

6. THREATS TO VALIDITY

There are several threats to the validity of the study results. We discuss these threats according to the guidelines in [17]. We note that internal validity is not discussed, because we do not investigate causal relationships.

6.1 Construct Validity

Construct validity is related to whether we can correctly use modularity metrics as substitutes for ANMCC. Both ANMCC and modularity metrics of a software system will change due to the evolution of the system. The modularity metrics of a software system at some specific point of development time can only be used to substitute ANMCC in a relatively short period after that point of time (e.g., a release of a project in this work), in which not too many commits occur. If the period is too long and too many commits happen, and the software system evolves dramatically, the modularity metrics may not be appropriate to be used to substitute ANMCC. To mitigate this threat, we have proposed three rules for release selection for each OSS project in Section 3.3 to ensure that the software system did not dramatically change but still significantly changed, and to reduce the unevenness of changes over commits.

6.2 External Validity

External validity is concerned with the generalization of the case study results. This is related to the representativeness of the selected OSS projects used in the case study. The rules for OSS project selection described in Section 3.3 may affect the representativeness of the selected projects. However, to a large extent, the project selection is random and representative. We searched OSS projects in GitHub, which is one of the largest OSS repositories. For each retrieved project, we checked if it meets all the project selection rules defined in Section 3.3. During the OSS searching and selecting process, we prevented introducing any personal preference or bias on the OSS selection. Furthermore, the selected OSS projects come from different application domains, and the projects have significantly different size and development duration. This also improves the representativeness of the selected OSS projects.

In this case study, only C# OSS projects were selected and used to validate the correlation between modularity metrics and ANMCC. Consequently, the conclusion drawn is only valid for C# projects. There is a need of conducting more studies for the projects written in other object-oriented languages, such as Java.

6.3 Conclusion Validity

Conclusion validity concerns the statistical significance of the study. In the data analysis of the case study, we carefully checked if the variables meet the prerequisites of using different statistical tests and in order not to use the wrong tests. For example, when selecting the appropriate correlation test, we checked the normality of variables (V1-V7), and then we found the variable V3 is not normally distributed. Thus, we choose the Spearman analysis rather than Pearson analysis. When conducting the Hotelling-Williams test, we used the correlation coefficients calculated by the same correlation test, i.e., the Spearman's rho test, as source data. To make sure the correctness of the statistical results, two authors separately used different tools (i.e., Matlab and SPSS) running the statistical tests and got the same results. We believe that the aforementioned actions mitigate the threats to conclusion validity.

7. CONCLUSION AND FUTURE WORK

In this paper, we provided evidence that the modularity metrics IPCI and IPGF have significant negative correlations with ANMCC – an ATD indicator. Therefore, we can consider the IPCI and IPGF metrics as alternative indicators of ATD. The advantage of using the modularity metrics IPCI and IPGF as ATD indicators is that these modularity metrics can be automatically calculated based on source code (i.e., the update-to-date and accurate structure data of a software system), while ANMCC should be calculated based on commit records that are not always available, and ANMCC calculation is hard to be performed automatically. Moreover, the modularity metric IPCI is more strongly correlated with ANMCC than IPGF, which means that IPCI is a more accurate substitute ATD indicator to ANMCC than IPGF.

Based on the results and findings of this work, we plan to do further research in the following directions. First, we intend to validate the correlation between modularity metrics and ANMCC with Java projects. Second, it will be interesting to define new system-wide modularity metrics or adapt existing modularity metrics defined in other perspectives (e.g., complex networks [14]), and investigate the correlation between the metrics and ATD indicators. We expect that the new modularity metrics can improve the accuracy or take less effort of predicting ANMCC. Third, it is practically valuable to develop plugins to calculate the modularity metrics IPCI and IPGF for IDE tools (e.g., in VS2012 or Eclipse).

8. ACKNOWLEDGMENTS

This work is partially supported by AFR-Luxembourg under the contract No. 895528 and the NSFC under the grant No. 61170025. Thanks to Can Menekse for his contribution to the tool *CodeMapParser* during his internship at the University of Groningen.

9. REFERENCES

[1] Abdeen, H., Ducasse, S., and Sahraoui, H., 2011. Modularization Metrics: Assessing Package Organization in Legacy Large Object-Oriented Software. In *Proceedings of the Proceedings of the 2011 18th Working Conference on Reverse Engineering* (2011), IEEE Computer Society, 2086275, 394-398. DOI= http://dx.doi.org/10.1109/wcre.2011.55.

[2] Basili, V.R., 1992. *Software modeling and measurement: the Goal/Question/Metric paradigm.* University of Maryland at College Park.

[3] Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., Maccormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., and Zazworka, N., 2010. Managing technical debt in software-reliant systems. In *Proceedings of the Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER'10)* (Santa Fe, New Mexico, USA2010), ACM, 1882373, 47-52. DOI= http://dx.doi.org/10.1145/1882362.1882373.

[4] Cunningham, W., 1992. The WyCash portfolio management system. In *Proceedings of the Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)* (Vancouver, British Columbia, Canada1992), ACM, 157715, 29-30. DOI= http://dx.doi.org/10.1145/157709.157715.

[5] Curtis, B., Sappidi, J., and Szynkarski, A., 2012. Estimating the size, cost, and types of Technical Debt. In *Proceedings of the 3rd International Workshop on Managing Technical Debt (MTD '12)*, 49-53. DOI= http://dx.doi.org/10.1109/mtd.2012.6226000.

[6] Field, A., 2013. *Discovering Statistics using IBM SPSS Statistics*. SAGE Publications Ltd.

[7] ISO/IEC, 2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. In *ISO/IEC FDIS 25010:2011*, 1-34.

[8] Kruchten, P., Nord, R.L., and Ozkaya, I., 2012. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software 29*, 6, 18-21. DOI= http://dx.doi.org/10.1109/MS.2012.167.

[9] Kruchten, P., Nord, R.L., and Ozkaya, I., 2013. 4th International workshop on managing technical debt (MTD 2013). In *Software Engineering (ICSE), 2013 35th International Conference on*, 1535-1536. DOI= http://dx.doi.org/10.1109/ICSE.2013.6606774.

[10] Li, Z., Liang, P., and Avgeriou, P., 2014. Architectural debt management in value-oriented architecting. In *Econimics-driven software architecture* Elsevier, In press.

[11] Li, Z., Liang, P., Avgeriou, P., Guelfi, N., and Chen, Y., 2014. A systematic mapping study on technical debt and its management. In *Under submission*.

[12] Marinescu, R., 2012. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development 56*, 5, 9:1-9:13. DOI= http://dx.doi.org/10.1147/JRD.2012.2204512.

[13] Microsoft, 2002. Understanding and Using Assemblies and Namespaces in .NET. DOI= http://dx.doi.org/http://msdn.microsoft.com/en-us/library/ms973231.aspx.

[14] Newman, M.E., 2003. The structure and function of complex networks. *SIAM review 45*, 2, 167-256. DOI= http://dx.doi.org/doi:10.1137/S003614450342480.

[15] Nord, R.L., Ozkaya, I., Kruchten, P., and Gonzalez-Rojas, M., 2012. In search of a metric for managing architectural technical debt. In *Proceedings of the 10th Working IEEE/IFIP Conference on Software Architecture (WICSA '12)* IEEE Computer Society, Helsinki, Finland.

[16] Ozkaya, I., Kruchten, P., Nord, R., and Brown, N., 2011. Second international workshop on managing technical debt (MTD 2011). In *Proceedings of the Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)* (Waikiki, Honolulu, HI, USA2011), ACM, 1986051, 1212-1213. DOI= http://dx.doi.org/10.1145/1985793.1986051.

[17] Runeson, P. and Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering 14*, 2 (2009/04/01), 131-164. DOI= http://dx.doi.org/10.1007/s10664-008-9102-8.

[18] Seaman, C. and Guo, Y., 2011. Measuring and Monitoring Technical Debt. In *Advances in Computers*, M. Zelkowitz Ed. Elsevier Science, 25-45.

[19] Sethi, K., Yuanfang, C., Wong, S., Garcia, A., and Sant'anna, C., 2009. From retrospect to prospect: Assessing modularity and stability from software architecture. In *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, 269-272. DOI= http://dx.doi.org/10.1109/WICSA.2009.5290817.