Quantifying the Encapsulation of Implemented Software Architectures

Eric Bouwers*†, Arie van Deursen†, Joost Visser*‡,

*Software Improvement Group, †Delft University of Technology,‡Radboud University Nijmegen
Email: e.bouwers@sig.eu, Arie.vanDeursen@tudelft.nl , j.visser@sig.eu

***Abstract*—Applying encapsulation techniques lead to software systems in which the majority of changes are localized, which reduces maintenance and testing effort. In the evaluation of implemented software architectures, metrics can be used to provide an indication of the degree of encapsulation within a system and to serve as a basis for an informed discussion about how well-suited the system is for expected changes. Current literature shows that over 40 different architecture-level metrics are available to quantify the encapsulation, but empirical validation of these metrics against changes in a system is not available.**

In this paper we investigate twelve existing architecture metrics for their ability to quantify the encapsulation of an implemented architecture. We correlate the values of the metrics against the ratio of local change over time using the history of ten open-source systems. In the design of our experiment we ensure that the values of the existing metrics are representative for the time period which is analyzed. Our study shows that one of the suitable architecture metrics can be considered a valid indicator for the degree of encapsulation of systems. We discuss the implications of our findings both for the research into architecture-level metrics and for software architecture evaluations in industry.

I. Introduction

When applied correctly, the process of encapsulation ensures that the design decisions that are likely to change are localized [2]. In the context of software architecture, which is loosely defined as the organizational structure of a software system including components, connections, constraints, and rationale [14], the encapsulation process revolves around hiding the implementation details of specific components.

Whether the encapsulation was done effectively, i.e., to what extent changes made to the system were indeed localized to a single component, can only be determined retrospectively by examining the history of the project. However, for evaluation purposes it is desirable to use the current design or implementation of a system to reason about quality attributes of the system [9]. Focussing on the implemented architecture, so-called "late" architecture evaluations [10] should be conducted to determine which actions need to be taken to improve the quality of an architecture when, for example, the current implementation deviates substantially from the original design or when no design documents are available.

A common approach in late architecture evaluations is to use a metric that can be calculated on a given snapshot of the system, i.e., the state of a system on a given moment in time, to reason about changes that will occur subsequently. On the class level, several metrics have been proposed which have also been evaluated empirically through a comparison with historical changes [19].

However, these metrics on the class-level do not reflect the level of encapsulation on the *system-level*, i.e., whether a system is decomposed into a set of independent components which hide the details of their implementation. This is important to determine whether the current architecture is fit for the changes that need to be made. Moreover, managers and external evaluators can use this system-level information to determine which systems within their large, heterogenous portfolio can benefit the most from architectural refactorings, and thus allocate resources more effectively [17].

Unfortunately, only a few metrics for encapsulation on the level of the system exist, and for those that exist no empirical validation against historical changes has been provided [15]. As a consequence, it is unclear which of the available metrics are valid to be used as a quantification of the current level of encapsulation of an implemented architecture. The goal of this paper is to fill this gap by means of an empirical study.

In this study we focus on 12 architecture-level metrics to quantify the encapsulation on the system-level. We examine whether the values of these metrics are correlated with the success of encapsulation in 10 open-source Java systems with an average history of six years. To quantify historical encapsulation, we follow Yu et al. [26] and classify each change-set in a system as either *local* (all changes occur within one component) or *non-local* (multiple components are involved in the change). A high ratio of local change-sets shows that frequently changing parts of the system were indeed localized, thus indicating that the encapsulation was done effectively.

The results of our study show that three of the evaluated architecture metrics, those that are aimed at quantifying the extent to which components are connected to each other, are correlated with the ratio of local change-sets. Of these three, one can be seen as a valid indicator for the success of encapsulation of a system. In contrast, metrics which are purely based on the number of components or on the number of dependencies between components were not found to bear a relationship to the success of encapsulation.

In the paper, we first introduce our research question in Section II, after which architecture metrics under evaluation are discussed in Section III. The design and implementation of our study is given in Section IV and Section V, the results are presented in Section VI. In Section VII the presented results are discussed and put into context. Section VIII discussed threats to validity, after which related work is discussed in Section IX. Lastly, Section X concludes the paper.[1]

[1]Note that a preliminary version of this study has been published as chapter eight of this PhD thesis [3]

II. Problem Statement

Following the Goal Question Metric approach of Basili et al. [1] we define the *goal* of our study as evaluating existing software architecture metrics *for the purpose of* assessing their indicative power for the level of encapsulation of a software architecture. The *context* of our study is late software architecture evaluations *from the point of view* of software analysts and software quality evaluators. From this, the following research question is derived:

Which software architecture metrics can serve as indicators for the success of encapsulation of an implemented software architecture?

III. Metrics for Encapsulation

As a first step towards answering our research question we review the architecture metrics currently available in the literature. The purpose of this review is to select a set of metrics for this initial study which are a) capable of providing a system-level indication of the encapsulation of a software system, and b) can be used within a late architecture evaluation context. Before discussing the selected metrics we first introduce the model we use to reason about an implemented software architecture.

In this study, we look at the implemented software architecture from the module viewpoint [8]. We define a *system* to consist of a set of high-level *components* (e.g., top-level packages or directories) which represent coherent chunks of functionality. Each component contains one or more *modules* (e.g., source files). Within modules, a *unit* represents the smallest block of code which can be called in isolation. Each module is assigned to a single component and none of the components overlap.

Directed dependencies exist between both modules (e.g., extends or implements relations) and units (e.g., call-relations) and have an attribute cardinality which represents, for example, the number of calls between two units. Dependencies between components are calculated by lifting the dependencies from the modules/units to the component-level. Both modules and units have (code-based) metrics assigned to them, for example lines of code or McCabe complexity [20], which are aggregated from the module-/unit-level to the component-/system-level.

A. Selected Metrics

Using the overview of Koziolek [15] we identified over 40 metrics in the literature. After applying a selection process to, amongst others, verify the ability of the metrics to quantify the level of encapsulation within a system, 12 software metrics remained for this initial study, these are listed in Table I. Of these 12 metrics, nine have a clear conceptual relation to encapsulation in the sense that they quantify the chance a change can propagate from one component to another, while three metrics are included as control variables. A detailed explanation of the evaluated metrics, the complete selection procedure and the application of the procedure can be found in this Phd thesis [3, chapter 8].

In the remainder of this section we introduce the selected metrics using the example system displayed in Figure 1. This system contains three components (A,B and C), which have two dependencies (A depends on B, B depends on C). The sizes of the components are 10, 40, and 20 KLOC for components A, B, and C, divided over 100, 200, and 300 modules respectively.

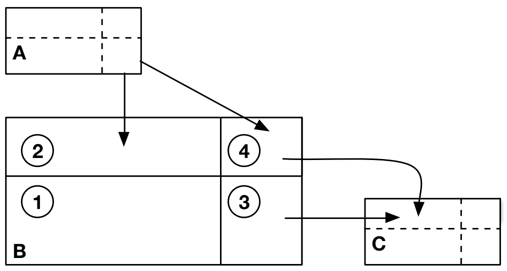

Fig. 1. Example system containing three components with four module-level dependencies, which are lifted to two component-level dependencies.

The first metric is the *Ratio of Cohesive Interactions* (RCI) introduced by Briand et al. [7]. The pessimistic variant of RCI is defined as the division of the number of known dependencies between components by the number of possible dependencies between components. For our example system this would result in a value of 2 (known dependencies) divided by 6 (possible dependencies) $= \frac{2}{6}$. Ideally, the value of RCI would be low, meaning that only a small part of all possible interactions between components is actually utilized, making it less likely that changes will propagate between components.

The second metric is introduced by Lakos [18] and is called *Cumulative Component Dependency* (CCD). It is defined as the sum of the number of components needed to test each components. In our example system the CCD of component B would be 1 (because it needs component C), and the CCD of component A would be 2 (since it needs both B and C). On a system level this results in the sum of all cumulative dependencies of all components, which in our example would be 3. Ideally, the value of CCD is low, which should mean that components are less dependent on each other.

The following two metrics, *Average CCD* (ACD) and *Normalized CCD* (NCD), are variants of the CCD. The NCD is derived by dividing the CCD by the total number of components, i.e., $\frac{3}{3} = 1$ in our example. The ACD is defined by dividing the CCD over the total number of modules in each component, leading to a value of $\frac{100+200+300}{3} = 200$ for our example system. On both cases the desired value of the metric is low, following the same reasoning for CCD.

The *Cyclic Dependency Index* (CDI) introduced by Sarkar et al. [23] quantifies the number of cyclic dependencies in the component graph. Since there are no cyclic dependencies in our example system the value is 0, but adding a dependency between component C and A would result in a CDI of 1. Again, lower values are desired to minimize the chance of changes being propagated between components.

The following three metrics are derived from our earlier work on *Dependency Profiles* [5]. In such a profile the modules in a component are categorized into four different categories as displayed in component B of our example. Category 1 is *internal* code, category 2 is *incoming* code, category 3 is *outgoing* code, and category 4 is *transit* code.

To derive a system-level value for *Internal code*, the sizes of all modules in category 1 is summed and divided by the total size of the system. This results in a value between 0 and 1 representing the percentage of code which is internal to components. Ideally, the value of this metric is high, meaning that there is less code dependent on (or dependent on by) other components, which lowers the chance of changes being propagated.

By combining the amount of code in category 2 and 4, we calculate the *Inbound Code* metric, i.e., the percentage of code which is dependent upon from other components. Similarly, combining the amount of code in categories 3 and 4 gives us *Outbound code*, which represents the percentage of code which depends upon other components. In both cases, the desired value is low, which would mean that the interface of each component is small, which in turn reduces the chance of changes being propagated.

The last non-control metric is the *Number of Binary Dependencies* (NBD), which simply counts the number of dependencies between components. In our example system this would be 2. Similar to the reasonings above the value of this metric should be low, which would indicate less dependencies and thus a lower chance of changes being propagated.

As a validation of our experiment we also take into account three control variables which quantify the way in which a system is decomposed into components. These three metrics are *Component Balance* [4], *Module Size Uniformity Index* [23] and the *Number of components* (NC). These three metrics are not aimed towards quantifying the encapsulation of the system, i.e., they do not quantify the likelihood of changes being propagated. If the experiment shows a strong correlation for any of these three metrics the design of the experiment needs to be critically reexamined.

IV. Experiment Design

The central question of this paper is which software architecture metrics can be used as an indicator for the success of encapsulation of an implemented software architecture. To answer this question, we need to determine whether the metrics listed in Table I are indicative for the degree of success of encapsulation within a system. We achieve this by performing an empirical study in which we correlate the *values of the selected metrics* with *historical data* that shows the success of encapsulation within a system in the past.

Since this type of evaluation of system-level architectural metrics has not been done before [15], we define how the success of encapsulation can be measured in Section IV-A. Next, in Section IV-B and Section IV-C we define how metrics based on a single snapshot of a system and a metric based on changes between snapshots can be compared. The procedure for correlating the different metrics is discussed in Section IV-D and augmented in Section IV-E. Lastly, Section IV-F provides a summary of the steps in the experiment.

In the design of the experiment we use the term *snapshot-based* metric to refer to metrics which are calculated on a single snapshot of a system (e.g., the number of components on a specific point in time). All metrics listed in Table I belong to this category. A metric which is calculated based on changes between snapshots of a system, for example the number of files that changes between two snapshots of a system, is referred to as a *historical* metric.

A. Measuring Historical Encapsulation

Encapsulation revolves around localizing the design decisions which are likely to change [2] (a process also known as "information hiding" [21]). In software architecture, measuring whether the changes made to a system are mainly local or spread throughout the system can be determined by looking back at the change-sets of a system.

In an ideal situation, a software system consists of highly independent components, encapsulating the implementation details of the functionality they offer. In this situation, a change to a specific functionality only concerns modules within a single component, which makes it easier to analyze and test the change made. Naturally, it is not expected that all change-sets of a system concern only a single component. However, a system with a high level of encapsulation is expected to have more localized changes compared to a system in which the level of encapsulation is low.

In this experiment, a change-set is defined as a set of modules (see Section III) that are changed together in *a unit of work* (e.g., a task, a commit or a bug-fix). Using the existing definition of Yu et al. [26] as a basis, each change-set is categorized as either *local* (all changes occur within a single component) or *non-local* (multiple components are involved in the change).

A *change-set series* is a list of consecutive change-sets representing all changes made to a system over a period of time. A series of change-sets can contain change-sets concerning different bug-fixes, it is not necessary for a change-set series to contain only change-sets that belong together. Our key-measure of interest is the ratio of change-sets in a series that is local: if this ratio is closer to one it means that more change-set are localized, which indicates a system with better encapsulation.

More formally, let $S = \langle M, C \rangle$ be a system, consisting of a set of modules M and a set of components C. Each module is assigned to a component and none of the components overlap. For each module $m \in M$ the containing component is obtained through a function: $component : M \rightarrow C$.

A change-set $cs = \{m_1 \ldots m_n\}$ is a set of modules that have been changed together. For a change-set series $CS_s = (cs_1, cs_2, \ldots, cs_m)$ we can determine for each change-set whether it is local by counting the number of components touched in this change-set, i.e., a change-set is local if and only if:

$$isLocal(cs) \Leftrightarrow |\{c | m \in cs \wedge c = component(m)\}| = 1$$

Given this property, the *ratio of local change* can be calculated by a division of the number of local change-sets by the total number of change-sets in a series:

$$ratioOfLocalChange(CS_s) = \frac{|\{cs | cs \in CS \wedge isLocal(cs)\}|}{|CS|}$$

In our experiment, we consider a change-set series with a high ratio of local change-sets to represent a high degree

TABLE I. ARCHITECTURE-LEVEL METRICS SUITABLE FOR USE IN SOFTWARE ARCHITECTURE EVALUATIONS

Name	Abbr.	Src.	Description	Desired	Control
Ratio of Cohesive Interactions	RCI	[7]	Division of known interactions by possible interactions	low	
Cumulative Component Dependency	CCD	[18]	Sum of outgoing dependencies of components	low	
Average CCD	ACD	[18]	CCD divided by number of modules	low	
Normalized CCD	NCD	[18]	CCD divided by the number of components	low	
Cyclic Dependency Index	CDI	[23]	Normalized number of cycles in the component graph.	low	
Inbound code	IBC	[5]	Percentage of code which is dependent upon from other components	low	
Outbound code	OBC	[5]	Percentage of code which depends on code from other components	low	
Internal code	IC	[5]	Percentage of code which is internal to a component	high	
Number of Binary Dependencies	NBD		The number of binary dependencies within a dependency graph	low	
Component Balance	CB	[4]	Combination of number of components and their relative sizes	high	x
Module Size Uniformity Index	MSUI	[23]	Normalized standard deviation of the size of the components	low	x
Number of components	NC		Counts the number of components in a dependency graph	low	x

of success in the encapsulation of the system. It is possible to split up a change-set series into multiple series to obtain insight into the success of encapsulation for a certain period of time. However, to obtain an accurate representation of the success of encapsulation the number of change-sets in a change-set series must be large enough to calculate a meaningful ratio. Therefore, the use of longer change-set series (e.g., covering a longer period of time) is advised.

B. Snapshot-based versus Historical

To compare a snapshot-based metric (a metric calculated on a specific time such as the number of components of a system) against a historical metric (a metric calculated on a change-set series such as the ratio of local change) two input parameters need to be defined: 1) the exact moment of the snapshot for the snapshot-based metric and 2) the change-set series for the historical metric. To increase the accuracy of the calculation of the historical metric, the change-set series should be as long as possible. On the other hand, the value of the snapshot-based metric needs to be representative for the chosen change-set series, e.g., it should be possible to link each change-set in the series to the value of the snapshot-based metric.

We obtain this balance by calculating the historical metric using a change-set series for which the snapshot-based metric is stable. To illustrate, consider the situation as shown in Figure 2, which shows the possible behavior of a snapshot-based metric given a series of change-sets. By instantiating this hypothetical graph with, for example, the number of components of a system, we can see that this number is stable for some periods, but also changes over time. This means that we cannot use the complete change-set series $(cs_0, \ldots, c_7)$ to calculate a historical metric since there is no single snapshot-based metric value we can compare against. However, the value of a historical metric based on the two change-set series (cs_0, cs_1, cs_2, cs_3) and (cs_4, cs_5, cs_6) can be meaningfully compared against the values of the snapshot-based metric (respectively 3 and 4).

This approach deviates from the commonly used design (see for example the experiments described in [26], [22]) in which a recent snapshot of the system is chosen and the historical metric is calculated based on the *entire* history of a system. The implicit assumption made in these experiments is that the value for the snapshot-based metric (calculated on the specific snapshot) is relevant for all changes throughout the history of the system. This assumption may not be valid in all situations, or should at least be verified to ensure that the comparison between these two types of metrics is meaningful.

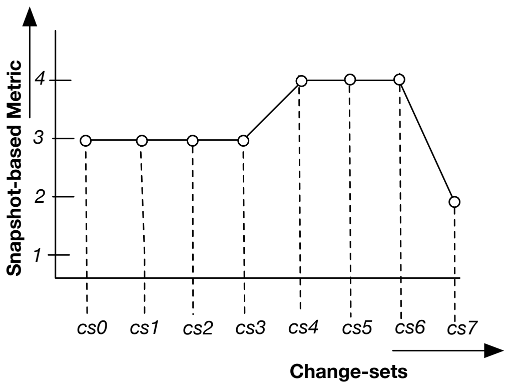

Fig. 2. The value of a snapshot-based metric over time determines the change-set series on which the historical metric should be calculated.

C. Metric Stability

One of the parameters to be instantiated is the definition of when a metric has changed significantly. For some metrics this definition is straight-forward, e.g., any change in the number of components is normally considered to be significant from an architectural point of view. However, for metrics defined on a more continuous scale, such as for example RCI, the definition is less straight-forward.

The definition of when a metric changes has an impact on the conclusions that can be drawn from the data and the length of the change-set series for a metric. In general, the definition of which change in the value of a metric is significant is most-likely dependent on both the expected variance in the metric value and the context in which it is used. For example, the number of components is not expected to change in a maintenance setting, while during the early stages of development this number fluctuates heavily. In the implementation of the experiment, a definition of "stable" related to the context of our goal is defined in Section V-A.

Note that different metrics are sensitive to different properties of a system, and that each metric is expected to change only of this specific property is changed in the implementation (when, for example, a component or a cross-component dependency is added or removed). Therefore, stable periods are not expected to be equal for all metrics.

D. Statistical Analysis

The aim of the experiment is to see whether the architecture metrics listed in Table I are correlated with the ratio of local change. To this end, we first define a null hypothesis for each of the twelve metrics that the desired value for the metric (e.g., a low number for the number of components or a high percentage of internal code) is not associated with a high (or low) ratio of local change.

To determine whether a null hypothesis can be rejected we perform a correlation test between the values of the snapshot-based metric and the ratio of local change. The specific correlation test used here is the Spearman rank correlation because we cannot assume a normal distribution in any of the metrics. Furthermore, because the hypotheses are directional a one-tailed test is performed. When the correlation test shows a moderate to strong correlation, the null-hypothesis can be rejected, meaning that for a specific snapshot-based metric the values are correlated with the ratio of local change.

Following Hopkins [13], we consider a significant correlation higher than 0.3 (or lower than -0.3) to indicate a moderate correlation, while a significant correlation score higher than 0.5 (or lower than -0.5) indicates a strong correlation. For a correlation to be significant the p-value of the test needs to be below 0.01.

In this set-up, multiple hypotheses are tested using the same dataset. In this case a Bonferroni correction [13] prevents the finding of a correlation by chance simply because one performs many different tests. The correction that needs to take place is the multiplication of the p-value of each individual test by the number of tests performed. If the resulting p-value is still below 0.01 the result of the test can be considered significant. Note that the use of the Bonferroni correction might lead to false negatives, i.e., not rejecting a null hypothesis even though there is a correlation. Our approach here is to be conservative by applying the correction. (The impact of this choice is discussed in Section VIII-3.)

E. Preventing Project Bias

In this setup, data-points from several projects are combined into a single data-set to derive a correlation, instead of calculating the correlation on a per project basis. This is primarily done because we are interested in a general trend across projects. Additionally, architecture-level metrics are expected to remain stable for long periods of time, resulting in just a few data-points per project, which makes it hard to derive statistically significant results.

However, it is possible that a single system contributes a disproportionately large number of data-points to the sample used for correlation. If this is the case, a significant correlation might be found just because the correlation occurs within a single system. To determine the impact of this issue we perform a multiple linear regression analysis for all significant correlations.

The input of such an analysis is a linear model in which the dependent variable (i.e., the ratio of local change) is explained by a set of independent variables (i.e., the value of one of the snapshot-based metrics) plus one dummy variable per project. To determine which independent variables are significant we apply a stepwise selection using a backward elimination model [13]. This process iterates over the input model and eliminates the least significant independent variables from the model until all independent variables are significant. If the resulting model contains the snapshot-based metric as the most significant factor (expressed by the R-squared value of the individual factor) we can conclude that the influence of the specific projects is negligible.

F. Summary

To summarize, the procedure for testing the correlation between each snapshot-based architecture metric and the historical ratio of localized change becomes:

Step 1:	Define when a metric is considered stable
Step 2:	Determine the change-set series for which the snapshot-based metric is stable on a per project/per metric basis
Step 3:	Calculate the historical metric per change-set series
Step 4:	Per metric, calculate the correlation between the snapshot-based metric value and the historical metric using data from all projects
Step 5:	Verify the influence of individual projects on significant correlations

V. Experiment Implementation

The metrics investigated in this experiment are architecture metrics, each quantifying a different property of the software architecture. Thus, we expect the metrics to be stable as long as the properties of architecture under investigation of a system is stable. The context of this experiment is that of software analysts and software quality evaluators. In such a context it is common practice to place a system within a bin according to its metrics in terms of different categories, e.g., 4 is a low number of components, 8 is a moderate number of components and 20 is a large number of components. A change in metric is interesting (and thus significant) when the value of the metric shifts from one bin to another.

However, for most of the snapshot-based metrics there is no intuitive value which indicates when a systems should be placed in the "low", "moderate" or "large" category. Therefore, we take a pragmatic approach by defining a bin-size of 1. For example, if the number of components for a system on (t_1,t_2,t_3,t_4,t_5) is $(4,4,5,6,6)$, the stable periods are considered to be $t_1 - t_2$ and $t_4 - t_5$.

Similarly, for percentage and ratio metrics a change is considered significant when the value is placed in a different bin, with an absolute bin-size of 0.01. For example, when the values of the metric on snapshots (t_1, t_2, t_3) are (0.243, 0.249, 0.251), the snapshots t_1 and t_2 are considered to be equal, while snapshot t_3 is considered to be a significant change. The implications of choosing this bin-size are discussed in Section VIII-2.

A. Stable Period Identification

To determine the time-periods for which a snapshot-based metric is stable, the value of the metric must be calculated for different snapshots of the system. To obtain the most accurate result a snapshot should be taken after each change-set. However, given the large number of change-sets this approach requires an enormous amount of calculation effort. To compromise between precision and calculation effort we use a sampling approach.

Snapshots of the system are extracted on regular intervals, i.e., on every first day of the month, and all change-sets between snapshots for which the snapshot-based metrics are stable are grouped together into a single change-set series. If the value of the snapshot-based metric changes significantly (as defined above) between snapshots t_n and t_{n+1} this period is called a *transition period*. All change-sets up until snapshot t_n are grouped into a single change-set series, while all change-sets within the transition period, i.e., between t_n and t_{n+1}, are discarded. The effects of this choice are reflected upon in Section VIII-2.

For a rapidly changing metric the effect may be that all data-points are discarded, simply because the metric changes significantly in between all the snapshots. An example of this would be the metric of Lines of Code, which is expected to change often. If this is the case we should observe a low number of short stable periods for a metric, which would call for taking a shorter time-interval.

A change in the value of the snapshot-based metric indicates a change in the architecture of the system. Because our study only investigates stable periods the study is focussed on determining the *effect* of these architectural changes, instead of the nature of the architecture changes themselves.

In this experiment we take one snapshot for each month that the system is active, i.e., changes are being made to the code. The snapshots are obtained from the source-code repository on the first day of each month. A more fine-grained interval (for example every week) might provide more accurate results, but since architecture metrics are not expected to change frequently a monthly interval is expected to be sufficient. The consequences of the decision for a sampling approach and the chosen sample-size are discussed in Section VIII-2.

B. Subject systems

We used the following guidelines to determine the set of subject systems:

1) *Considerable length of development*: At least a year's worth of data needs to be available in order to provide a representative number of change-sets.
2) *Subversion repository*: This source-code repository system facilitates easy extraction of individual change-sets by assuming that each commit is a change-set. In addition, this source-code repository exists long enough to enable the extraction of long histories of native commits.
3) *Written in Java*: Although the metrics are technology independent we have restricted ourselves to the Java technology because tool-support for calculating metrics for Java is widely available.

While choosing the systems we ensured that the set contains a mix of both library projects as well as complete applications. Table II lists the names of the systems used together with the overall start date and end date considered. The start date has been chosen based on the availability in the repository, the end date is either the last date in the repository or the date on which the experiment has been executed. The last two columns show the size of the subject system on respectively the start date and the end date to show that the systems have indeed changed over time.

TABLE II. SUBJECT SYSTEMS USED IN THE STUDY

	Period		**Size (KLOC)**	
Name	**Start**	**End**	**Start**	**End**
Ant	2000-02	2011-05	3	97
Argouml	2008-03	2011-07	113	108
Beehive	2004-08	2008-10	45	86
Crawljax	2010-01	2011-07	6	7
Findbugs	2003-04	2011-07	7	97
Jasperreports	2004-01	2011-08	28	171
Jedit	2001-10	2011-08	35	79
Jhotdraw	2001-03	2005-05	8	20
Lucene	2001-10	2011-08	6	67
Struts2	2006-06	2011-07	25	22

C. Architectural Model Instantiation for Java

To calculate the different metrics, we need to define components for each of the subject systems. To accurately model whether a developer needed to change a single or multiple components we use the development view-point [16] on the implemented architecture of the systems. Based on the positive results in our earlier approaches [4], [5] we define the components of a system to be the top-level packages of a system (e.g., `foo.bar.baz`, `foo.bar.zap`, etc).

Direct call relations between modules are considered to be a dependency. In line with the mechanisms of hiding implementation details in Java calls to an interface or an super-class create a dependency to the interface or the declared type, but not to classes implementing the interface or classes overriding the called method. The volume of source code modules is quantified by Lines of Code, i.e., the sum of all lines which contain non-empty, non-comment characters.

All metrics are calculated on relevant source-code modules using the Software Analysis Toolkit of the Software Improvement Group (SIG).[2] In this experiment, a module is considered to be relevant if it is production code which resides in the main source-tree of the system. Code written for testing or demo purposes is considered to be out of scope for this experiment (and therefore not included in the numbers of Table II).

VI. Experiment Results

The raw data of the experiment is available in an on-line experiment package located at:
http://www.sig.eu/en/QuantifyingEncapSA
This package contains:

D1: The descriptions of the top-level components and the scope used for each project

D2: The data-sets containing the change-sets with relevant modules used as an input to calculate the ratio of local change for a given period

D3: The data-sets listing the values of the snapshot-based metric for each month in which changes to the system have been made used to determine the stable periods

D4: The result of combining data-set D2 and D3 using a bin-size of 1 and 0.01, respectively

[2]http://www.sig.eu

TABLE III. DESCRIPTIVE STATISTICS FOR THE STABLE PERIODS PER SNAPSHOT-BASED METRIC

		Months				change-sets series length					Ratio of local change		
Metric	periods	Min	Med.	Max	covered	Min	Med.	Max	total	> 10	Min	Median	Max
RCI	94	1	4.0	38	80.9 %	3	113.0	968	17760	93.6 %	0.00	0.84	1
CCD	71	1	6.0	40	85.9 %	3	222.0	1178	19011	97.2 %	0.37	0.84	1
ACD	111	1	3.0	38	75.6 %	1	92.0	954	16564	91.9 %	0.00	0.85	1
NCD	74	1	4.5	40	83.6 %	3	192.5	1174	17922	95.9 %	0.37	0.84	1
CDI	65	1	6.0	50	88.3 %	1	224.0	2334	20526	95.4 %	0.00	0.84	1
IBC	122	1	3.0	35	68.1 %	3	67.5	715	13811	95.9 %	0.24	0.86	1
OBC	111	1	3.0	42	71.8 %	3	68.0	1337	15346	94.6 %	0.25	0.86	1
IC	119	1	2.0	41	71.2 %	2	50.0	1257	14759	91.6 %	0.16	0.86	1
NBD	108	1	3.0	38	75.8 %	3	88.5	846	15436	94.4 %	0.00	0.84	1
CB	82	1	3.0	77	80.6 %	3	76.5	5147	19345	91.5 %	0.35	0.86	1
MSUI	99	1	3.0	35	77.1 %	1	91.0	1176	18028	93.9 %	0.36	0.84	1
NC	59	1	6.0	53	90.8 %	7	262.0	1805	21428	96.6 %	0.18	0.83	1

A. Stable Periods

The first step in the experiment is to determine the stable periods for each of the twelve snapshot-based metrics. Descriptive statistics of the stable periods per metric illustrating four important characteristics of the data-set are shown in Table III.

First of all, in the second column of Table III the number of stable periods per metric is shown. Because this number exceeds the number of projects (10) in all cases we observe that the metrics are changing over time, which indicates that our definition of stability for the metrics is not too lenient.

Secondly, descriptive statistics of the number of months per stable period are shown in columns $3-5$ of Table III. As discussed in Section V-A it is desirable for a snapshot-based metric to remain stable for a considerable period of time to enable the definition of corrective actions. We observe that the median number of months in a stable period variates between two and six months, with higher values up to three to six years. Even though on the low end there exists stable periods that last only a single month, such a short time-frame is still sufficient to define corrective actions.

Thirdly, the sixth column of Table III shows the percentage of development time which is covered by the months in all stable periods. We observe that this percentage is at least 65% for all metrics, i.e., more than half of the total development time of the systems is covered by stable periods. Thus, the metrics are stable enough to be used in the context of our experiment.

Lastly, columns $7-11$ of Table III show descriptive statistics for the length of the change-set series based on the stable periods. As discussed in Section IV-B it is desirable to have longer change-set series to ensure an accurate representation of the ratio of local change. However, Table III shows that there are change-set series containing only a single change-set, which means that the ratio of local change will either be one or zero. When many change-sets series contain only a few change-sets the accuracy of the ratio of local change could be considered inadequate. However, as column 11 shows that for all metrics at least 91% of the change-sets series contain more than ten change-sets (up to over 5000 change-sets), indicating that these change-sets series are sufficiently accurate and therefore all of the series can be used in the current experiment.

B. Ratio of local change

For each of the snapshot-based metrics the ratio of local change is calculated based on the stable periods, columns $12-14$ of Table III shows descriptive statistics of the result of this calculation. We observe that all metrics show considerable variation in the ratio of local change. The central tendency of the ratio of local change appears to be close to 0.85 for all metrics. This indicates that it is common to make more local than non-local changes during periods in which a snapshot-based metric is stable, which is inline with the expectations that design decisions that change often are indeed encapsulated.

TABLE IV. CORRELATION VALUES BETWEEN EACH SNAPSHOT-BASED METRIC AND THE RATIO OF LOCAL CHANGE

Metric	Correlation	Corrected p-value	p-value
RCI	0.16	11.3	0.94
CCD	-0.27	0.13	0.01
ACD	-0.26	0.04	< 0.01
NCD	-0.19	0.59	0.05
CDI	0.32	11.94	1.00
IBC	**-0.30**	< 0.01	< 0.01
OBC	**-0.31**	< 0.01	< 0.01
IC	**0.47**	< 0.01	< 0.01
NBD	-0.22	0.14	0.01
CB	0.29	0.05	< 0.01
MSUI	-0.08	2.42	0.20
NC	-0.26	0.27	0.02

C. Correlation values

Using the values of the snapshot-based metrics and the ratio of local change we calculate the Spearman rank correlation between the two samples. Table IV shows the results of the tests with both the corrected and the original p-values.

As can be seen from the results, many of the correlation tests do not result in a significant correlation. This result is expected for the control variables MSUI, CB and NC, but for the other metrics a lack of significance is unexpected. One reason for this result could be that the number of data-points in the sample for a metric is not large enough to detect correlation. However, looking at the size of the samples as displayed in the second column of Table III this is not likely. Moreover, using a power t-test to determine the required sample size needed to find correlation shows that all samples contain enough data-points [13].

For IBC, OBC and the IC metric the result of the correlation tests is significant even after applying the correction. For these three metrics we performed a multivariate regression analysis to determine whether any of the projects have a significant influence on the found correlation (see Section IV-E for details about this approach). In all three cases we determined that the models are suitable for the data and that the value of the snapshot-based metric is by far the most significant factor in each model. In other words, the snapshot-based metric explains most of the variation in the ratio of local change. The detailed models can be found in [3, Chapter 8].

VII. Discussion

The results of the experiment shows that there is not enough evidence to reject the null hypothesis that the value of the metric is related to a higher/lower historical ratio of local change for the metrics RCI, CCD, ACD, NCD, NBD, MSUI, CDI, CB, and NC.

For IBC, OBC and IC the found correlation is moderate, thus the null hypothesis may be rejected. In other words, the results of the experiment shows that *the percentage of inbound code*, *the percentage of outbound code* and *the percentage of internal code* are correlated with *the historical ratio of local change*.

This correlation can be explained by the fact that these metrics quantify the percentage of code in the "requires" interface (OBC), the "provided" interface (IBC) and non-interface code (IC) of the components of a system. The larger the interfaces of a component, the more likely it is that changes in one place will propagate to other components. From this point of view, these metrics are measuring the extent, e.g., the "width", of the connection between components instead of the strength of these connections.

Despite the relationship between the metrics, e.g., a larger "requires" interface automatically leads to a lower percentage of non-interface code, we observe that the quantification of all non-interface code provides a stronger correlation than a quantification of either the "required" or the "provided" interfaces within a system. Moreover, the percentage of internal code is more closely related to the notion of encapsulation as defined in Section IV-A. Based on these observations the answer to our research question is:

> *The percentage of internal code can serve as an indicator for the success of encapsulation of an implemented software architecture.*

The results show that for nine metrics there is insufficient evidence to conclude that these metrics have indicative power for the level of encapsulation of a software architecture. For the three control variables (MSUI, CB and NC), this result can be attributed to a difference in goal. These metrics are primarily designed to quantify the analyzability of a system instead of the encapsulation.

The other metrics for which no significant correlations were found (i.e., RCI, CCD, ACD, NCD, CDI and NBD) are all based on a graph view (boxes and arrows) of the software architecture. Possibly, the inability of these metrics to measure encapsulation derives from the over-simplification inherent in such a view. More specifically, even though these metrics capture the strength of the dependencies between components, we suspect that they are not able to properly quantify the extent of the dependency between components.

A. Generalization

The current implementation of the experiment limits the generalizability of the results to open-source, Object-Oriented systems written in Java. In our previous work [5] we already investigated the behavior of the percentage of inbound, outbound and internal code on different technologies and found little variance between technologies, but replication of our experiment using systems with other characteristics (i.e., non object-oriented systems, industry systems) is needed to determine the exact situations in which the metrics are usable. Because the design of the experiment as described in Section IV does not impose any limitations on the characteristics of the systems we believe that this can be done with relatively little effort.

Furthermore, the fact that we only examined the stable periods of these systems means that the indicative power of the metrics cannot be ensured while a system is undergoing large refactorings on the level of the architecture. We do not consider this a problem, since the snapshot-based metrics aim to quantify characteristics of the architecture of a system and are therefore expected to remain stable for longer periods. This is supported by the data in Table III which shows that the metrics are stable for an average period of at least two months, and are stable for more than 60% of the time a system is under development.

B. Implications for Architecture Evaluations

The implication of these results for late architecture evaluations is that the percentage of internal code can be used to reason about the level of encapsulation within a system. We envision that a low percentage of internal code could be a reason to steer the refactoring of a code base to internalize modules within components.

Based on the findings in this report, the Software Improvement Group (SIG, a consultancy firm specialized in the analysis of the quality of software systems) has decided to include the percentage of internal code in its suite of metrics used to conduct (repeated) architectural evaluations. This gave us the opportunity to observe experts in the fields of architecture evaluations while they were using this percentage in the evaluation and monitoring of over 50 industrial systems implemented in a wide variety of technologies (including Java, C#, Cobol, and legacy 4GL languages). Based on these observations, and on semi-structured interviews with 11 experts, we were able to get a first insight into those situations in which the percentage is considered useful.

In particular, this evaluation showed that the percentage of internal code, combined with the percentages of outbound and inbound code, are used for targeted improvements, provide a basis for useful discussions, and can serve as a communication device between developers and project management. The details about the design and results of this study are described by Bouwers et al. [6].

C. Metric Stability

As can be seen in Table III, the metrics measured on the level of the architecture of a system have the tendency to be stable for a period between two and six months. The implication of this finding is that the assumption that the value of a snapshot-based metric is representative for all changes that occurred during the entire history of a system is not correct. If this is the case on the system-level, this assumption must also be verified when these types of experiments are performed on the level of modules or units. An alternative solution is to explicitly encode these assumptions into the design of the experiment, as we have done in Section IV.

VIII. Threats to Validity

Following the guidelines of Wohlin et al. [24] the threats to the validity of the results of the experiment are categorized in four categories addressing construct, internal, external and conclusion validity. Because the generalization of the results (external validity) has already been addressed in Section VII-A, this category of threats is not discussed in detail in this section.

1) Construct validity: The basis for our experiment is the assumption that the ratio of local change accurately models the concept of encapsulation. We believe that this is the case because the relation between encapsulation and the localization of change has been made explicit by, amongst others, Booch et al. [2]. In addition, "encapsulate what changes" is a well known and widely recognized design principle [11].

A second question regarding construct validity is whether the top-level packages of a software system can be used as the architectural components of a software system. Even though we did not perform an explicit validation of the component-structure with the developers of the systems in our experiment, manual inspection of the naming of the top-level packages suggests they comprise valid chunks of functionality.

A last question is whether the assumption that a commit into the Subversion source-code repository of the systems is a coherent unit of work is valid. This assumption might not hold for developers who have a particular style of committing code, for example always committing several fixes at once or committing changes made to each component in isolation. Although this effect might exist we believe that this threat is countered by taking into account several projects, and thus different developers with a different style of committing.

2) Internal validity: As discussed before, there might be confounding factors which explain the correlation between the snapshot based metric values and the ratio of local change. Two of these confounding factors, the influence of specific projects on the significant correlations and the influence of the size of the system in the metrics, have already been addressed in the design and results of the study.

A second confounding factor is the choice for monthly snapshots to determine stable periods for the snapshot-based metrics. Taking different periods can result in different stable time-periods, which can influence the variance of the ratio of local change. As can be determined from the data in Table III, the median number of months which are taken into account per stable period is two to six months, covering over 60% of the development time. Thus, a one month period between snapshots already covers a considerable portion of the development of the system, using a shorter period of time between snapshots does not seem warranted.

A related issue is that the value of a snapshot-based metric could fluctuate significantly between two snapshots of the system, but is still considered to be equal because the value has not changed significantly on the first of the month. Given the average length of the stable-periods this situation does not occur often enough to influence the results significantly.

A third factor is the choice to consider a percentage stable as long as the value stays within the same bin with an absolute bin-size of 0.01. As discussed before, taking different bin-size can lead a more (or less) variance in the snapshot based metric, leading to more (or less) co-variance with the ratio of local change. The median length of the periods, the number of change-sets per period and the variation in the metrics as shown in Table III do not indicate that the absolute bin-size is too small or too large for any of the metrics.

Moreover, we believe that determination of the optimal thresholds per snapshot based metric is a new research topic in its own right. Note that in our situation the pragmatic choice of bin-size can only cause false negatives, e.g., using a different bin-size might lead to finding significant correlations where there is none with the current bin-size. In contrast, changing the bin-size does not invalidate the correlations found with the current bin-size.

3) Conclusion validity: The final question is whether the conclusions drawn from the data are sound. On the one hand, the metrics for which we do reject the null-hypothesis might not be valid indicators. This would be the case when there is no rationale for the correlation between the value of the snapshot-based metric and the ratio of local change. However, as discussed in Section VII there is a logical explanation for this correlation, thus we believe that the conclusions drawn from the data are valid.

On the other hand, the metrics for which we do not reject the null-hypothesis might be valid indicators due to the use of the Bonferroni correction. Inspecting the non-corrected p-values shown in Table IV shows that without correction ACD and CB also provide significant correlations with $p < 0.01$. However, in both cases the correlation values is below 0.3 and thus considered to be low, which means that not rejecting the null hypothesis remains correct.

IX. Related Work

The change-history of a software system has previously been used, amongst others, to validate designs [27], predict source-code changes [25] and for predicting faults [12]. The majority of this work is focussed on predicting which artifacts are going to change together, while our focus is on correlating snapshot-based metrics with historical metrics. Apart from this difference in goal, the artifacts which are considered are on a different level (i.e., file versus components) or of a different nature (i.e., code versus faults).

With respect to the topic of validating snapshot-based metrics against change history of a system there is again a large body of work. As mentioned before, many class level metrics have been validated extensively, see Lu et al. [19] for an overview.

On the component level this type of validation has been done by Yu et al. [26]. In this experiment, the relationship between the external co-evolution frequency (e.g., non-local change) and several size and coupling-related metrics is investigated using the complete history of nine open-source projects.

However, we are not aware of any study which validates system-level architecture metrics against the change-history of a system. Because of this we considered the validation of system-level architecture metrics for measuring encapsulation as an unresolved problem.

X. Conclusion

The goal of this paper is to determine which existing system-level architecture metrics provide an indication of the level of encapsulation of an implemented software architecture in the context of late architecture evaluations. In this paper, we make the following contributions:

1) An experiment design in which we correlate the value of twelve architecture metrics with the ratio of local change during periods for which the metrics are representative.
2) A stability analysis on twelve metrics that shows that the variability of a metric needs to be taken into account when comparing snapshot-based metrics against metrics based on multiple snapshots of a system.
3) Strong evidence that the percentage of internal code provides an indication of the success of encapsulation of an implemented architecture.

The key implications of these results are two-fold: first, the percentage of internal code is suitable to be used in the evaluation of an implemented software architecture, a finding which is confirmed by our follow-up study [6]. Secondly, the results show that the twelve architecture metrics tend to be stable for a period of two to six months. This property needs to be taken into account in any experiment in which these specific snapshot-based metrics are correlated against metrics based on multiple snapshots of a system. More generally, the assumption that a snapshot-based metric is representative for the period of time on which an historical metric is calculated must be verified for any experiment in which these two types of metrics are correlated.

There are two main areas for future work. First of all, to continue our earlier verification efforts of the usefulness of these architecture metrics [6] and investigate the relationship between these metrics and this type of non-technical aspects of the software development process such as costs and operational risks. In addition, to determine which additional architecture metrics (i.e., metrics quantifying other quality attributes) should be used in combination with these metrics to come to a well balanced assessment.

Secondly, we envision a study aimed towards determining the best way to define the stability of software metrics. Such a study would not only improve the experiment design as proposed in this paper, it would also help in interpreting metrics currently used in the monitoring of software systems.

References

[1] V. R. Basili, G. Caldiera, and H. D. Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[2] G. Booch. *Object-oriented analysis and design with applications (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[3] E. Bouwers. *Metric-based Evaluation of Implemented Software Architectures*. PhD thesis, Delft University of Technology, 2013. Available at http://repository.tudelft.nl/view/ir/uuid:6b65c5f5-398c-4a41-8806-31c638b1891c/.

[4] E. Bouwers, J. Correia, A. van Deursen, and J. Visser. Quantifying the analyzability of software architectures. In *Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA 2011)*. IEEE Computer Society, 2011.

[5] E. Bouwers, A. van Deursen, and J. Visser. Dependency profiles for software architecture evaluations. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011)*. IEEE Computer Society, 2011.

[6] E. Bouwers, A. van Deursen, and J. Visser. Evaluating usefulness of software metrics: An industrial experience report. In *Proceedings of the 35th International Conference on Software Engineering*, 2013.

[7] L. C. Briand, S. Morasca, and V. R. Basili. Measuring and assessing maintainability at the end of high level design. In *Proceedings of the Conference on Software Maintenance (ICSM 1993)*, pages 88–97, Washington, DC, USA, 1993. IEEE Computer Society.

[8] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Boston, MA, 2003.

[9] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, 2002.

[10] L. Dobrica and E. Niemelä. A survey on software architecture analysis methods. *IEEE Trans. on Software Engineering*, 28(7):638–653, 2002.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[12] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26:653–661, July 2000.

[13] W. G. Hopkins. *A new view of statistics*. Internet Society for Sport Science, 2000.

[14] P. Kogut and P. Clements. The software architecture renaissance. *Crosstalk - The Journal of Defense Software Eng.*, 7:20–24, 1994.

[15] H. Koziolek. Sustainability evaluation of software architectures: a systematic review. In *Proceedings 7th ACM SIGSOFT International Conference Quality of Software Architectures (QoSA 11)*, QoSA-ISARCS, pages 3–12, New York, NY, USA, 2011. ACM.

[16] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[17] T. Kuipers and J. Visser. A tool-based methodology for software portfolio monitoring. In *Software Audit and Metrics, Proceedings of the 1st International Workshop on Software Audit and Metrics*, pages 118–128. INSTICC Press., 2004.

[18] J. Lakos. *Large-scale C++ software design*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[19] H. Lu, Y. Zhou, B. Xu, H. Leung, and L. Chen. The ability of object-oriented metrics to predict change-proneness: a meta-analysis. *Empirical Software Engineering*, 17:200–242, 2012.

[20] T. J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 1976.

[21] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[22] D. Romano and M. Pinzger. Using source code metrics to predict change-prone java interfaces. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011)*. IEEE Computer Society, 2011.

[23] S. Sarkar, G. M. Rama, and A. C. Kak. API-based and information-theoretic metrics for measuring the quality of software modularization. *IEEE Trans. of Software Engineering*, 33(1):14–32, 2007.

[24] C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: an Introduction*. Kluver Academic Publishers, 2000.

[25] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30:574–586, September 2004.

[26] L. Yu, A. Mishra, and S. Ramaswamy. Component co-evolution and component dependency: speculations and verifications. *IET Software*, 4(4):252–267, 2010.

[27] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 73–83, Washington, DC, USA, 2003. IEEE Computer Society.