Delft University of Technology
Software Engineering Research Group
Technical Report Series

Quantifying the Encapsulation of Implemented Software Architectures

Eric Bouwers, Arie van Deursen, Joost Visser

Report TUD-SERG-2011-031-a

TUD-SERG-2011-031-a

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Under Review

A

Quantifying the Encapsulation of Implemented Software Architectures

Eric Bouwers, Software Improvement Group and Delft University of Technology
Arie van Deursen, Delft University of Technology
Joost Visser, Software Improvement Group and Radboud University Nijmegen

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 encapsulation, but empirical validation of these metrics against changes in a system is not available.

In this paper we survey existing architecture metrics for their suitability to be used in a late software evaluation context. For 12 metrics that were found suitable we correlate the values of the metric, which are calculated on a single point in time, against the ratio of local change over time using the history of 10 open-source systems. In the design of our experiment we ensure that the value 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 these findings both for the research into architecture-level metrics and for software architecture evaluations in industry.

Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Metrics

General Terms: Design, Measurement, Verification

Additional Key Words and Phrases: Encapsulation, Architecture Evaluation, Metrics, Experiment

ACM Reference Format:
Bouwers, E., van Deursen, A., Visser, J. 2012. Quantifying the Encapsulation of Implemented Software Architectures ACM Trans. Softw. Eng. Methodol. V, N, Article A (January YYYY), 22 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

When applied correctly, the process of encapsulation ensures that the design decisions that are likely to change are localized [Booch 1994]. In the context of software architecture, which is loosely defined as the organizational structure of a software system including components, connections, constraints, and rationale [Kogut and Clements 1994], the encapsulation process revolves around hiding the implementation details of a specific component. Whether the encapsulation was done effectively, i.e., to what extent changes made to the system were indeed localized, can only be determined retrospectively by examining the history of the project.

However, for evaluation purposes it is desirable to use the current implementation or design of a system to reason about quality attributes such as the encapsulation of the system [Clements et al. 2002]. A common approach for this is to use a metric which 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 for encapsulation have been proposed, see for example the overview by Briand et al. [1998], which have also been evaluated empirically through a comparison with historic changes [Lu et al. 2012]. At the broader system level, however, few metrics for en-

Author's addresses: E. Bouwers, Software Improvement Group, Amsterdam, The Netherlands; A. van Deursen, Delft University of Technology, Delft, The Netherlands; J. Visser, Software Improvement Group, Amsterdam, the Netherlands;

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

capsulation exist, and for those that exist no empirical validation against historic changes has been provided [Koziolek 2011]. The goal of this paper is to fill this gap by means of an empirical study.

In our empirical study we take into account system-level architecture metrics capable of quantifying the encapsulation of a system during the evaluation of an implemented software architecture. These so-called "late" architecture evaluations [Dobrica and Niemelä 2002] are 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. Moreover, performing this type of architecture evaluation on a regular basis helps in identifying and preventing architecture erosion [Perry and Wolf 1992]. Additionally, this type of analysis is also employed to get a first overview of the current state of a large portfolio of systems.

In such a context, the relative ease of calculation of the metrics is important to ensure that (repeated) evaluation is economically feasible. In addition, the ability to perform root-cause analysis is a key-factor in deriving low-level corrective actions based on the high-level metrics, which in turn helps in the acceptance of the metrics with practitioners [Heitlager et al. 2007]. Moreover, metrics which can be calculated on a broad range of technologies are preferred in order to make it possible to compare systems taken from a large, heterogeneous portfolio of systems. The first step in our study is to identify a set of existing metrics that adhere to these properties, which is done by surveying the over 40 available architecture-level metrics.

In the second step of the study the remaining 12 architecture-level metrics are correlated with the success of encapsulation in 10 open-source Java systems having an average history of six years. To quantify historic encapsulation, we follow the proposal of Yu et al. [2010] and classify each change-set in a system as either *local* (all changes occur within a single component) or *non-local* (multiple components are involved in the change). A high ratio of local change-sets shows that the frequently changing parts of the system were indeed localized, indicating that the encapsulation was done effectively.

The results of our study show that three of the suitable 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 is chosen 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. The implications of this finding on both the research into architecture-level metrics as well as the use of these metrics in an architecture evaluation setting are discussed.

2. PROBLEM STATEMENT

Following the Goal Question Metric approach of Basili et al. [1994] we define the *goal* of our study to be to evaluate 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 goal the following research question is derived:

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

In order to answer our research question we first survey the currently available architecture metrics for their suitability to be used in a late software architecture evaluation context in Section 3. We then design and execute an empirical study to determine the relationship between the selected software architecture metrics and the success of encapsulation. The design and implementation of this study is given in Section 4 and Section 5, the results are presented in Section 6. In Section 7 the presented results are discussed and put into context. Section 8 discussed threats to validity, after which related work is discussed in Section 9. Lastly, Section 10 concludes the paper.

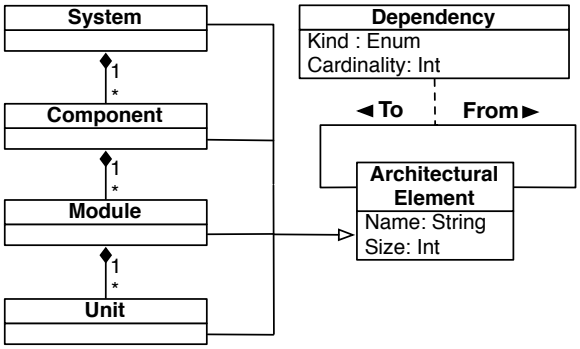

Fig. 1. An UML-diagram of the used architectural meta-model

Table I. Criteria used for the selection of architecture metrics

	The architecture metric:
C1	has the potential to measure the level of encapsulation within a system
C2	is defined at (or can be lifted to) the system level
C3	is easy to compute and implement
C4	is as independent of technology as possible
C5	allows for root-cause analysis
C6	is not influenced by the volume of the system under evaluation

3. 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 those metrics 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 selection criteria we first introduce the model we use to reason about an implemented software architecture.

3.1. Architectural Meta-Model

In this paper, we look at the software architecture from the module viewpoint [Clements et al. 2003]. The model we use is displayed in Figure 1 in the form of a UML-diagram. 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 exists between both modules (e.g., extends or implements relations) and units (e.g., call-relations) and have an attribute cardinality which represent, for example, the number of calls between two units. Dependencies between components can be calculated by lifting the dependencies from the modules/units to the component-level. Both modules and units can have (code-based) metrics assigned to them, for example lines of code or McCabe complexity [McCabe 1976], which can be aggregated from the module-/unit-level to the component-/system-level.

3.2. Metric Selection Criteria

We use the criteria as listed in Table I to identify metrics that are relevant to our experiment. The first two criteria expresses our focus on system level architecture metrics capable of quantifying the encapsulation of a system. Criteria C3, C4 and C5, which have been identified by Heitlager et al. [2007], relate to the suitability of a metric for use within a late architecture evaluation context. Lastly, criterion C6 ensures that a metric can be used to compare different snapshots of a system over time. Together, these six criteria ensure that the metrics can be used to evaluate the encapsulation of a system in a repeated manner.

To ensure criterion C1 (measuring encapsulation) we follow the definition of Booch [1994] and equate the level of encapsulation with the locality of changes. Using this definition we analytically assess each metric to determine to what extent the metric quantifies the propagation of changes from one module to another. However, metrics which do not have a (straight-forward) potential for measuring the level of encapsulation within a system are not automatically excluded from our experiment. Instead, we include these metrics as control variables in order to verify the validity of our experimental set-up.

To ensure criterion C2 (system level metric) we check whether a metric defined at the component level can also be calculated on the level of a complete system. If this is the case (for example by using the dependencies on a higher level) we include the metric in our experiment. If the metric can only be calculated on the level of a component the metric is excluded. Determining the best way to aggregate component level metrics to the system level (i.e., determining the best aggregation function for each metric) is considered to be a separate study. This criterion ensures that the metric can be used to compare different systems.

To ensure criterion C3 (that the metric is easy to compute), we adopt an architectural description making as little assumptions as possible. Following the terminology as defined in Section 3.1 this minimal description consists of a) the source-code modules of a system, b) the dependencies between these modules (e.g., call relations), and c) a mapping of the source-code modules to high-level components. For most programming languages, tool support exists to identify the required architectural elements and their dependencies, thus making metrics based on these inputs easy to implement which reduces the initial investment for performing evaluations.

To ensure criterion C4 (technology independence) we restrict the module-level metrics to those that can be applied to any technology. Consequently, metrics that are specific to, e.g., object-oriented systems (such as the depth-of-inheritance tree) are not taken into account in the current experiment. This criterion enables the evaluation of a diverse application portfolio.

To ensure criterion C5 (allow for root-cause analysis) we require that it is possible to identify architectural elements that cause an undesirable metric result. In other words, when the final value of the metric is undesired (i.e., either too high or too low) it should be possible to determine which source-code modules are the cause of the value (and are thus candidates for further inspection). This criterion ensures that the metrics can provide a basis to determine which actions need to be taken to improve the system if needed.

Heitlager et al. [2007] also identified a fourth criterion, that a metric is simple to explain to ensure that technical decision makers can understand them. Because we have no objective way to determine whether a metric is simple to explain this criterion is not used in the selection process.

To ensure criterion C6 we do not include metrics that are influenced by the volume of the system under review in a straight-forward manner (e.g., the absolute number of calls between components) in the current experiment. This criterion ensures that the selected metrics can be used to compare systems with varying sizes, thus allowing the comparison of different systems across a portfolio or the comparison of different versions of the same systems over time. Determining the best way to normalize this type of metrics is outside the scope of the current research. Note that the number of components is independent of the size of a system. An empirical study by Bouwers et al. [2011a] established that the number of top level components of any system (irrespective of the volume) revolves around 7.

3.3. Metric Selection

Using the overview of Koziolek [2011] as a basis we identified over 40 metrics in the literature, of which we provide a full account in Appendix A. In the remainder of this section we discuss the most important ones, including the 12 which are included our in experiment. Of these 12, nine have a clear conceptual relation to encapsulation while three metrics are included as control variables.

Briand et al. [1993] define three different coupling and cohesion metrics. First, three definitions of the *ratio of cohesive interactions* (RCI) are given. As input for these ratios the concepts of “known”, “possible” and “unknown” interactions need to be instantiated. In order to make it possible to cal-

culate these metrics on a large scale we consider all dependencies between source-code modules to be "known" interactions. Furthermore, following the definition of the pessimistic variant of the metric all "unknown" interactions are treated as if they are known not to be interactions. Then, we define the RCI to be the division of the number of "known" dependencies between components by the number of "possible" dependencies between components.

Intuitively, when this ratio is low (e.g., only a limited percentage of all possible dependencies is defined) a change is more likely to remain local since there is only a limited number of dependencies over which it can propagate, which might indicate a higher success of encapsulation. This metric adheres to all criteria and is therefore included in our experiment.

The other two metrics defined by Briand et al. [1993] are the "Import Coupling" and the "Export Coupling" of a component. Because these metrics are defined at the level of components and cannot be calculated on the system level we do not include them in our experiment.

Lakos [1996] defines a metric called *Cumulative Component Dependency* (CCD), which is the sum of the number of components needed to test each component. On the system-level, this is equivalent to the sum of all outgoing dependencies of all components. Two variations are defined as well, the *Average Cumulative Component Dependency* (ACD) and the *Normalized Cumulative Component Dependency* (NCD). To arrive at the NCD, the CCD is divided by the number of components of the system. For ACD, the CCD is divided by the total number of modules in the system. Higher values of these metrics indicate that components are more dependent on each other, which increases the likelihood for changes to propagate. Thus a lower value for these metrics could indicate a more successful encapsulation. All three metrics adhere to all criteria and are therefore included in our experiment.

Mancoridis et al. [1998] propose a metric aimed at expressing the proper balance between coupling and cohesion in a modularization. Their metric is intended for steering a (search-based) clustering algorithm. Unfortunately, their metric takes into account an average coupling value. By using an average value, important information about outliers is lost during the calculation of the metric, which makes it hard to perform a root-cause analysis. Therefore, we do not include clustering metrics like these in our experiment.

Allen and Khoshgoftaar [1999] use information theory as a foundation for measuring coupling. Using a graph-representation of a system, the average information per node (i.e., the *entropy*) is determined and the total amount of information in the structure of the graph is calculating by a summation. The use of "excess entropy" (the average information in relationships) in the calculation of the metric makes it hard to perform root-cause analysis. The same reasoning applies to the metrics introduced by Anan et al. [2009]. Because of this property we do not include these metrics in the current experiment.

Martin [2003] defines several metrics based on the concepts of *Afferent Coupling*, which is the number of different components depending on modules in a certain component, and *Efferent Coupling*, which is the number of components the modules of a certain component depend upon. Both of these metrics are defined at the level of components and are therefore not selected for our experiment. However, the metrics can be lifted to the system level. The result is that both metrics become equal to the number of dependencies between components, a metric which is considered in our experiment under the name Number of Binary Dependencies (NBD) as discussed at the end of this section.

Sant'Anna et al. [2007] defines a set of concern-driven metrics. Part of the input of these metrics is a mapping between the source code and functional concerns (e.g., "GUI" or "Persistence") implemented in the system, which is not available in our operationalization of criterion C3. Moreover, the metrics are either defined at the level of a concern or at the level of a component, thus we exclude these metrics from the current experiment.

Sarkar et al. [2007] define an extensive set of architecture-level metrics. To start, they define the *Module Size Uniformity Index* which measures the distribution of the overall size of the system over the components. This metric adheres to the three criteria for metrics we use, but its primary goal is to quantify the analyzability of a system; not encapsulation. Nonetheless, we include the metric

in our experiment as a control variable. The accompanying metric, *Module Size Boundness Index* relies on an (unspecified) upper limit for the size of components. Because this upper limit is not available the metric cannot be calculated. Additionally, the aggregation to system-level requires an additional (user-specified) limit. Determining the best values for these parameters is considered to be a separate study, therefore we exclude the metric from our current experiment.

Another metric which is defined by Sarkar et al. [2007] is the *Cyclic Dependency Index*, a metric which quantifies the number of cycles in the component dependency graph. Similar to the other dependency based metrics, a higher value of this metric potentially leads to a higher degree of propagation of changes and thus indicates a low success of encapsulation. Because this metric adheres to all criteria it is included in the current experiment. Lastly, Sarkar et al's *Normalized Testability Dependency Metric* is equivalent to Briand's RCI, and included under that name in our experiment.

Other metrics defined by Sarkar et al. [2007] such as the *API Function Usage Index* or the *Non-API Function Closedness* rely on the definition of a formal API of the components of a system. Other metrics such as the *The Layer Organization Index* or the *Concept Domination Metric* rely on a mapping of components to layers, or on a mapping of functional concerns to source-code. As stated in the operationalization of criterion C3 this information is not available due to the manual effort needed to create and maintain this information. Therefore we exclude these metrics from our experiment. In addition, Sarkar et al. extend their list of architecture metrics towards object-oriented concepts in [Sarkar et al. 2008]. Because these metrics rely on paradigm-dependent concepts we do not consider these metrics in our experiment.

Sangwan et al. [2008] define a metric called *Excessive Structural Complexity* which combines low level complexity with a quantification of higher level dependencies. Unfortunately, the normalization involved in the calculation of the metric prevents a straight-forward root-cause analysis. Because of this property we do not include this metric in the current experiment.

In our previous work on architecture metrics we have defined *Component Balance* [Bouwers et al. 2011a], an architecture-level metric which combines the number of components and the relative size of the components. Because this metric is designed to quantify the analyzability of a software system rather than the encapsulation this metric is included in the current experiment as a control variable.

More recently, we have introduced the concept of *Dependency Profiles* [Bouwers et al. 2011b] aimed at quantifying the level of encapsulation within a system. This is done by categorizing all code of a system as either being internal to a component, meaning that it is not called from or depends on code outside its own component, or being part of required interface of a component (*outbound* code, representing code which calls code from other components) or the provided interface of a component (*inbound* code, representing code which is called from other components). These metrics adhere to all criteria [Bouwers et al. 2011b], thus we take into account the percentage of inbound code (IBC), the percentage of outbound code (OBC) and the percentage of internal code (IC).

Lastly, several standard architecture metrics are taken into account. As a start the number of components (NC) is considered to be a candidate to represent the size of the architecture. Similar to Component Balance this metric does not seem to target encapsulation directly but is added to the experiment as a control variable for the experiment. Furthermore, the number of dependencies between the components is also considered. Systems with a higher number of dependencies are expected to have more propagation of changes. Note that for the number of dependencies the number of binary dependencies between components (i.e., 1 if there are dependencies, 0 if there are no dependencies) or the precise number of dependencies between components (i.e., each dependency between modules is counted separately) can be taken into account. The first metric is defined as the number of binary dependencies (NBD) and included in the experiment under that name. The latter metric, however, is not included in the current experiment because this absolute number is influenced by the volume of a component.

3.4. Selection result

Table II provides a summary of the metrics suite for encapsulation that we will investigate in our experiment. The first nine directly address encapsulation, and adhere to all criteria. The last three

Table II. Architecture-level metrics suitable for use in software architecture evaluations

Name	Abbr.	Src.	Description	Desired	Control
Ratio of Cohesive Interactions	RCI	[Briand et al. 1993]	Division of known interactions by possible interactions	low	
Cumulative Component Dependency	CCD	[Lakos 1996]	Sum of outgoing dependencies of components	low	
Average CCD	ACD	[Lakos 1996]	CCD divided by number of modules	low	
Normalized CCD	NCD	[Lakos 1996]	CCD divided by the number of components	low	
Cyclic Dependency Index	CDI	[Sarkar et al. 2007]	Normalized number of cycles in the component graph.	low	
Inbound code	IBC	[Bouwers et al. 2011b]	Percentage of code which is dependent upon from other components	low	
Outbound code	OBC	[Bouwers et al. 2011b]	Percentage of code which depends on code from other components	low	
Internal code	IC	[Bouwers et al. 2011b]	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	[Bouwers et al. 2011a]	Combination of number of components and their relative sizes	high	x
Module Size Uniformity Index	MSUI	[Sarkar et al. 2007]	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

do not directly address encapsulation, but are included as control variables for our experiment. Note that the metrics which are excluded may still be considered to be suitable for specific situations, but are out of scope of the current experiment. However, we envision additional experiments in which these metrics are addressed specifically as part of our future work.

4. 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 II are indicative for the degree of success of encapsulation within a system. This is done by performing an empirical study in which historical data is used to analyze the success of encapsulation within a system in the past. We then try to correlate this success with the values of the selected metrics.

Since this type of evaluation of system-level architectural metrics has not been done before [Koziolek 2011] we first define how the success of encapsulation can be measured in Section 4.1. Next, we define how metrics based on a single snapshot of a system and a metric based on changes between snapshots can be compared in Section 4.2 and Section 4.3. The procedure for correlating the different metrics is discussed in Section 4.4 and augmented in Section 4.5. Lastly, Section 4.6 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 II 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 *historic* metric.

4.1. Measuring Historic Encapsulation

Encapsulation revolves around localizing the design decisions which are likely to change [Booch 1994] (a process also known as "information hiding" [Parnas 1972]). In the context of software architectures, 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 components which are highly independent, 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 which has a high level of encapsulation (i.e., in which the design decisions which are likely to change are localized) 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 3.1) which are changed together in *a unit of work* (e.g., a task, a commit or a bug-fix). Using the concepts of Yu et al. [2010] 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. Note that a series of change-sets does not necessarily contain change-sets which belong together, it can very well be that each change-set concerns a different bug-fix. Our key-measure of interest is the ratio of change-sets in a series that is local: the closer this ratio is to one, the better the system was encapsulated.

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. More formally, the set $C \subseteq \mathcal{P}(M)$ is a partition of M, i.e.,

— $\forall c_1, c_2 \in C : c_1 \neq c_2 \Rightarrow c_1 \cap c_2 = \emptyset$ (no overlap)
— $\bigcup_{c \in C} = M$ (complete coverage)

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 which 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 of success in the encapsulation of the system. Note that 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 it is important that the number of change-sets in a change-set series is 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.

4.2. Snapshot-based versus Historic

To compare a snapshot-based metric, e.g., the number of components of a system, against a historic metric, e.g., the ratio of local change, two input parameters need to be defined: the exact moment of the snapshot for the snapshot-based metric and the change-set series for the historic metric. To increase the accuracy of the calculation of the historic 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.

In our experiment, this balance is obtained by calculating the historic metric using a change-sets 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. 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 historic metric since there is no single snapshot-based metric value we can compare against. However, the value of a historic 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).

Note that this approach deviates from the commonly used design (see for example the experiments described in [Yu et al. 2010; Romano and Pinzger 2011]) in which a recent snapshot of the system is chosen and the historic metric is calculated based on the *entire* history of a system. The implicit assumption which is 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. We believe that this assumption is not valid in all situations, or should at least be verified to ensure that the comparison between these two types of metrics is meaningful.

4.3. Metric Stability

One of the parameters that needs to be instantiated for this approach 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

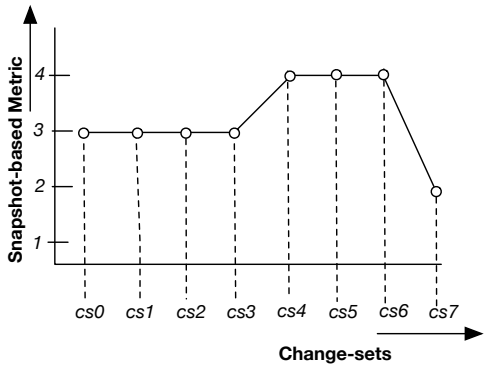

Fig. 2. The value of a snapshot-based metric over time determines the change-set series on which the historic metric should be calculated.

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 variability of the metric and the context in which the metric is used. For example, the number of components is not expected to change in a maintenance setting, while during the early stages of development it is expected that this number fluctuates heavily. In the implementation of the experiment a definition of "stable" related to the context of our goal is chosen.

4.4. Statistical Analysis

Given this approach, the aim of the experiment is to see whether the architecture metrics listed in Table II 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 (for example 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. Because we cannot assume a normal distribution in any of the metrics, the specific correlation test used here is the Spearman rank correlation. 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.

Using the thresholds defined by Hopkins [2000] 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, indicating that the chance that the found correlation is due to random chance is less than 1 percent.

In this set-up, multiple hypotheses are tested using the same dataset. In this case a Bonferroni correction [Hopkins 2000] is appropriate to prevent 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 8.3.

4.5. Preventing Project Bias

In this set-up 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 proportionally 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 a multiple regression analysis will be performed 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 a stepwise selection using a backward elimination model is applied [Hopkins 2000]. 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.

4.6. Summary

To summarize, the procedure for testing the correlation between each snapshot-based architecture metric and the historic 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 historic metric for all selected change-set series

Step 4: For each metric, calculate the correlation between the snapshot-based metric value and the historic metric using data from all projects

Step 5: Verify the influence of individual projects on significant correlations

5. EXPERIMENT IMPLEMENTATION

5.1. Metric Stability

The context of this experiment is that of software analysts and software quality evaluators. In such a context it is useful to place a system within a bin according to its metrics in terms of different categories, i.e., 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) as soon as 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 as soon as the value is placed in a different bin, with a bin-size of 0.01. For example, when the values of the metric on snapshots (t_1, t_2, t_3) are (0.243, 0.246, 0.253), 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 8.2.

5.2. Stable Period Identification

To determine the time-periods for which a snapshot-based metric is stable, the value of the metric must 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. In order to compromise between precision and calculation effort a sampling approach is used.

Snapshots of the system are extracted on regular intervals, i.e., on every first day of the month, and all change-sets in 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 in Section 5.1) in between snapshots t_n and t_{n+1} all change-sets up until snapshot t_n are grouped into a single change-set series, while all change-sets in between t_n and t_{n+1} are discarded.

Note that for a highly unstable metric the effect may be that all data-points are discarded. This side-effect is not seen as a problem in this experiment because our aim is to identify metrics which can be used to steer development efforts, which requires the metric to be stable for a considerable period of time to enable the definition of corrective actions.

Also note that a change in the value of the snapshot-based metric indicates a change in the architecture of the system. Because our study only takes into account stable periods the study is focussed on determining the *effect* of these architectural changes, instead of the nature of the the architecture changes themselves. Even though we consider the actual architecture changes out of scope of the current research, we envision a thorough exploration of the unstable periods of a metric to be part of our future work.

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 8.2.

5.3. Subject systems

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

— *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.

— *Subversion repository*: this source-code repository system facilitates easy extraction of individual change-sets by assuming that each commit is a change-set.

— *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 III lists the names of the systems used together with the overall start date and end date considered. 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.

5.4. Architectural Model Instantiation for Java

Following our earlier approaches [Bouwers et al. 2011a; Bouwers et al. 2011b] 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 taken as dependencies. Virtual calls (for example created by polymorphism or by interface implementations) are not considered to be a dependency. In other words, a call to an interface creates a dependency on that interface, but not on classes implementing that interface. All metrics are calculated on relevant source-code modules using the Software Analysis Toolkit of the Software Improvement Group (SIG)[1]. 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, for example, testing or demo purposes is considered to be out of scope for this experiment (and therefore not included in the numbers of Table III).

[1] `http://www.sig.eu`

Table III. Subject systems used in the study

		Period		Size (KLOC)	
Name	**Repository**	**Start**	**End**	**Start**	**End**
Ant	http://svn.apache.org/repos/asf/ant/core/trunk	2000-02	2011-05	3	97
Argouml	http://argouml.tigris.org/svn/argouml/trunk/src/argouml-app	2008-03	2011-07	113	108
Beehive	http://svn.apache.org/repos/asf/beehive/trunk/	2004-08	2008-10	45	86
Crawljax	http://crawljax.googlecode.com/svn/trunk/	2010-01	2011-07	6	7
Findbugs	http://findbugs.googlecode.com/svn/trunk/findbugs	2003-04	2011-07	7	97
Jasperreports	http://jasperforge.org/svn/repos/jasperreports/trunk/jasperreports	2004-01	2011-08	28	171
Jedit	https://jedit.svn.sourceforge.net/svnroot/jedit/jEdit/trunk	2001-10	2011-08	35	79
Jhotdraw	https://jhotdraw.svn.sourceforge.net/svnroot/jhotdraw/trunk/JHotDraw	2001-03	2005-05	8	20
Lucene	http://svn.apache.org/repos/asf/lucene/dev/trunk/lucene/	2001-10	2011-08	6	67
Struts2	http://svn.apache.org/repos/asf/struts/struts2/trunk	2006-06	2011-07	25	22

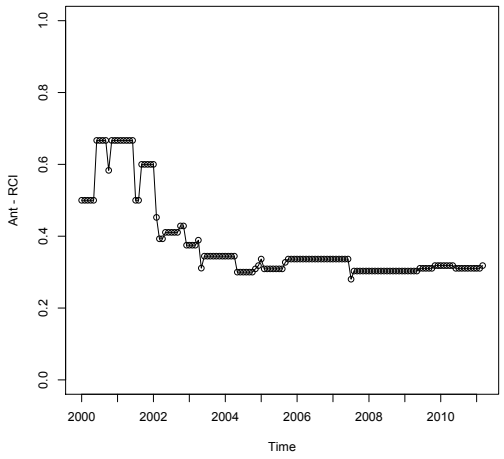

Fig. 3. The value of the RCI metric plotted over the life-time of the Ant system.

6. EXPERIMENT RESULTS

6.1. Experiment Package

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 1/0.01

6.2. Stable Periods

The first step in the experiment is to determine the stable periods for each of the twelve snapshot-based metrics. To illustrate the need for determining these stable periods the value for the metric RCI for the system Ant is plotted Figure 3. This graph shows that there is considerable fluctuation in this metric during the development of the system, resulting in a total of 15 stable periods.

Descriptive statistics of the stable periods per metric illustrating four important characteristics of the data-set are shown in Table IV.

Table IV. Descriptive statistics for the stable periods per snapshot-based metric

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

Table V. Descriptive statistics of the ratio of local change calculated on the stable periods described in Table IV

	Ratio of local change		
Metric	Min	Median	Max
RCI	0.00	0.84	1
CCD	0.37	0.84	1
ACD	0.00	0.85	1
NCD	0.37	0.84	1
CDI	0.00	0.84	1
IBC	0.24	0.86	1
OBC	0.25	0.86	1
IC	0.16	0.86	1
NBD	0.00	0.84	1
CB	0.35	0.86	1
MSUI	0.36	0.84	1
NC	0.18	0.83	1

Table VI. 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

First of all, in the second column of Table IV the number of stable periods per metric is shown. In all cases this number exceeds the number of projects (10), from which we can conclude that each metric changes over time and thus can be used to distinguish different states of a system.

Secondly, descriptive statistics of the number of months per stable period are shown in columns $3-5$ of Table IV. As discussed in Section 5.2 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, we consider such a time-frame to be long enough to define corrective actions.

Thirdly, the sixth column of Table IV 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, which means that more than half of the total development time of the systems is covered by the stable periods. From this we conclude that the metrics are stable enough to be used in the context of our experiment.

Lastly, columns $7-11$ of Table IV show descriptive statistics for the length of the change-set series based on the stable periods. As discussed in Section 4.2 it is desirable to have longer change-set series to ensure an accurate representation of the ratio of local change. However, Table IV 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), we consider these change-sets series to be accurate enough and use all of the series in the current experiment.

Table VII. Linear model for predicting the ratio of local change using the percentage of Inbound Code and specific projects

Model	Coef.	Std. Error	t	p-value	R^2
(Constant)	1.032	0.033	31.114	< 0.01	–
IBC	-0.616	0.079	-7.811	< 0.01	0.275
Beehive	-0.321	0.073	-4.419	< 0.01	0.072
Findbugs	0.087	0.030	2.843	< 0.01	0.032
Lucene	-0.065	0.032	-2.011	< 0.05	0.022
Model Summary: $R2 = 0.4006; p \leq 0.01$					

Table VIII. Linear model for predicting the ratio of local change using the percentage of Outbound Code and specific projects

Model	Coef.	Std. Error	t	p-value	R^2
(Constant)	0.987	0.036	27.528	< 0.01	–
OBC	-0.460	0.061	-7.517	< 0.01	0.167
Ant	0.253	0.038	6.720	< 0.01	0.094
ArgoUML	0.245	0.043	5.650	< 0.01	0.067
Findbugs	0.180	0.031	5.818	< 0.01	0.064
Lucene	0.145	0.026	5.509	< 0.01	0.080
Model Summary: $R2 = 0.4733; p \leq 0.01$					

Table IX. Linear model for predicting the ratio of local change using the percentage of Internal Code and specific projects

Model	Coef.	Std. Error	t	p-value	R^2
(Constant)	0.564	0.030	19.200	< 0.01	–
IC	0.641	0.076	8.427	< 0.01	0.247
Ant	0.228	0.039	5.873	< 0.01	0.074
ArgoUML	0.139	0.046	2.984	< 0.01	0.013
Beehive	-0.190	0.087	-2.191	< 0.05	0.023
Findbugs	0.199	0.037	5.334	< 0.01	0.057
Lucene	0.107	0.027	3.843	< 0.01	0.051
Model Summary: $R2 = 0.4672; p \leq 0.01$					

6.3. Ratio of local change

For each of the snapshot-based metrics the ratio of local change is calculated based on the stable periods described in Table IV. Table V 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.

In this paper we do not study the ratio of local change within the *unstable* periods. We suspect that the ratio of local change during unstable periods tends to be lower than during stable periods, but an in-depth analysis of this hypothesis is future work.

6.4. 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 VI shows the results of the tests together with both the corrected as well as 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 IV 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 [Hopkins 2000].

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 4.5 for details about this approach). The resulting models for respectively IBC, OBC and IC are listed in table Table VII, VIII and IX. In all three cases some of the projects have a significant influence (the other projects are eliminated from the model because their influence is non-significant), but the value of the snapshot-based metric is the most significant factor (see the last column) in each model. In other words, the snapshot-based metric explains most of the variation in the ratio of local change.

7. DISCUSSION

7.1. Key Findings

The results of the experiment shows that there is not enough evidence to reject the null hypothesis 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 historic ratio of local change*.

In our opinion, the reasoning behind this correlation lies in 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 only the intensity 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 4.1. Based on these observations the answer to our research question becomes:

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

7.2. Beyond "boxes and arrows"

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 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 intensity of the dependencies between components, we suspect that they are not able to properly quantify the extent of the dependency between components.

Since more sophisticated views apparently allow for the definition of more powerful metrics, we recommend investigating metrics based on even more sophisticated views. For example, one could augment the view with information on the entry points of the system (e.g., the external API for a library or user interface actions for an application) or on the participation of the various components in the implementation of various concerns. Construction of such views currently requires manual input. To make construction of such views feasible in the context of (repeated) evaluations, techniques must be found to lift such limitations. The first steps in this area have already been taken [Olszak et al. 2012].

7.3. Generalization

The current implementation of the experiment limits the generalizability of the results to open-source systems written in Java. In our previous work [Bouwers et al. 2011b] 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 4 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 IV 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.

7.4. 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.

From an organizational perspective, this metric can be used to provide a first indication of the level of the encapsulation of many systems across an application portfolio. By combining the value of this metric with other key-properties of the systems (e.g., size, business criticality or expected change rate) the allocation of resources can be conducted on a more informed basis.

Based on the findings in this report, 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. The results of applying these metrics in over 500 industrial projects are currently being gathered and analyzed. Reporting on these findings is part of our future work.

As with all metrics, these metrics should not be used in an evaluation setting in isolation [Bouwers et al. 2012]. To ensure a balanced evaluation we recommend to combine the metrics with other metrics which assess, for example, the analyzability or the complexity of the software architecture under review. In our future work, we will report on our experiences with using the percentage of internal code in combination with other metrics in the context of software architecture evaluations in industry.

7.5. Metric Stability

As can be seen in Table IV, 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 4.

8. THREATS TO VALIDITY

Following the guidelines of Wohlin et al. [2000] 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 7.3, this category of threats is not discussed in detail in this section.

8.1. 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 [1994]. In addition, "encapsulate what changes" is a well known and widely recognized design principle [Gamma et al. 1995]. Additionally, the control variables MSUI, CB and NC do not show any significant moderate to strong correlation, which limits the threat that the ratio of local change measures a different external quality attribute.

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. We have performed many such componentizations for industry systems and validated them with the development teams. In many of these cases the top-level packages are indeed considered to be the top-level components. Thus, even though we did not perform an explicit validation of the component-structure with the developers of the systems in our experiment, the naming of the top-level packages seems to indicate a valid component structure.

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 which 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.

Additionally, it could be the case that a commit consists solely of automated refactorings such as renaming a class, or changing the license statement in comments only. We did not explicitly deal with these cases, but believe that the first type is not problematic since an automated refactoring which impacts multiple components is a legitimate non-local change-set. The second type might be problematic, but we consider the impact of this case to be minor due to the large number of change-sets used both per stable period and in total.

8.2. 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. One of these confounding factors, the influence of specific projects on the significant correlations, has already been addressed in the design and results of the study.

A second confounding factor is the choice for taking monthly snapshots to determine the stable periods of the snapshot-based metrics. Taking longer or shorter periods could result in a different number of stable time-periods, which could influence the variance of the ratio of local change. However, as can be determined from the data in Table IV, the median number of months which are taken into account as a stable period is between two and six months and over 60% of the development time is covered by these periods. This shows that using a one month period between snapshots already covers a considerable portion of the development of the system, thus using a shorter period of time between snapshots is not immediately 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 we do not believe that this situation occurs often enough to influence the results significantly.

A third factor is the pragmatic choice to consider a percentage stable as long as the value stays within the same bin with a bin-size of 0.01. As discussed before, taking a more narrow or a broader bin-size can lead to more (or less) variance in the snapshot based metric which in turn leads to more (or less) co-variance with the ratio of local change. Again, the median length of the periods,

the number of change-sets per period and the variation in the metrics as shown in Table IV do not indicate that the 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. We hypothesize that the bin-size could be defined on the stability characteristics of the metrics determined by, for example, techniques taken from time-series analysis. 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 will not invalidate the correlations found with the current bin-size.

8.3. 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 7.1 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 actually be valid indicators. As discussed before, the application of a Bonferroni correction could cause false negatives. Inspecting the non-corrected p-values shown in Table VI 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.

9. RELATED WORK

The change-history of a software system has previously been used to, amongst others, validate designs [Zimmermann et al. 2003], predict source-code changes [Ying et al. 2004] and for predicting faults [Graves et al. 2000]. 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 historic 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. [2012] for an overview.

On the component level this type of validation has been done by Yu et al. [2010]. 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.

10. 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. The contributions of this paper are:

— A selection of twelve existing system-level architecture metrics which are suitable candidates to be used in an late architecture evaluation context.

— An experiment design in which the value of these twelve metrics are correlated with the ratio of local change during periods for which the metrics are representative.

— A stability analysis on twelve metrics which 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.

— 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. For technical stakeholders the correlation between these percentages and the ratio of local change is expected to be sufficient to justify using the metrics within a decision making process. However, based on our experience we expect non-technical stakeholders to require a stronger relation between these values and, for example, costs or operational risks. In our future work we plan to investigate the relationship between these metrics and this type of non-technical aspects of the software development process. In addition, we plan 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, 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 historic metric is calculated must be verified for any experiment in which these two types of metrics are correlated.

There are two main areas in which future work is needed. First of all, we plan on verifying the usefulness of the percentage of internal code by using this metrics in a late architecture evaluation setting. This is done by incorporating this metrics in the suite of metrics which is used by SIG to conduct (repeated) architectural evaluations. The first results of applying this metric in the evaluation of over 500 industrial systems are promising.

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

ALLEN, E. B. AND KHOSHGOFTAAR, T. M. 1999. Measuring coupling and cohesion: An information-theory approach. In *Proceedings of the 6th International Symposium on Software Metrics*. IEEE Computer Society, Washington, DC, USA.

ANAN, M., SAIEDIAN, H., AND RYOO, J. 2009. An architecture-centric software maintainability assessment using information theory. *Journal of Software Maintenance and Evolution 21*, 1–18.

BASILI, V. R., CALDIERA, G., AND ROMBACH, H. D. 1994. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley.

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

BOUWERS, E., CORREIA, J., VAN DEURSEN, A., AND VISSER, J. 2011a. Quantifying the analyzability of software architectures. In *Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA 2011)*. IEEE Computer Society.

BOUWERS, E., VAN DEURSEN, A., AND VISSER, J. 2011b. Dependency profiles for software architecture evaluations. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011)*. IEEE Computer Society.

BOUWERS, E., VISSER, J., AND VAN DEURSEN, A. 2012. Getting what you measure. *Communications of the ACM 55,* 7, 54–59.

BRIAND, L. C., DALY, J. W., AND WÜST, J. 1998. A unified framework for cohesion measurement in object-orientedsystems. *Empirical Software Engineering 3,* 1, 65–117.

BRIAND, L. C., MORASCA, S., AND BASILI, V. R. 1993. Measuring and assessing maintainability at the end of high level design. In *Proceedings of the Conference on Software Maintenance (ICSM 1993)*. IEEE Computer Society, Washington, DC, USA, 88–97.

CLEMENTS, P., BACHMANN, F., BASS, L., GARLAN, D., IVERS, J., LITTLE, R., NORD, R., AND STAFFORD, J. 2003. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Boston, MA.

CLEMENTS, P., KAZMAN, R., AND KLEIN, M. 2002. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional.

DOBRICA, L. AND NIEMELÄ, E. 2002. A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering 28,* 7, 638–653.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

GRAVES, T. L., KARR, A. F., MARRON, J. S., AND SIY, H. 2000. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering 26*, 653–661.

HEITLAGER, I., KUIPERS, T., AND VISSER, J. 2007. A practical model for measuring maintainability. In *QUATIC '07: Proc. 6th Int. Conf. on Quality of Information and Communications Technology*. IEEE Computer Society, 30–39.

HOPKINS, W. G. 2000. *A new view of statistics.* Internet Society for Sport Science.

KOGUT, P. AND CLEMENTS, P. 1994. The software architecture renaissance. *Crosstalk - The Journal of Defense Software Engineering 7*, 20–24.

KOZIOLEK, H. 2011. Sustainability evaluation of software architectures: a systematic review. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*. QoSA-ISARCS '11. ACM, New York, NY, USA, 3–12.

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

LU, H., ZHOU, Y., XU, B., LEUNG, H., AND CHEN, L. 2012. The ability of object-oriented metrics to predict change-proneness: a meta-analysis. *Empirical Software Engineering 17*, 200–242. 10.1007/s10664-011-9170-z.

MANCORIDIS, S., MITCHELL, B. S., RORRES, C., CHEN, Y., AND GANSNER, E. R. 1998. Using automatic clustering to produce high-level system organizations of source code. In *IWPC '98: Proc. 6th Int. Workshop on Program Comprehension*. IEEE.

MARTIN, R. C. 2003. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

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

OLSZAK, A., BOUWERS, E., JØRGENSEN, B. N., AND VISSER, J. 2012. Detection of seed methods for quantification of feature confinement. In *TOOLS Europe 2012: Proceedings of the 50th 50th International Conference on Objects, Models, Components, Patterns*.

PARNAS, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15,* 12, 1053–1058.

PERRY, D. E. AND WOLF, A. L. 1992. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes 17,* 4, 40–52.

ROMANO, D. AND PINZGER, M. 2011. 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.

SANGWAN, R., VERCELLONE-SMITH, P., AND LAPLANTE, P. 2008. Structural Epochs in the complexity of Software over Time. *Software, IEEE 25,* 4, 66–73.

SANT'ANNA, C., FIGUEIREDO, E., GARCIA, A., AND LUCENA, C. 2007. On the modularity of software architectures: A concern-driven measurement framework. In *Software Architecture*, F. Oquendo, Ed. Lecture Notes in Computer Science Series, vol. 4758. Springer Berlin / Heidelberg, 207–224.

SARKAR, S., KAK, A. C., AND RAMA, G. M. 2008. Metrics for measuring the quality of modularization of large-scale object-oriented software. *IEEE Transactions on Software Engineering 34*, 700–720.

SARKAR, S., RAMA, G. M., AND KAK, A. C. 2007. API-based and information-theoretic metrics for measuring the quality of software modularization. *IEEE Transactions of Software Engineering 33,* 1, 14–32.

WOHLIN, C., RUNESON, P., HOST, M., OHLSSON, C., REGNELL, B., AND WESSLÉN, A. 2000. *Experimentation in Software Engineering: an Introduction*. Kluver Academic Publishers.

YING, A. T. T., MURPHY, G. C., NG, R., AND CHU-CARROLL, M. C. 2004. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering 30*, 574–586.

YU, L., MISHRA, A., AND RAMASWAMY, S. 2010. Component co-evolution and component dependency: speculations and verifications. *IET Software 4,* 4, 252–267.

ZIMMERMANN, T., DIEHL, S., AND ZELLER, A. 2003. How history justifies system architecture (or not). In *Proceedings of the 6th International Workshop on Principles of Software Evolution*. IEEE Computer Society, Washington, DC, USA, 73–83.

A. ARCHITECTURE METRICS OVERVIEW

Table X provides an overview of the metrics considered in the evaluation of the metrics as described in Section 3.

Table X. The overview of a comparison of architecture metrics against the criteria as listed in Section 3 grouped per source

Source	Name	Abbr.	C1	C2	C3	C4	C5	C6
Briand et al. [1993]	Ratio of Cohesive Interactions	RCI	Y	Y	Y	Y	Y	Y
Briand et al. [1993]	Import Coupling	ImC	N	-	-	-	-	-
Briand et al. [1993]	Export Coupling	ExC	N	-	-	-	-	-
Lakos [1996]	Cumulative Component Dependency	CCD	Y	Y	Y	Y	Y	Y
Lakos [1996]	Average CCD	ACD	Y	Y	Y	Y	Y	Y
Lakos [1996]	Normalized CCD	NCD	Y	Y	Y	Y	Y	Y
Mancoridis et al. [1998]	Modularization Quality	MQ	Y	Y	Y	N	-	-
Allen et al. [1999]	Intramodule Coupling	INMC	Y	Y	Y	N	-	-
Allen et al. [1999]	Cohesion of a Modular System	COMS	Y	Y	Y	N	-	-
Anan et al. [2009]	Information Entropy of an Architectural Slicing	IEAS	N	-	-	-	-	-
Anan et al. [2009]	Coupling between Architecture Slicings	ASC	N	-	-	-	-	-
Anan et al. [2009]	System Coupling	SC	Y	Y	Y	N	-	-
Anan et al. [2009]	Architecture Slicing Cohesion	ASC	N	-	-	-	-	-
Martin [2003]	Afferent Coupling	Ca	N	-	-	-	-	-
Martin [2003]	Efferent Coupling	Ce	N	-	-	-	-	-
Martin [2003]	Instability	I	N	-	-	-	-	-
Martin [2003]	Abstractness	A	N	-	-	-	-	-
Martin [2003]	Distance from the Main Sequence	D	N	-	-	-	-	-
Sant'Anna et al. [2007]	Concern Diffusion over Architectural Components	CDAC	N	-	-	-	-	-
Sant'Anna et al. [2007]	Concern Diffusion over Architectural Interfaces	CDAI	N	-	-	-	-	-
Sant'Anna et al. [2007]	Concern Diffusion over Architectural Operations	CDAO	N	-	-	-	-	-
Sant'Anna et al. [2007]	Component-level Interfacing Between Concerns	CIBS	N	-	-	-	-	-
Sant'Anna et al. [2007]	Interface-level Interfacing Between Concerns	IIBC	N	-	-	-	-	-
Sant'Anna et al. [2007]	Operation-level Overlapping Between Concerns	OOBC	N	-	-	-	-	-
Sant'Anna et al. [2007]	Lack of Concern-based Cohesion LLC	IIBC	Y	N	-	-	-	-
Sarkar et al. [2007]	Module Interaction Index	MII	Y	Y	N	-	-	-
Sarkar et al. [2007]	Nn-API Function Closedness	NAFC	Y	N	-	-	-	-
Sarkar et al. [2007]	API Function Usage Index	APIU	Y	N	-	-	-	-
Sarkar et al. [2007]	Implicit Dependency Index	IDI	Y	N	-	-	-	-
Sarkar et al. [2007]	Module Size Uniformity Index	MSUI	Y	Y	Y	Y	N	Y
Sarkar et al. [2007]	Module Size Boundedness Index	MSBI	Y	N	-	-	-	-
Sarkar et al. [2007]	Cyclic Dependency Index	CDI	Y	Y	Y	Y	Y	Y
Sarkar et al. [2007]	The Layer Organization Index	LOI	Y	N	-	-	-	-
Sarkar et al. [2007]	Module Interaction Stability Index	MISI	Y	N	-	-	-	-
Sarkar et al. [2007]	Normalized Testability-Dependency Metric	NTDM	Y	Y	Y	Y	Y	Y
Sarkar et al. [2007]	Concept Domination Metric	CDM	Y	N	-	-	-	-
Sarkar et al. [2007]	Concept Coherency Metric	CCM	Y	N	-	-	-	-
Sarkar et al. [2008]	Base-Class Fragility Index	BCFI	Y	Y	N	-	-	-
Sarkar et al. [2008]	Inheritance Based Coupling	IBC	Y	Y	N	-	-	-
Sarkar et al. [2008]	Not-Programming-to-Interfaces Index	NPII	Y	Y	N	-	-	-
Sarkar et al. [2008]	Association-induced Coupling	AIC	Y	Y	N	-	-	-
Sarkar et al. [2008]	State Access Violation Index	SAVI	Y	Y	N	-	-	-
Sarkar et al. [2008]	Plugin Pollution Index	PPI	Y	Y	N	-	-	-
Sarkar et al. [2008]	API Usage Index	APIU	Y	Y	N	-	-	-
Sarkar et al. [2008]	Common Reuse of Modules	CReuM	Y	Y	N	-	-	-
Sangwan et al. [2008]	Excessive Structural Complexity	XS	Y	Y	Y	N	-	-
Bouwers et al. [2011a]	Component Balance	CB	Y	Y	Y	Y	N	Y
Bouwers et al. [2011b]	Internal Code	IC	Y	Y	Y	Y	Y	Y
Bouwers et al. [2011b]	Inbound Code	IBC	Y	Y	Y	Y	Y	Y
Bouwers et al. [2011b]	Outbound Code	OBC	Y	Y	Y	Y	Y	Y
	Number of components	NC	Y	Y	Y	Y	Y	Y
	Number of Binary Dependencies	NBD	Y	Y	Y	Y	Y	Y
	Number of Absolute Dependencies	NAD	Y	Y	Y	Y	Y	N

TUD-SERG-2011-031-a
ISSN 1872-5392