Delft University of Technology
Software Engineering Research Group
Technical Report Series

Dependency Profiles for Software Architecture Evaluations

Eric Bouwers, Arie van Deursen, Joost Visser

Report TUD-SERG-2011-015

TUD-SERG-2011-015

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: Accepted for publication in the Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM), 2011, IEEE Computer Society.

Dependency Profiles for Software Architecture Evaluations

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

* Software Improvement Group, Amsterdam, The Netherlands
E-mail {e.bouwers,j.visser}@sig.eu
‡ Delft University of Technology, Delft, The Netherlands
E-mail {Arie.vanDeursen,E.M.Bouwers}@tudelft.nl

***Abstract*—In this paper we introduce the concept of a "dependency profile", a system level metric aimed at quantifying the level of encapsulation and independence within a system. We verify that these profiles are suitable to be used in an evaluation context by inspecting the dependency profiles for a repository of almost 100 systems. Furthermore we outline the steps we are taking to validate the usefulness and applicability of the proposed profiles.**

I. Introduction

Software architecture is loosely defined as the organizational structure of a software system including components, connections, constraints, and rationale [7]. Since the architecture of a software system greatly influences all of a system's quality attributes [4], it is important to regularly evaluate the actual, as-implemented, software architecture of a system.

In order to reduce the amount of time and effort needed to perform such an evaluation, an evaluator can use software metrics to spot outliers and identify areas within a system which are in need of a more detailed evaluation. Additionally, the use of metrics reduces the need for expert opinion, thus making the evaluation more objective and repeatable.

For a metric to be useful in an evaluation context, several characteristics are desirable [6]. For instance, the metric needs to be *simple to explain* to ensure that non-technical decision makers can understand them. Furthermore, in order to allow an evaluation of a diverse application portfolio the metrics should be *as independent of technology as possible*. The ability to perform a *root-cause analysis* is also desirable to ensure that the metrics can provide a basis to determine which actions need to be taken. Lastly, metrics which are *easy to implement and compute* are desired as to reduce the initial investment for performing evaluations.

Research on metrics for software architectures has traditionally focussed on the way components depend on each other and how components are internally structured (coupling and cohesion [9], [10]). To the best of our knowledge, all of the existing metrics for architecture level dependencies fail to meet at least one of the desired characteristics outlined above.

In this paper we propose the concept of a *dependency profile* which categorizes all modules in a system based on their dependencies. This purpose of the dependency profile is two-fold. On the one hand it is aimed at capturing the degree in which the components within a system encapsulate the functionality they offer. On the other hand, the profile quantifies the degree in which components are dependent upon each other. We assess to what extent the dependency profile meet the four criteria just discussed by examining a benchmark of almost 100 systems totaling over 12.5 million lines of code. Additionally we outline a plan to validate the profile against the type of changes that occur within a system.

II. Background

To illustrate why existing metrics for quantifying the dependencies between components of a system are less suitable to be used in an evaluation context we present a short overview of typically found shortcomings.

To start, metrics which are simple to explain such as the basic number of incoming and outgoing dependencies allow for root-cause analyses. However, since larger systems tend to have a higher number of dependencies these metrics should be normalized against the size of the system to allow systems of various sizes to be compared.

More complex coupling/cohesion metrics such as those defined by Briand et al. [2] or in the well-known C&K suite of metrics [3] (including variations), suffer from the same problem of not being normalized against the size of the software unit they are measuring. Additionally, these class-level metrics are designed to target systems written in object-oriented languages, while ideally a metric would be independent of technology.

And although there are extensions to these coupling metrics that are normalized, see for example Gui [5], the proposed normalization process tends to decrease the ability to perform root-cause analyses because the outliers in the data, which are the interesting data-points, are usually hidden by the normalization. The same problem applies to metrics defined to rank cluster algorithms, for example the Modularization Quality-metric defined by Mancoridis et al. [8].

III. Dependency Profiles

We define a metric to quantify the dependencies within a system by placing all *modules* of a system (e.g., Java classes or C files) into four distinct categories. This categorization is based on the way in which the modules are grouped into

components (e.g., Java packages or C directories) and how the modules interact with modules outside their own component.

A. Terminology

Let $S = \langle M, C, D \rangle$ be a system, consisting of a set of modules M, a set of components C and a set of dependencies between modules D. 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.,

- $\forall c_1, c_2 \in C : c_1 \neq c_2 \Rightarrow c_1 \cap c_2 = \emptyset$.
- $\bigcup_{c \in C} = M$

For $(m, m') \in D$ we write $m \rightarrow m'$ to represents a directed dependency from module $m \in M$ to module $m' \in M$.

For each module $m \in M$ it is possible to obtain the containing component through a function $component : M \rightarrow C$. In addition, for a component $c \in C$ we use $\overline{c}$ to denote the complement of c, i.e., all modules not contained in c.

Lastly, each module has a given size (measured by, for example, the lines of code or function points), which is captured by a function $size : M \rightarrow \mathbb{N}$. The volume of a component $c \in C$ is defined simply as the sum of the size of its modules, thus:

$$volume(c) = \sum_{m \in c} size(m)$$

B. Types of code

Each module within the components of a system can be divided into one of four categories, see Fig. 1:

- *Hidden modules* (1): modules which only have dependencies (either incoming or outgoing) involving modules inside the component.
- *Inbound modules* (2): modules which do not have outgoing dependencies to modules outside the component, but have incoming dependencies from modules outside the component.
- *Outbound modules* (3): modules which do not have incoming dependencies from modules outside the component, but have outgoing dependencies to modules outside the component.
- *Transit modules* (4): modules which have dependencies (both incoming and outgoing) coming from/going to modules outside the component.

For each of these categories a function of type $C \rightarrow 2^M$ can be defined which, given a component C, returns the set of modules within that component which belong to that category. Table III-A lists the definitions of those functions. Using these functions, each category of modules can be turned into a normalized metric by calculating the percentage of code in a system which belongs to each category. For example, the percentage of *hiddenCode* of a system is defined as:

$$hiddenCode(S) = \sum_{c \in S} \frac{volume(hiddenModules(c))}{volume(c)}$$

Definitions of the metrics for *inboundCode*, *outboundCode* and *transitCode* are similar.

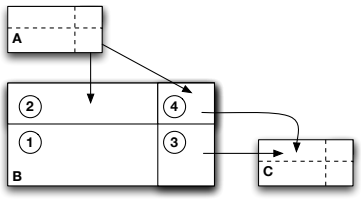

Fig. 1. Three components illustrating the four different types of modules within a system; 1) hidden modules, 2) inbound modules, 3) outbound modules and 4) transit modules. Arrows denote dependencies from/to modules within other components.

C. Dependency Profile

Using the metrics defined above we define a *Dependency Profile* as a quadruple of the four types of code:

$$\langle \quad hiddenCode(S) \quad , inboundCode(S) \\ outboundCode(S) \quad , transitCode(S) \quad \rangle$$

A typical instantiation of such a profile is $\langle 75\%, 10\%, 15\%, 5\% \rangle$, which means that 75 percent of the volume of the system falls into the *hiddenCode*-category, 10 percent falls into the *inboundCode*-category, etc. We hypothesize that this dependency profile can be used to quantify two quality aspects of a software system: the degree of encapsulation and the degree of independence.

The concept of *encapsulation* is often used to refer to the level in which the implementation details of functionality are abstracted away by an interface. A high level of encapsulation is desirable since this should mean that changes to the implementation can be done without the need to change clients which are using the interface. We expect that the *inboundCode* metric can be used to measure this quality aspect. To illustrate we compare a system A with a dependency profile of $\langle 50\%, 30\%, 18\%, 2\% \rangle$ with a system B with a dependency profile of $\langle 50\%, 15\%, 34\%, 1\% \rangle$. In system A there is a higher percentage of code which is called from outside the component in which it is defined, which leads to a higher chance that a change in this specific component propagates to other components in the system. We hypothesize that a high value of *inboundCode* shows that there is a low level of encapsulation in the system.

Analogously, *independence* is used to refer to the level in which components of a system rely on other components (either interface or implementation) in the implementation of their own functionality. A high level of independence is desirable since this should mean that changes in modules outside the component should not propagate to the component itself. We expect that the *outboundCode* metric can be used to measure this quality aspect since this metric quantifies the portion of the system which is used by other components. In the example systems above, system B has a higher percentage of code which depends on code outside the component in which it is defined. This leads to a higher chance that a change in a component will propagate to this specific component. We hypothesize that a high value of *outboundCode* indicates that there is a low level of independence in the system.

$hiddenModules(c)$	$\{m \in c \mid \nexists\ m_i \in \overline{c} : m_i \to m \in D \ \wedge \ \nexists\ m_o \in \overline{c} : m \to m_o \in D\}$
$inboundModules(c)$	$\{m \in c \mid \exists\ m_i \in \overline{c} : m_i \to m \in D \ \wedge \ \nexists\ m_o \in \overline{c} : m \to m_o \in D\}$
$outboundModules(c)$	$\{m \in c \mid \nexists\ m_i \in \overline{c} : m_i \to m \in D \ \wedge \ \exists\ m_o \in \overline{c} : m \to m_o \in D\}$
$transitModules(c)$	$\{m \in c \mid \exists\ m_i \in \overline{c} : m_i \to m \in D \ \wedge \ \exists\ m_o \in \overline{c} : m \to m_o \in D\}$

TABLE I
CONDITIONS FOR EACH OF THE FOUR CATEGORIES OF MODULES

In both cases the percentage of *transitCode* should also be taken into account. This category contains those modules which both use and are used by modules in other components and are thus even more likely to propagate changes between components. Because of this issue, we hypothesize that although there might be some need for *transitCode*, for example in a component which connects two other components, it is desirable to have a low percentage of *transitCode* in a system.

IV. PRELIMINARY OBSERVATIONS

As a first evaluation of the dependency profiles we instantiate the above metric framework and use a repository of systems to observe the distribution for this specific instantiation. The repository is an extended version of the one used in [1] and contains systems of different sizes, development context (open-source versus industry) and technologies. The following table characterizes the repository in terms of number of systems per technology and development context:

	Java	.NET	C/C++	Total
Industry	45	17	6	68
Open source	20	4	3	27
Total	65	21	9	95

To ensure that the metrics can be calculated for all technologies we instantiate "module" as a source-file, "dependency" as a direct call relation and "component" as the first level of decomposition in the system. Determining the components of the systems follows the approach in [1], i.e., for all systems the top-level decomposition was made by a technical analyst of the Software Improvement Group based on the directory structure of the system and available documentation. For the industry systems this decomposition was validated with the development team. A chart showing the distribution of the dependency profiles for this repository and this instantiation is given in Fig. 2.

A. General Observations

A first observation that is clear from Fig. 2 is that the percentage of *hiddenCode* differs considerably for the systems in the repository, ranging from 7 to 100 percent with a median of 35 percent. Since having 100 percent of *hiddenCode* is strange, we investigated this particular system, an industry system written in C#, and found that each top-level component in the system was a specific service built upon an external framework. Since each service is independent from the all other services none of the services have code in common.

Another observation that can be made about the distribution is that a large portion of the repository (18 systems) does not have any *transitCode*, which corresponds with our initial

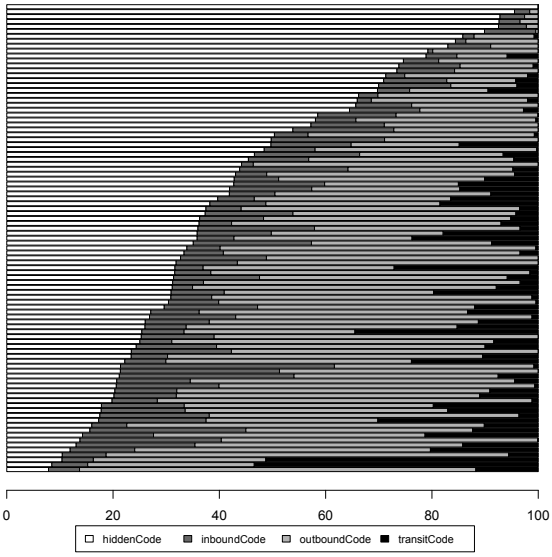

Fig. 2. Dependency profiles for a repository of systems, ordered by the percentage of hiddenCode. Each line represents a system.

expectation. However, the amount of *transitCode* rises to over 20 percent for 10 systems, having a maximum of 53 percent in the repository. Within these systems we expect to see a high frequency of propagating changes. In Section VI we provide an outline of how we plan to validate this hypothesis.

A last observation that can be made is that in almost all cases (only 9 exceptions) the amount of *outboundCode* is larger than the amount of *inboundCode*. This could indicate that, in general, there is a stronger focus on the design of the *provided* interface of a component than on restricting the *required* interface of a component.

B. Statistical Observations

To enable a fair comparison between systems of different sizes we need to ensure that there is no strong correlation between the size of the systems and the percentages in the dependency profiles. To assess this we use a Spearman rank correlation test using the size of the system in lines of code and the percentage of code in each of the four categories. No significant correlations were found for *hiddenCode* and *transitCode*, while both *inboundCode* and *outboundCode* have a weak correlation of -0.28 and 0.32 ($p < 0.01$) respectively. Thus we can conclude that there is no strong correlation between the size of a system and any of the four categories.

In addition, using a two-sided Kolmogorov−Smirnov test we can determine whether there is a significant difference between the distributions of two data samples. Using this test we did not find any significant differences between the distribution of the values for different development contexts

(industry versus open-source) or system type (application versus libraries). However, there are differences between the distribution in the values of *hiddenCode* for the Java technology versus other technologies. Inspecting the distributions shows that systems written in Java tend to have a lower percentage of *hiddenCode*. This difference in distribution does not mean that the metrics are technology dependent, but only that the metrics might consistently produce lower values for certain technologies. Determining the reasons, scope and impact of this issue is part of our future work.

V. Discussion

As discussed before there are four desirable characteristics of metrics to be useful in an practical evaluation setting [6]. We argue that the metrics used in the dependency profile as described in Section III feature these characteristics.

First of all, the metrics should be *simple to explain*. Even though the formal definition of the metrics can be considered complex, we believe that the intuition behind the metrics are easy to explain given the visual support of Fig. 1.

Secondly, the metrics should be *as technology independent as possible*. The definition of the four metrics contains no technology specific constraints, although certain definitions of "module" or "dependency" could make the metrics technology specific. By using a generic instantiation of the metrics as given in Section III there are no practical problems in comparing systems written in different technologies.

Furthermore, the metrics should allow for a *root-cause analysis*, which is relatively straight-forward. After first using the system level dependency profiles to discover a system in need of further investigation, the profile can be calculated on component level to determine which component contributes the most to each category of code. After the most interesting component has been found, the modules in the category of interest can be sorted according to their size to discover which module is contributing the most to this category. After determining the most interesting modules an expert should inspect the dependencies to/from these modules to determine *why* these dependencies are there and whether they are problematic.

Lastly, the metrics should be *straight-forward to implement*. We believe that existing tools which are capable of extracting dependencies and calculating the size of modules should have no problems implementing the needed metrics, and that this should require only a small amount of effort.

VI. Evaluation Design

To determine whether the intuition as described in Section III-C is correct we plan to test the following hypothesis:

- Systems with a low percentage of *inboundCode* plus *transitCode* have a better encapsulation and therefore changes in a component will less likely propagate to other components
- Systems with a low percentage of *outboundCode* plus *transitCode* have more independent components and therefore changes in a component will less likely propagate to other components

In order to validate these hypothesis we plan to perform a case-study examining the change-sets of a system using the framework proposed by Yu et al. [11]. This framework defines co-evolution of a system as either being internal (i.e., all modules in a change-set belong to a single component) or external (a change-set contains modules of multiple components).

We plan to calculate the frequency of external co-evolutions for a number of open-source systems. By correlating this frequency with the values of *inboundCode* plus *transitCode*, *outboundCode* plus *transitCode*, and a combination of the two measures we plan to validate or reject our two hypothesis.

Concurrently a more qualitative study will be performed in the form of case-studies on previously analyzed systems. In these case-studies the intuition of the metrics will be validated by connecting the dependency profiles with known architectural problems within the studied systems. Alternatively, the profiles are used in combination with existing techniques to determine whether there are problems in systems not previously evaluated. This qualitative study will also address the issue raised in Section IV-B.

VII. Conclusions

This paper makes the following contributions:

- The definition of a dependency profile with desirable characteristics for use in a software evaluation setting
- A first analysis of these profiles using a large repository of systems
- An outline of the evaluation strategy for the profiles

We are currently working on setting up the evaluation experiment and hope to report on the results in the near future.

References

[1] E. Bouwers, J. Correia, A. van Deursen, and J. Visser. Quantifying the analyzability of software architectures. In *Proc. 9th Working IEEE/IFIP Conf. on Software Architecture*. IEEE Computer Society, 2011.

[2] L. Briand, J. Daly, and J. Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Softw. Eng.*, 25(1):91 –121, jan/feb 1999.

[3] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20:476–493, 1994.

[4] P. Clements, R. Kazman, and M. Klein. *Evaluating software architectures*. Addison-Wesley, 2005.

[5] G. Gui and P. Scott. Ranking reusability of software components using coupling metrics. *Journal of Systems and Software*, 80(9):1450–1459, Sept. 2007.

[6] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *QUATIC '07: Proc. 6th Int. Conf. on Quality of Information and Communications Technology*, pages 30–39. IEEE Computer Society, 2007.

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

[8] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proc. IEEE Int. Conf. on Software Maintenance*, ICSM '99, pages 50–, Washington, DC, USA, 1999. IEEE Computer Society.

[9] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Syst. J.*, 13(2):115–139, 1974.

[10] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., 1979.

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

TUD-SERG-2011-015
ISSN 1872-5392