Understanding Some Software Quality Aspects from Architecture and Design Models

Lars Bratthall and Claes Wohlin
Dept. Communication Systems, Lund University
Box 118, SE 221 00 Lund, Sweden
+46-46-2220000
lars.bratthall|claes.wohlin@telecom.lth.se

Abstract

Software systems evolve over time and it is often difficult to maintain them. One reason for this is often that it is hard to understand the previous release. Further, even if architecture and design models are available and up to date, they primarily represent the functional behaviour of the system. To evaluate whether it is possible to also represent some non-functional aspects, an experiment has been conducted. The objective of the experiment is to evaluate the cognitive suitability of some visual representations that can be used to represent a control relation, software component size and component external and internal complexity. Ten different representations are evaluated in a controlled environment using 35 subjects. The results from the experiment show that it is possible to also represent some non-functional aspects. It is concluded that the incorporation of these representations in architecture and design descriptions is both easy and probably worthwhile. The incorporation of the representations should enhance the understanding of previous releases and hence help software developers in evolving and maintaining complex software systems.

1. Introduction

Software systems grow continuously, i.e. new versions of the software are released either to enhance, improve or correct its behaviour. Few software products are released once and then never updated. The evolution of software systems calls for an understanding of previous versions and releases of the software system. Moreover, it is not possible to rely solely on the understanding of the code. The code is important, but other descriptions are needed to support the understanding of software, for example, architecture and design descriptions. The latter descriptions provide valuable tools on a higher abstraction level than the code. Software architecture has gained an increased interest in the last few years. The value of architecture models has been highlighted among others by [5, 16, 24], and several types and views of these models have been suggested, along with some UML models, e.g. [1].

The architecture and design models are, however, primarily formulated to be used during development. This conjecture is based on the fact that no models, to the best of our knowledge, include information regarding issues that only can be gained after having developed the system. In other words, the models have no means of incorporating experiences from previous releases, for example, in terms of quality aspects. The models are developed to form a basis for the continuation of the implementation within a project and not between successive projects. We believe, however, that there are simple ways of including quality aspects in architecture and design descriptions. In particular, it should be possible to add information to the descriptions in the coding and testing phase of one release to help in the understanding as new versions of the software are to be developed.

Code decay and identification of fault-proneness between releases have gained an increasing interest, for example, [19, 22]. These are important areas to help guide development activities in future releases or identify candidates for reengineering efforts. A problem encountered in these studies is that the information about non-functional aspects of the software has to be found by archival analysis of the software, testing records and problem reports. Four major reasons for fault-proneness are size, relationships, and internal and external complexity. See e.g. [4, 21].

The intention of this paper is to study whether these aspects easily can be included in architecture and design descriptions, and hence help in the understanding of some non-functional aspects between releases. A model is of little value unless it is understood. Once semantics and syntax have been established, there is a ground for understanding a model. Is this enough or can we increase the intuitive understanding of a model, i.e. can we organize a model in such a way that our cognitive processes shaped by prior experience and knowledge help us interpret the model more easily? This is basically the approach taken in the pattern community with respect to functional behaviour. Pattern efforts, e.g. [7, 12] help us traverse the path from the problem domain to the solution domain. All of the pattern approaches require prior knowledge of the patterns to be truly effective in terms of how easily a particular pattern is recognized in models of source code or in the source itself.

The objective of this paper is also to take a cognitive approach based on previous experience and knowledge. The focus is, however, on some quality-related aspects rather than on functional aspects. This paper presents an experiment where the intuitive understanding various representations for size, relationships, internal and external complexity is evaluated. The basic hypothesis of the study is that it is possible to decorate or extend existing architecture and design description techniques fairly easily to include these four aspects related to quality. By decorating models instead of creating new models, we adhere to Baker's and

Eick's proposition that a visualization should adhere to the structure of the software [2], which in this case is represented by an existing architecture model of the software.

In the experiment different ways of representing four aspects are presented to subjects and the intuitive understanding of the representation is evaluated. From the experiment, it can be concluded that different individuals have the same intuitive understanding of the representations evaluated. This shows that it should be possible to include intuitive representation for these four aspects to help in the understanding of software systems between new versions of the software. This is feasible since it is easy to add information about these aspects as the software is being released. The information provides then valuable input to forthcoming versions of the software.

This paper is organized as follows. Section 2 defines research questions, variables and hypothesis. Section 3 discusses the methods used to conduct the experiment. Section 4 presents an analysis of collected data. Section 5 shows some possible applications of the results. Finally, section 6 summarizes the paper.

2. Experiment definition

In this experiment, it is analysed how software developers can be supported during evolution of a software system. The objective of the experiment is to evaluate whether some relationships and quality related issues can be represented easily and intuitively in software architecture or design description. Inspired by e.g. [10, 11], it is believed that different two-dimensional visual representations better express some software engineering concepts in terms of how intuitive the mapping between the representation and the concept is. The representation that is the most intuitive representation of a software engineering concept is said to have the minimum accessibility weight. For example, in Figure 1 there are two symbols, that both represent that smoking is not allowed. The left symbol is a more intuitive representation of the concept. Out of the two symbols, the left symbol can be said to have the minimum accessibility weight - no previous training or information is needed to understand what the symbol represents. By ensuring minimum accessibility weight in models, the need for training and manuals is minimized.

Four research questions are formulated:

Fig. 1. Two symbols that represent the concept of no smoking.

RQ1 How do we best show that component A controls the operation of a component B? That a component controls another component is referred to as the 'CC' relationship.

RQ2 How do we best show that it is more complex to modify the internals of component A than it is to modify the internals of component B? This is referred to as the internal complexity relationship ('IC').

RQ3 How do we best show that the externally visible interface of component A is more complex than the externally visible interface of component B? This is referred to as the external complexity relationship ('EC').

RQ4 How do best show that the size of component A is larger than the size of component B? This is referred to as the component size relationship ('CS').

All of the research questions refer to an intuitive mapping between a concept used within software engineering and a visual representation. There are a large number of classes of spatial concepts used for visual representation, e.g. [13] lists 51 spatial concepts and indicates that there are many more. This experiment investigates only a few visual representations, that have been chosen because they share three properties: i) They are suitable both for paper and screen representation, ii) They can be combined with many existing architecture and design diagrams and iii) The concepts can be used concurrently to decorate an existing diagram with several new properties. The entities studied for the different relationships are shown in Figure 2. Representations 1 to 4 are assessed as good candidates for the CC relationship, i.e. possible indications of that component A controls component B. Representations 5 to 10 are assessed for the EC, IC and CS relationships. For example, it is possible to evaluate whether representation 6 is a better indicator of that component A is larger (CS relationship) than component B, than, for example, representation 9.

From the research questions, a number of hypotheses are derived. These are described in Table 1. The hypotheses all indicate that we believe that there is a representation that is better than the other ones.

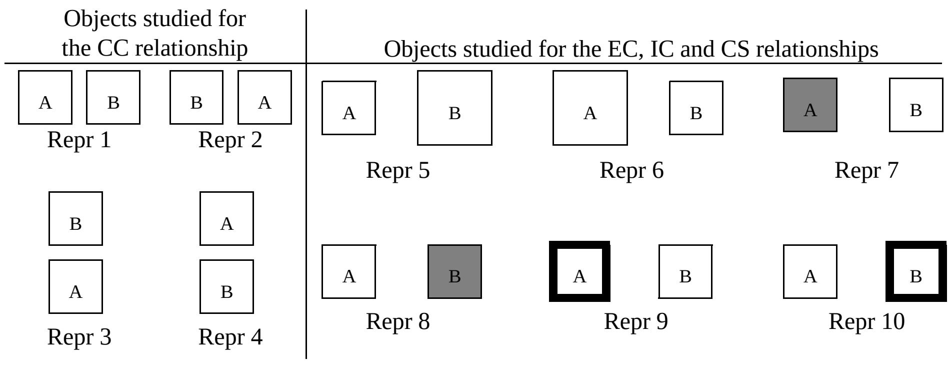

Objects studied for the CC relationship
Objects studied for the EC, IC and CS relationships
A
B
Repr 1
B
A
Repr 2
B
A
Repr 3
A
B
Repr 4
A
B
Repr 5
A
B
Repr 6
A
B
Repr 7
A
B
Repr 8
A
B
Repr 9
A
B
Repr 10

Fig. 2. Objects studied

Table 1. Hypotheses. All hypotheses are assessed at the $p = 0{,}05$ significance level. $m, n \in \{1, 2, 3, 4\}$, $m \neq n$, $p, q \in \{5, 6, 7, 8, 9, 10\}$, $p \neq q$

RQ	Name	Definition
RQ1	$H_{CC,1}$	$Avg_{CC,\text{ Repr n}} \neq Avg_{CC,\text{ Repr m}}$ There is some representation that has minimum accessibility weight in terms of how well it represents that software component A controls software component B.
	$H_{CC,0}$	$Avg_{CC,\text{ Repr n}} = Avg_{CC,\text{ Repr m}}$ Null hypothesis: There is no statistical difference between the representations that may represents that software component A controls software component B. Explanation of variables: The average weight assigned to representation n regarding how well it represents the “A controls B” relationship (CC) is equal to the average weight assigned to representation m for the same relationship.
RQ2	$H_{IC,1}$	$Avg_{IC,\text{ Repr p}} \neq Avg_{IC,\text{ Repr q}}$ There is some representation that has minimum accessibility weight in terms of how well it represents that software component A is internally more complex than software component B. Null hypothesis: $H_{IC,0}$.
RQ3	$H_{EC,1}$	$Avg_{EC,\text{ Repr p}} \neq Avg_{EC,\text{ Repr q}}$ There is some representation that has minimum accessibility weight in terms of how well it represents that software component A is internally more complex than software component B. Null hypothesis: $H_{EC,0}$.
RQ4	$H_{CS,1}$	$Avg_{CS,\text{ Repr p}} \neq Avg_{CS,\text{ Repr q}}$ There is some representation that has minimum accessibility weight in terms of how well it represents that software component A is larger than software component B. Null hypothesis: $H_{CS,0}$.

3. Method

3.1 Introduction

The strategy used to answer the research questions is an experiment in a laboratory environment [26]. For data collection, questionnaires have been used. The method used for data collection and analysis is summarized in Figure 3 and described further below.

3.2 Sampling and generalization

From [18] it is known that entry-level programmers work differently from more experienced programmers. In particular, experienced programmers are able to concurrently draw on more sources of information such as different models. In this experiment, it is studied how entry-level programmers and non full-time programmers can be aided. There have been 35 participants in the experiment. 26 of them are university computer science and electronics master students, and 9 of them are Ph.D.s or Ph.D. students. ANOVA tests comparing the two types of participants show no significant difference in the results from the two types of participants. Therefore, all participants are treated as one homogeneous group, despite that the participants have different exposure to various tools, technologies and methodologies. All participants have had some exposure to object-oriented methods prior to the experiment. We believe that the way the objects studied are perceived is not changed much by the increased time pressure and competitiveness that may exist in industry as compared to a student/ university environment. Given the background of the participants, it should be possible to generalize to at least entry level programmers due to the student participants, and probably also to somewhat more experienced designers as there is no significant difference in the results from students and Ph.D.s/Ph.D. students.

3.3 Data collection

Data collection has taken place in a laboratory setting. Data collection took place at three different occasions. The same introduction to the experiment was given to all participants. This introduction made sure that all participants knew how to fill in the questionnaires. It was also made sure that all participants knew that they were not to think more than 15 seconds for each question. This was to ensure that the initial understanding of each question was captured. No hypotheses regarding the outcome of the experiment were presented to the participants.

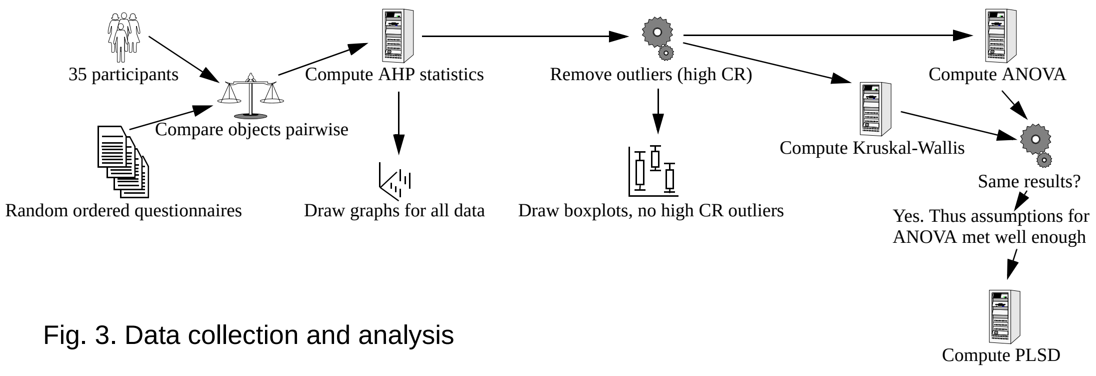


Fig. 3. Data collection and analysis

Four different questionnaires have been used. Each of the questionnaires addressed only one research question. For each of these four questionnaires, there have been three different versions. The ordering of questions in each version has been different, but the set of questions has been equal across all versions. Participants were assigned one of each questionnaire in random order. The version used of each questionnaire was also randomly assigned. Thus, the order of questions answered is random for each participant.

3.4 Analysis

The main statistical analysis method applied to the quantitative data collected through the questionnaires is the Analytical Hierarchical Process (AHP) [15, 23]. The AHP was developed to improve decision-making through giving priority to items pair-wise, rather than prioritizing all items at once. Thus the method simplifies complex decision-making. The output from an AHP analysis is a value denoting the relative weight of items compared. In this paper, the AHP is used to compute the relative accessibility weight of items denoted objects. A number of tests have been used[1] on the output of the AHP, notably the Kruskal-Wallis tests [17], Analysis of Variance (ANOVA) [20] and Protected Least Significant Difference tests (PLSD) [20]. The Kruskal-Wallis test is non-parametric and does not make any assumptions of the distribution or variance of data collected. This test is used to initially get a picture of the data analysed. The ANOVA makes two assumptions: The variance in the groups being compared must be equal, and the data must be distributed according to a normal distribution. If both a Kruskal-Wallis test and an ANOVA show the same result, PLSD tests have been used as it is assumed that the preconditions for the ANOVA have been met well enough. PLSD tests have been used to identify what contributes the most to the results of an ANOVA.

The AHP has an additional feature. Using AHP, it is possible to see how consistent answers are. For example, if a participant claims that item A is more important than item B, and that item B is more important than item C, and that both item A and item B is much more important than item C, there is high consistency in answers. On the other hand, if the participant claims that A is more important than B, and that B is more important than C, then answers are not consistent if the participant also considers item C to be more important than item A. The consistency is measured by computing a consistency index (CI) and adjusting it according to the number of different items compared, to an adjusted index (CR). By looking at the CR, it is possible to objectively identify respondents who have either not put a lot of thought into the answers, or have not understood the questions, or have not enough knowledge to compare the items to prioritize. A low CR value is a sign of high consistency in answers given. As a rule of thumb, answers with a CR > 0.40 are treated as outliers. For details in how to compute the relative weight and CI and CR, see e.g. [15].

1. SPSS 8.0 running on Windows NT has been used for all tests except the AHP.

3.5 Threats and validity

The validity of the findings reported depends to a large extent on how well threats have been handled. Four types of validity threats [9] are analysed: Conclusion validity, construct validity, internal validity and external validity. The relationship between these is illustrated in Figure 4.

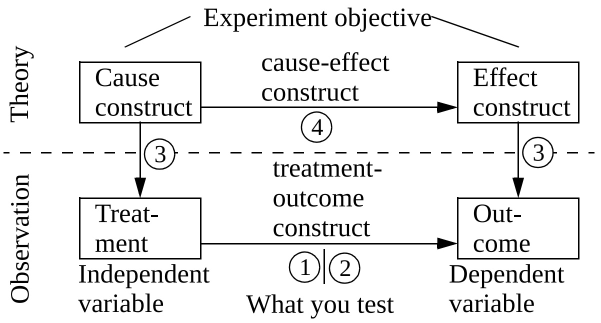


Fig. 4. Experiment principles as described by [26]

Conclusion validity (marked 1 in Figure 4) concerns the relationship between the treatment and the outcome. Care has been taken not to violate any assumptions made by the statistical tests. The questions used in the questionnaires have been easy to answer, and the participants were trained in how to answer the questions. Therefore, there is high reliability in measures. As the data collection took place at three different occasions, there has been a risk that the reliability of the implementation of the experiment could be degraded. This threat has been addressed by using a highly standardized experiment introduction. By collecting data at different occasions, random irrelevancies in the experimental setting have been accounted for. The set of respondent is fairly homogenous, in terms of that there is an overlap in previous experiences in courses studied at the university. On the other hand, there is no statistically significant difference between the student participants and the non-student participants. Therefore, the present random homogeneity (or heterogeneity) of respondents should not affect the conclusion validity much.

Internal validity (2) concerns matters that may affect an independent variable's causality, without the knowledge of the researcher. History effects, i.e. how previous events affect participants, have been taken into account in several ways. First, there is some heterogeneity in the group of respondents. This balances long-term history effects. Secondly, the data collection has taken place at three different occasions. This accounts for effects of recent events. Thirdly, respondents answered questions in different order by using different versions of four questionnaires answered in random order. The latter ensures that history effects from the last few questions are accounted for. As data collection took no more than an hour, there should be no maturation effects. Testing effects have been minimized by making sure that there was no gain or loss in how questions were answered. No participants left the experiment, i.e. there has been no mortality.

Construct validity (3) concerns whether we measure what we believe we measure. Constructs used in the

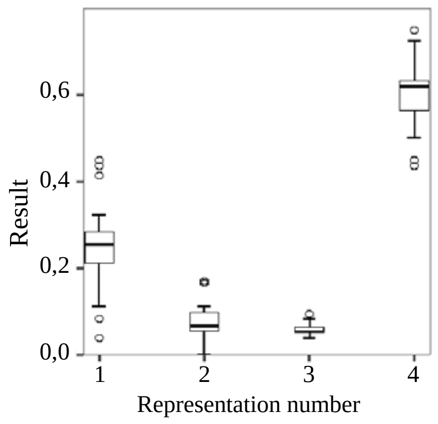


Fig. 5. Boxplot for the CC relationship

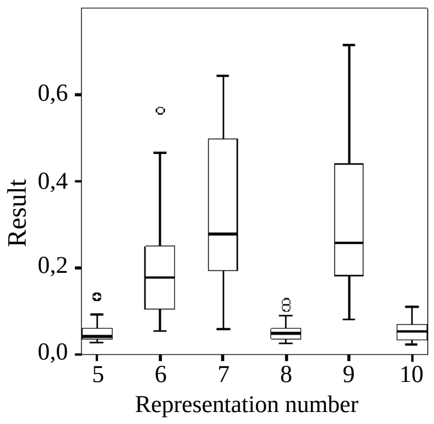


Fig. 6. Boxplot for the IC relationship

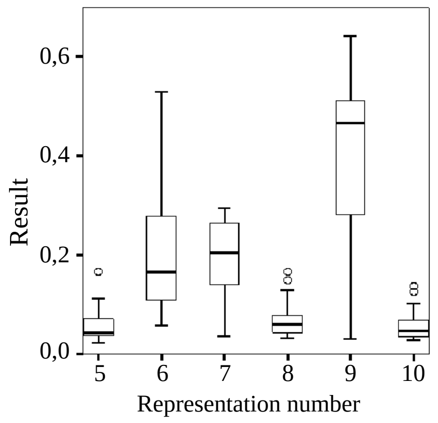


Fig. 7. Boxplot for the EC relationship

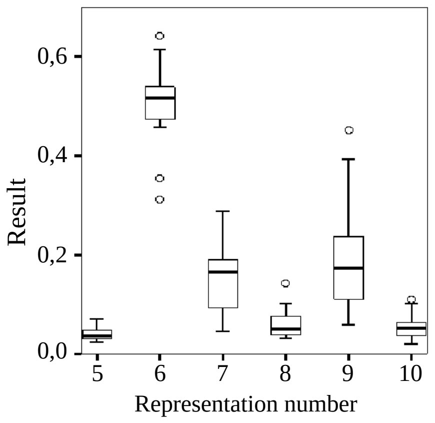


Fig. 8. Boxplot for the CS relationship

questionnaires have been well defined. The use of the scale used to compare objects has been explained to all participants. The use of the AHP and using the consistency ratio to identify outliers is a way of reducing the mono-method bias by making each participant compare each object studied to all other objects. Thus, the results from each respondent are not dependent on a single answer. There is always a risk that respondents guess the hypotheses at hand. By making sure questions are answered in different orders, we balance both the threat of hypotheses guessing as well as the threat of interaction between different questions and objects.

External validity (4) concerns generalization of the findings to other contexts and environments than the one studied. All participants have been active at a university, either in teaching or studying. The teachers have varying degree of industrial background. First, there is no statistically significant difference between the teachers and the students. Secondly, which may be more important for the external validity: Students leave the university for industry. Student participants are close to the end of their university education (M.Sc. level). It is believed that possible change in competitiveness and time pressure that may be different in industry and a university setting does not affect the perception of the objects studied. Therefore, results should be valid at least for the first few years in industry. Another threat is that the experiments compare the objects in Figure 2 in isolation, not in complex combinations such as shown in Figure 11. It is possible that this affects the outcome, and further studies is needed to verify that understanding does not change in complex combinations. A threat related to the understanding of software, is the wide range of comprehension strategies used by different individuals [18, 25]. The objective of this paper is not to generate a silver-bullet solution, rather a particular set of comprehension issues are investigated. The biggest threat to external validity we can think of, is the prior exposure by other populations to particular mappings between the software engineering concepts and the visual representations studied. This could possibly be the case if some tools have been used before.

4. Results and analysis

In this section, results from the experiment are presented. First, the analysis for the CC relationship is presented in some detail, to explain how analysis has taken place. In table 2, results for all four relationships are summarized. Finally, in Figure 9 all results are graphically summarized.

Table 2. Summary of analysis for the four relationships studied

	CC	IC	EC	CS
Representations compared	1, 2, 3, 4	5, 6, 7, 8, 9, 10	5, 6, 7, 8, 9, 10	5, 6, 7, 8, 9, 10
Boxplots of all data	Figure 5	Figure 6	Figure 7	Figure 8
Outliers due to CR>40, participant(s) no.	None	2, 20	9	2
H_x rejected by a Kruskal Wallis at a 0.05 significance level	$H_{CC,0}$ rejected	$H_{IC,0}$ rejected	$H_{EC,0}$ rejected	$H_{CS,0}$ rejected
H_x rejected by an ANOVA test at a 0.05 significance level	$H_{CC,0}$ rejected	$H_{IC,0}$ rejected	$H_{EC,0}$ rejected	$H_{CS,0}$ rejected
Representation with best average minimum accessibility weight	Repr 4	Repr 7	Repr 9	Repr 6
Representation with next best minimum accessibility weight	Repr 1	Repr 9	Repr 7	Repr 9
PLSD indicates difference at 0.05 significance level between the two best representations	Yes	No	Yes	Yes
Suggested mapping with minimum accessibility weight	Repr 4	Repr 7[a]	Repr 9	Repr 6
Answers research question	RQ1	RQ2	RQ3	RQ4

a. Representation 9 is significantly better at representing the EC relationship than representation 7. Therefore, representation 9 is chosen for the EC relationship. There is no statistically significant difference between representations 7 and 9 for the IC relationship, but since representation 9 already is accounted for, and representation 7 is the best on average, it is chosen to represent the IC relationship.

Detailed analysis of the CC relationship

$H_{CC,1}$ is tested. The participants were asked "Does object 1 or object 2 indicate the stronger the fact that software component A controls software component B?". The objects referred to are representations 1, 2, 3, and 4 in Figure 2. The individual participants' results and a boxplot are shown in Figure 5. From Figure 5, it is seen that representation 4 seems to be the best and that the next best is representation 1. Thus, the next step is to evaluate if the differences seen in Figure 5 are statistically significant. At this stage, it is also noted that no participants had a CR>0.40, so there are no outliers.

Both a Kruskal-Wallis test and an ANOVA test show that there is a statistically significant difference at the 0.005 level in how designers rank the accessibility-weight of the four objects studied.

Repr4 is on average the best representation of the CC relationship. Since an PLSD shows that $Avg_{CC,\ Repr4}$ is statistically different at the 0.005 level from the next best representation of the CC relationship, Repr1, $H_{CC,0}$ can be rejected. The conclusion is that Repr4 is the representation of those studied that represents the controlling/controlled relationship with the smallest accessibility weight. This answers RQ1 and addresses the H_{CC} hypotheses.

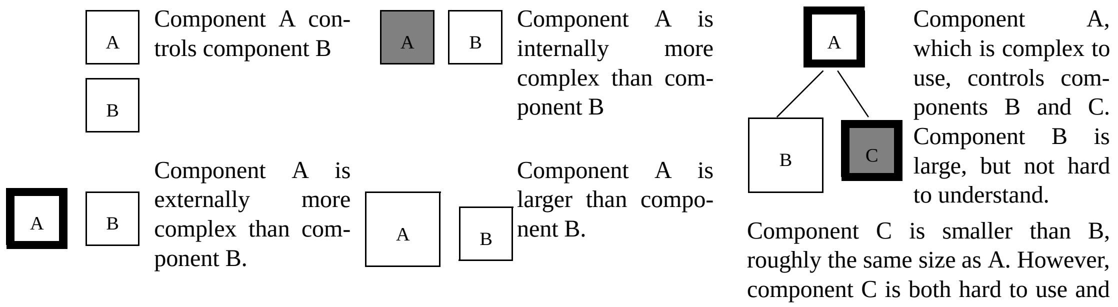
This analysis and the corresponding analysis for the other relationships are summarized in table 2. In addition, box plots for the other three hypotheses are shown in Figures 6-8.

The mappings with the minimum accessibility weight are illustrated in Figure 9. There is also an example of how the properties can be combined together.

5. Application of results

In order to illustrate how the results can be applied, a scenario is presented below.

A Software Development Team (SDT) is presented with a static architecture level diagram at two aggregation levels [6] as illustrated in Figure 10. The software illustrated



Component A controls component B

Component A is internally more complex than component B

Component A is externally more complex than component B.

Component A is larger than component B.

Component A, which is complex to use, controls components B and C. Component B is large, but not hard to understand.

Component C is smaller than B, roughly the same size as A. However, component C is both hard to use and hard to understand the internals of.

Fig. 9. Results and example of combined use

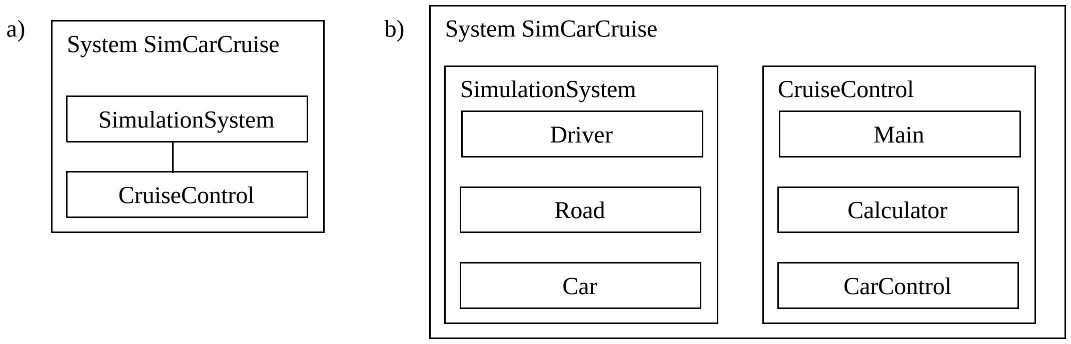


Fig. 10. Cruise-control without understandability mode. a) is at a high aggregation level, b) is a lower aggregation level

is parts of an industrial cruise control system, with the ability to simulate the car, the driver and the road. The SDTs task is to make the system behave better in very steep uphill and downhill conditions. The SDT does not have any previous knowledge of the software.

Seeing Figure 10a, the SDT draws a number of immediate conclusions: The Simulator should be changed, so that higher degrees of road elevation could be handled. Also, the CruiseControl should be changed to contain better algorithms. The team looks into architecture models at a lower aggregation level as illustrate in Figure 10b. There are no more fine-grained static models available to the team. The team decides that the Road, Calculator and CarControl components are likely to be changed. However, without experience from cruise-control systems, it can be hard for the team to predict for example change effort in each component, and where the largest risks for errors are. This is important to know, since in some systems, a small number of components are likely to account for a large amount of externally visible errors [4, 14]. Therefore, the SDT enters the understandability mode of their software development tool. This is illustrated in Figure 11.

Using the information in Figure 11, the SDT makes a number of new decisions: The Road is a small component. However, as its internals are hard to understand, ample time for understanding the component, as well as reviewing and testing the component should be planned for. The Calculator component is small and not internally complex. Therefore, less time is allocated into understanding that component. Finally, the CarControl component is both large and complex (internally and externally). Therefore, two developers are devoted entirely to this component. By these adjustments to the development process and organization, it is likely that the fault-prone component types identified by [22] are effectively handled. Thus this scenario exemplifies how the representations suggested by this experiment can aid in making higher quality software.

In this scenario, the visual representation suggested for the controlling/controlled relationship is not in use. The reason for this is that it is believed that this relationship best should be modelled at design-time, by humans, because it can be very hard to determine which component controls another component from just looking at machine readable code. The other properties can possibly be automatically computed.

6. Summary

The presented experiment is formulated based on the belief that it should be possible to represent some quality related aspects clearly in architecture and design models. To the best of our knowledge, no architecture or design representation include representations to capture control, size, external and internal complexity. Thus, the objective of the experiment was to evaluate whether it would be possible to augment some existing architecture and design models with this type of information, in an intuitive way.

Ten different representations have been evaluated, and the results from the experiment have shown with statistical significance, in most cases, that it is indeed possible to represent non-functional aspects in architecture and design models. The evaluated representations can easily be introduced as an integral part in most graphical architecture and design descriptions, which differ from for example [8] where qualitative information is displayed as separate

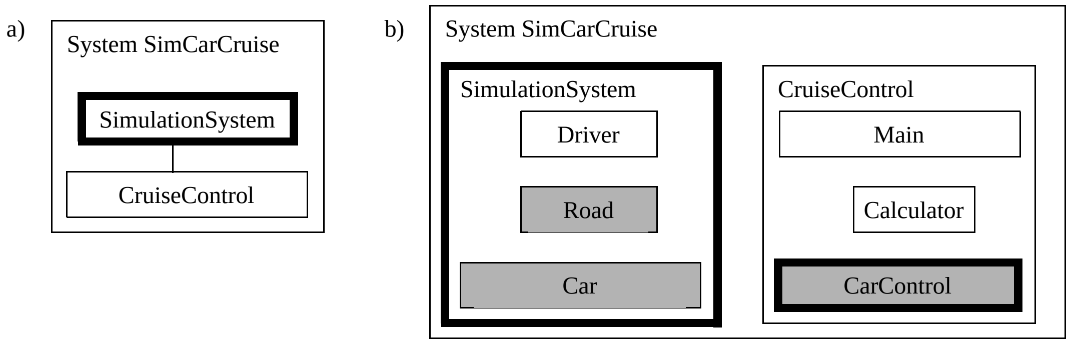


Fig. 11. Cruise-control in understandability mode. a) is at a high aggregation level, b) is a lower aggregation level

models generated from text-oriented code.

The main value of the results are is in software evolution. Representations of the evaluated types can support architecture and design understanding during evolution and maintenance. In addition to the experiment, it has been illustrated through an example how tool support can be enhanced to include representations of non-functional aspects. For example, [2, 3] use component area to illustrate component size. The findings in this study strengthen the idea that this is a sensible representation.

In summary, the experiment indicates that representations can be found that can be used when software is evolving and we would like to stay in control of the evolution. The representations suggested can be viewed as warnings signs together with, for example, methods for classifying software components. Further work includes comparing the representations suggested by this study in more complex environments as well as the use in large scale software systems.

References

[1] Alhir, S.S. *UML in a Nutshell*. O'Reilly & Associates, Sebastopol, CA, USA, 1998.

[2] Baker, M.J., and Eick, S.G. "Space-Filling Software Visualization", *Journal of Visual Languages and Computing* 6(2), June 1995.

[3] Ball, T.J., and Eick, S.G. "Software Visualization in the Large", *IEEE Computer* 29(4), Apr 1996.

[4] Basili, V.R., and Perricone, B.T. "Software Errors and Complexity: An Empirical Investigation", *Comm. of the ACM*. Vol. 27, No. 1, Jan 1994.

[5] Bratthall, L., and Runeson, P. "Architecture Design Recovery of a Family of Embedded Software Systems", Proc. TC2 First Working IFIP Conference on Software Architecture, San Antonio, TX, USA, 1999.

[6] Bratthall, L., and Runeson, P. "A Taxonomy of Orthogonal Properties of Software Architecture", Proc. Second Nordic Workshop on Software Architecture. Ronneby, Sweden, Aug. 1999.

[7] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. Pattern-Oriented Software Architecture: *A System of Patterns*, John Wiley & Sons, Great Brittain, 1996.

[8] Chuah, M.C., and Eick, S.G. "Glyphs for Software Visualization". 5th Int'l Workshop on Program Comprehension, IEEE Computer Society Press, Dearborn, Michigan, USA, May 1997.

[9] Cook, T.D., and Campbell, D.T. *Quasi-Experimentation - Design and Analysis Issues for Field Settings*, Houghton Mifflin Company, 1979.

[10] Egenhofer, M. J., and Mark, D.M. "Naïve Geography", In Proc. Spatial Information Theory: A Theoretical Basis for GIS COSIT'95. Springer Verlag, Germany. 1995

[11] Freksa, C. "Spatial and Temporal Structures in Cognitive Processes", *Foundations of Computer Science: Potential - Theory - Cognition*, Springer Verlag, Germany, 1997.

[12] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[13] Habel, C., and Eschenbach, C. "Abstract Structures in Spatial Cognition", *Foundations of Comp. Science. Potential - Theory - Cognition*, Springer Verlag, Berlin, Germany, 1997.

[14] Haglund, S., and Persson, J. "A Process for Reverse Engineering of AXE 10 Software", Proc. 6th Reengineering Forum, Fierenze, Italy, Mar 1998.

[15] Karlsson, J., and Ryan, K. "A Cost-Value Approach for Prioritizing Requirements", *IEEE Software*, Vol. 14, No. 5, Sep./Oct. 1997.

[16] Kruchten, P., "The 4+1 View Model of Architecture", *IEEE Software*, vol. 12, no. 6, Nov. 1995.

[17] Kruskal, W.H., and Wallis, W.A. "Use of Ranks on One Criterion Variance Analysis", *Journal of the Americal Statistical Association*, Vol. 47. Corrections appear in Vol. 48, 1952.

[18] von Mayrhauser, A., and Vans, A.M. "Comprehension Processes During Large Scale Maintenance", Proc. 16th Intl. Conf. Software Engineering. Sorrento, Italy, May 1994.

[19] von Mayrhauser, A, Wang, J., Ohlsson, M.C., and Wohlin, C. "Deriving a Fault Architecture from Defect History", Proc. Intl. Symposium on Software Reliability Engineering, ISSRE'99.

[20] Montgomery, D.C. *Design and Analysis of Experiments, Third Edition*, John Wiley & Sons, New York, 1991.

[21] Ohlsson, M.C., von Mayrhauser, A, McGuire, B., and Wohlin, C. "Code Decay Analysis of Legacy Software through Successive Releases", Proc. IEEE Aerospace Conference. Colorado, USA. March 1999.

[22] Ohlsson, M.C., and Wohlin, C. "Identification of Green, Yellow and Red Legacy Components", Proc. Int'l. Conf. on Software Maintenance. Bethesda, Washington D.C., USA, Nov. 1998.

[23] Saaty, T.L., *The Analytic Hierarchy Process*, McGraw-Hill, New York, USA, 1980.

[24] Soni, D., Nord, R., and Hofmeister, C., "Software Architecture in Industrial Applications", Proc. 17th Int'l. Conf. Software Eng., Apr 1995.

[25] Storey, M.D., Wong, K., and Müller, H.A. "How do Program Understanding Tools Affect How Programmers Understand Programs?", Proc. Fourth Working Conf. Reverse Engineering. Amsterdam, The Netherlands, Oct. 1997.

[26] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., and Wesslén, A. *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers, Boston, MA, USA, 1999.