Empir Software Eng (2018) 23:3578–3625
https://doi.org/10.1007/s10664-017-9583-4

Supporting the analyzability of architectural component models - empirical findings and tool support

Srdjan Stevanetic[1] · **Uwe Zdun**[1]

Published online: 29 March 2018
© Springer Science+Business Media, LLC, part of Springer Nature 2018

Abstract This article discusses the understandability of component models that are frequently used as central views in architectural descriptions of software systems. We empirically examine how different component level metrics and the participants' experience and expertise can be used to predict the understandability of those models. In addition, we develop a tool that supports applying the obtained empirical findings in practice. Our results show that the prediction models have a large effect size, which means that their prediction strength is of high practical significance. The participants' experience plays an important role in the prediction but the obtained models are not as accurate as the models that use the component level metrics. The developed tools combine the DSL-based architecture abstraction approach with the obtained empirical findings. While the DSL-based architecture abstraction approach enables software architects to keep source code and architecture consistent, the metrics extensions enable them, while working with the DSL, to continuously judge and improve the analyzability of architectural component models based on the understandability of their individual components they create with the DSL. Provided metrics extensions can also help in assessing how much each architectural rule used to specify the DSL affects the understandability of a component which enables for instance finding the rules that contribute the most to a limited understandability. Finally, our approach supports change impact analysis, i.e., the identification of changes that affect different analyzability levels of the component models. We studied the applicability of our approach in a case study of an existing open source system.

Communicated by: Richard Paige

✉ Srdjan Stevanetic
srdjan.stevanetic@univie.ac.at

Uwe Zdun
uwe.zdun@univie.ac.at

[1] Software Architecture Research Group, University of Vienna, Währingerstraße 29, 1090 Vienna, Austria

Keywords Software architecture · Architectural component models · Understandability · Software metrics · Empirical study · Architecture evolution

1 Introduction

In the process of software systems development software architecture represents a key artefact that affects all later activities such as design and implementation and plays a crucial role in achieving the desired software qualities (Losavio et al. 2003). Software architecture focuses on a high level view of a software system and it is defined as : "the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them" (Bass et al. 1998).

According to the software architecture community, an architectural description can comprise multiple views concentrating on one of many system concerns, such as logical, implementation, deployment, process, or architectural knowledge view, and from the viewpoint of different stakeholders, such as end-users, developers, project managers, and business analysts (Kruchten 1995; Clements et al. 2003). Architectural component and connector models (or shortly component models), that are part of the implementation view, are frequently used as a central view of the architectural descriptions of software systems (Clements et al. 2003). Component models represent high-level abstractions of the system implementation and are often considered to contain the most significant architectural information (Clements et al. 2003). In this view, components could refer to different system entities such as processes, objects, clients, servers, data stores, modules, subsystems, etc., while connectors represent the interaction mechanisms between components (Clements et al. 2003). In this article, we consider a component more in the sense of software modules by adopting the definition of Clemens et al., i.e. a component represents an implementation unit of software that provides coherent unit of functionality at the first level of decomposition in the system (Clements et al. 2003). This definition is adopted because our work focuses on the understandability of component models which mainly relates to understanding a functional decomposition of the system and the effect of modifying the system functionalities, i.e. the impact analysis. Please note that component decomposition can be made independent of the functionality type implemented in a component. For example, a decomposition can consider both technical functionalities (e.g. components for file-access or network connection) and business functionalities (e.g. components for savings or accounts). Since a component in a component model represents a high-level abstraction of the entities in the source code of the system, it can be broken down into (i.e., is refined by) more fine-grained, technical components or classes that realize the component in the technical design or implementation of the system. In the context of object-oriented software systems that we focus on, a component usually groups a set of source code classes and/or packages with similar functionalities, while a connector could represent any kind of dependency between classes like method calls, fields access, etc.

Understandability is one of the most important characteristics of software quality (Pacione et al. 2004). The difficulty of understanding the software system limits its reuse and maintenance. Boehm defined software understandability as a feature of software quality which means ease of understanding software systems (Boehm 1978). In the context of component models, understandability refers to understanding the functionalities of individual components together with the functional relatedness among them (Dugerdil and Niculescu 2014). Understandability is a critical aspect for the component models, as their

main purpose is to " ... enable designers to abstract away fine-grained details that obscure understanding and focus on the "big picture:" system structure, the interactions between components, ..." Oreizy et al. (1999). This, however, is not possible if the given models themselves and/or the links to other design and code artefacts are hard to understand.

In our previous work (Stevanetic and Zdun 2016), we examined the relationships between the effort required to understand a component, measured through the time that participants spent on studying a component, and the hierarchical quality metrics originally designed to assess the understandability of the modular design of an object-oriented software system (Hwa et al. 2009). Those metrics refer to 6 design properties found to have an impact on the understandability of the modular design of a system: size, complexity, encapsulation (i.e. information-hiding), coupling, cohesion and modular abstraction. In the same study, we have further examined the impact of personal factors (i.e. the participants' experience and expertise), and compared the efficiency of both personal and system related factors (metrics) with the prediction models obtained in our previous studies (Stevanetic and Zdun 2014a, b). In another study reported in a position paper (Stevanetic et al. 2014), we presented a tool for supporting software evolution by integrating a DSL-based architecture evolution approach with our empirically evaluated understandability metrics. In this article, we provide: 1) an extended description of the results obtained in our previous work (Stevanetic and Zdun 2016) consisting of more detailed description of the studied metrics and applied statistical techniques as well as more detailed explanations and discussions of the obtained results, 2) a new metric for measuring the analyzability of component models based on the integration of our empirical evaluations and the existing work on the analyzability related metrics proposed by Bouwers et al. (2011), and 3) significant tool extensions compared to our previous work reported in a position paper (Stevanetic et al. 2014) including the realization of the new analyzability metric by supporting how much each of the architectural rules used to specify a DSL-based architectural abstraction specification contributes to the understandability of components and enabling change impact analysis, i.e. the identification of changes in the system that affect different analyzability levels of the component models.

The results of our empirical analyses show that the hierarchical understandability metrics can predict the understandability with high practical significance. On the one hand, the obtained prediction models are significantly better then the models obtained using the graph based metrics (examined in Stevanetic and Zdun 2014a), the package based metrics (examined in Stevanetic and Zdun 2014b) or the models that use the participants' experiences as predictors. On the other hand, those models are not significantly different or worse in prediction from the models that combine both the system related metrics (the graph based, package based, and hierarchical understandability metrics) and the participants' experiences. This means that, from all studied predictors, the system related metrics (i.e. the hierarchical understandability metrics) are enough to consider for the prediction. We also find that the participants' experiences are important and can predict a significant amount of variance in the data but the obtained models are not as accurate as the models that use the metrics related to the software system itself (concretely the hierarchical understandability metrics). Regarding the tool support, we demonstrate in a case study how it can be used to create component models with appropriate analyzability level by incrementally improving an initial component model of the system. In addition, we show how the tool can be used for change impact analysis, i.e for detecting the changes that exist between different component models that affect their different analyzability levels.

This article is organized as follows: In Section 2, we discuss the related work. In Section 3 we describe the study design. Section 4 describes the statistical methods we applied and the

analysis of our data. In Section 5 we discuss the threats to validity. Section 6 describes the tool we developed together with a case study on how the tool can be utilized in a practical context. In Section 7 we conclude and discuss future directions of our research.

2 Related work

So far very few studies investigate the empirical evidence on the architectural understandability. One of them examines the influence of package coupling on the understandability of the software systems (Gupta and Chhabra 2009), while another one examines the relationships between some package-level metrics and package understandability (Elish 2010). None of the studies examines the understandability of architectural components. In this section we discuss the existing works in several fields closely related to our work.

2.1 Measuring the understandability

In the work by Patig (2008) the variables and tasks that have been proposed by cognitive psychology or applied in computer science to test understandability are extracted. Those variables and tasks are summarized in Fig. 1 and they represent a theoretical framework for investigations on understandability. The variables have been theoretically justified by the authors who used them. In our case, the independent variables represent the metrics that we collected (in the work by Patig they are related to abstract/concrete syntax and therefore this part of the figure is adapted from the original one). The dependent variable in our case is the understandability of components. As we see from Fig. 1, different measures can be used to quantify the dependent variable(s) such as frequency (the number of correct answers), selection (which of several answers participants choose), response latency (how quickly participants reacts), response duration (how long participants deal with a task), and amplitude (measuring the strength of response, i.e. brain activities in performing a task). In our case we measured the correctness of the answers and the time that participants spent on resolving the questions. Regarding the comprehension tasks the participants of an experiment need

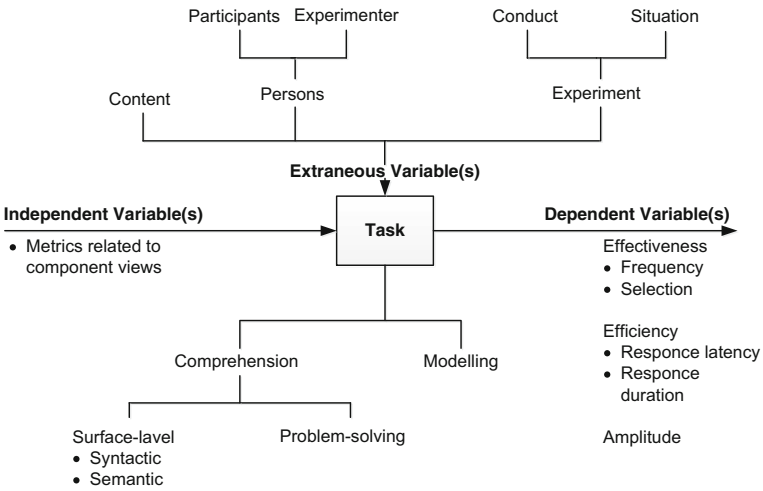


Fig. 1 Theoretical framework for investigations on understandability (adapted from Patig 2008)

to answer an appropriate set of questions. If the questions are related to the syntax of the model (constructs of the model) the task is called syntactic. If the questions are related to the understanding of the context described the task is called semantic. Both of these two types of tasks are related to surface level understanding. In problem-solving tasks that address deeper understanding participants have to resolve whether and how certain information can be extracted from a model. In our case the problem-solving tasks are more suitable because the participants have to understand not just the component models themselves, i.e. how the components interact in the model, but also the relations between them and the concrete system implementation. Modelling tasks are used more for measuring the general ease of use of some notation and therefore they are not suitable for our case.

In the work by Patig all proposed dependent variables are externally measured in terms of using some external means like the time that participants spent on answering the questions or the percentage of the correct answers on those questions. Beside the external means it is also possible to use the participants subjective ratings in the measurement process. In the context of model understandability Moody proposes three ways how to assess understandability: the model user's rating of model understandability, the ability of users to interpret the model correctly, and the model developer's rating for model understandability (Moody 1998). The first and the third way are based on the subjective ratings of users/developers. However, Lindland et al. explain that the ability of model users to interpret the model correctly is the best operational test whether the model is actually understood rather than whether it is understandable (Lindland et al. 1994; Moody 1998).

2.2 Architecture and design metrics and their empirical evaluations

There exist plenty of software metrics for measuring the system's architecture, architectural components, and other high level software artefacts and structures (packages, modules, graph-based structures). For example, metrics related to components and component models measure different attributes like size, coupling, cohesion, and dependencies of components as well as the complexity of the whole component models (Sharma et al. 2009; Sartipi 2001; Sengupta et al. 2011). Regarding the software packages, different metrics that measure size, coupling, stability, and cohesion are proposed (Elish 2010; Gupta and Chhabra 2009; Martin 2003; Gupta and Chhabra 2012). Graph-based metrics measure the complexity of interactions between the graph nodes (Bhattacharya et al. 2012; Ma et al. 2006; Allen et al. 2007). Certain graph-based metrics are evaluated to be useful for measuring large scale software systems that are observed to share some properties that are common for complex networks across many fields of science (Ma et al. 2006). Most of the above given metrics lack of the links to the quality attributes. Stevanetic and Zdun (2015) present a systematic mapping study on software metrics related to the understandability concepts of software architectures with regard to their relations to the system implementation. In this article and the previous ones that empirically investigate the understandability of components, the examined metrics are chosen from the given mapping study and tested in the given context.

There exist several studies that empirically evaluate metrics. In contrast to our work, they usually evaluate the usefulness of a metric for its proposed purpose, but do not test the relationships of specific metrics, as in our case the prediction of the understandability using predictor metrics. Also, none of the studies focuses on architectural component models. Among many others, Basili et al. evaluate object-oriented design metrics as quality indicators (Basili et al. 1996). Albrecht and Gaffney provide one of many examples for a study on development effort metrics (Albrecht and Gaffney 1983). Similarly to our work Moody presents an empirical evaluation of the use of data model quality metrics (Moody 2003). In

this approach a broad set of quality metrics is investigated. The result obtained is that only a few of these quality metrics have an influence on the quality as perceived by the model users. These are the system complexity, the number of data items duplicated in existing systems, the development cost estimation, the reuse percentage, and the number of defects by quality factor.

2.3 Understandability of UML models and process models

There exist a variety of studies in the literature that examine the understandability of different UML models. Some of them examine the layout or visualization aspects of UML models. Purchase et al. (2001) show that certain visualizations are better than the other depending on the kind of comprehension tasks that is used. Criteria and guidelines of how to create effective layout for UML class and sequence diagrams are established in the work by Sun and Wong (2005). They are based on perceptual theories.

Some other studies related to UML model understandability compare the effect of using different UML diagram types (e.g., sequence and collaboration diagrams). For example, Otero and Dolado take different UML diagrams types, sequence, collaboration, and state diagrams, and evaluate the semantic comprehension of the diagrams when used for different application domains (Otero and Dolado 2004).

Some authors investigate the styles and rigor in UML models and how they affect the understandability of the models. For example, Briand et al. (2005) investigate the impact of using OCL (object constraint language) in UML models on defect detection, understandability, and impact analysis of changes. They find that the benefits for the individual activities are modest but the overall benefits of using OCL on the aforementioned activities are significant. None of the aforementioned studies examine the understandability of architectural components, the central high level organizational units of the architectural descriptions of software systems.

The work in the field of process model related metrics emphasize the importance of model characteristics for assessing model understandability. Such metrics measure structural properties of a process model, motivated by prior work in software engineering related to lines of code, cyclomatic number, or object-oriented metrics (McCabe 1976; Chidamber and Kemerer 1994; Fenton and Pfleeger 1998). Soo and Jung-Mo (1992), Nissen (1998), and Morasca (1999) focus on defining metrics. Different metrics have been also validated empirically. Cardoso adapts the cyclomatic number metric for business processes (called it control-flow complexity (CFC)) and proves the correlation of the metric with perceived complexity of process models (Cardoso 2006). Canfora, Rolon, and Garcia analyse understandability as an aspect of maintainability using different metrics of size, complexity, and coupling in their experiments. They identify several significant correlations (Canfora et al. 2005; Aguilar et al. 2007). Some other metrics are related to cognitive research, e.g. Vanderfeesten et al. (2008), and based on concepts of modularity, e.g. Vanhatalo et al. (2007) and van der Aalst and Bisgaard Lassen (2008).

Different empirical validations in the field of process models clearly show that size is an important model factor for understandability, but does not fully determine phenomenons of understanding. It means that additional metrics like structuredness can help to improve the explanatory power significantly (Mendling 2008). In our case, we examine the effect of different metrics, that measure more/less the same concepts as those mentioned for process model understandability (size, coupling, complexity), on understandability of components' functionalities implemented by the corresponding set of source code classes. We also show that the size is not enough to fully determine the understandability and additional properties

need to be taken into account. Similar to our work, Reijers and Mendling (2011) investigate the impact of personal and model related factors on understandability of process models. They show that expert modelers perform significantly better and that the complexity of the model affects understanding. A combined regression model is calculated that permits preliminary conclusions on the relative importance of both groups of factors. They find that personal factors (theoretical knowledge, practical experience, educational background) have a stronger explanatory power in terms of adjusted R^2 than model related factors but they kept the size of the models constant by intentionally selecting models of equivalent size. We also find that the participants' experiences are important as well as the system related metrics but in contrast to the work by Reijers and Mendling, we find that the system related metrics have a significantly stronger explanatory power and even alone can be used for the prediction, i.e. combining them with the experiences does not produce a stronger explanatory power. Furthermore, we take into account the size. Also, all our participants are students and we do not consider experts from industry as it is the case in the previous study.

2.4 Software quality models

To assess design quality different object-oriented software quality models have been proposed and validated in the literature (Chidamber and Kemerer 1994; Bansiya and Davis 2002; Genero Bocco et al. 2005; Harrison et al. 1998; Basili et al. 1996). In those models, software quality is assessed using several software metrics that are used to quantitatively assess design properties such as coupling and cohesion. But those models are insufficient to manage understandability in the high level system representations such as module-view, package-view, or component-view because they capture a software system as the set of classes and their relationships, but not the set of modules, packages or components and their relationships.

Contrary to the given quality models, Bansiya and Davis (2002) proposes a hierarchical quality model for object-oriented design quality assessment (QMOOD) which is able to assess understandability of a system. Their model extends Dromey's quality framework used for building product based quality models (Dromey 1995; Dromey and McGettrick 1992). However, QMOOD can only consider the dependencies between classes in a module without considering the dependencies between classes of different modules as well as a module hierarchy and therefore cannot assess the quality of modular design properly. Sarkar et al. (2008) examine different metrics that can be used to assess modularization quality of a large-scale object-oriented software system. But the authors do not provide relationships between their metrics and the high-level quality attributes. Therefore more investigations are necessary to establish the links between those metrics and high-level quality attributes. Hwa et al. (2009) propose a hierarchical model to assess understandability of modularization in large-scale object-oriented software. They define several design properties, which capture the characteristics influencing on understandability, and design metrics based on the properties, which are used to quantitatively assess understandability. In this article, we use the concepts and metrics defined in the work by Hwa et al. to improve the explanatory power of our previously obtained models on understandability of architectural components.

2.5 Other aspects related to architectural component models

Even though there is a lack of empirical studies on architectural component models understandability, other aspects like fault density and reuse of components have been studied before. In the work by Fenton and Ohlsson the relations between fault density and component

size are examined (Fenton and Ohlsson 2000). Mohagheghi et al. use the historical data on defects, modification rate, and software size to investigate the comparison between software reuse and defect density and stability (Mohagheghi et al. 2004). Malaiya and Denton study the factors that can be used to determine the "optimal" component size with regard to fault density (Malaiya and Denton 2000). They identified component partitioning and implementation as influencing factors. Graves et al. examine the software change history of components in order to create a fault prediction model (Graves et al. 2000). Metrics such as change times, time elapsed since the last changes, and number of changes are used in the model, while size and complexity metrics are not deemed useful. These and similar studies have in common with our one that a link between software quality or desired properties, such as fault density or reuse rate, and component properties, such as size, complexity, or change rate, are made. These studies are different from our one as they examine aspects that can be studied without considering the human participants: They only analyse aspects that can solely be studied using the software systems and their historical data.

A number of authors propose ways to improve the understandability of architectural models through additional models or documentation artefacts. A major research direction deals with documenting architectural decisions and architectural knowledge in addition to component models (Babar and Lago 2009; Jansen and Bosch 2005; Zimmermann et al. 2007). Another major research direction deals with architectural views (Clements et al. 2002; Hofmeister et al. 2000; Kruchten 1995) which enable different stakeholders to view the architectures from different perspectives. Both research directions only complement component models with additional knowledge, but neither of them studied the understandability issues of component models with regard to their relations to the system implementation.

2.6 Architecture abstraction and evolution

There exist several approaches that support the abstraction of the architecture from other system artefacts as well as the architecture evolution. Here, we discuss some of those approaches that are closely related to the approach used in our tool.

Konersmann et al. (2013) describe the ADVERT approach that provides support for software evolution on an architectural level. Their approach is based on two ideas: (1) Maintaining trace links between requirements, design decisions, and architecture elements, and (2) explicitly integrating software architecture information into the code. Contrary to our approach the ADVERT approach assumes that the architecture already exists (is built from the design solutions) and it does not provide architecture level quality checks. Another approach that focuses on architecture evolution is proposed by Barnes et al. (2014). They support the modelling of different evolution paths and allow reasoning about architecture evolution based on these different paths. Cuesta et al. (2013) extends the approach by Barnes et al. by proposing the documentation of architecture evolution using architectural knowledge. These approaches are more focussed on reasoning about architecture evolution while our approach aims at supporting architecture evolution in order to evolve source code and architecture documentation in a synchronized fashion, allowing at the same time architecture quality evaluation.

There exist several approaches that focus on the automatic creation of source code abstractions using automatic clustering. The comparison and review of those approaches and the corresponding clustering measures can be found in the work by Maqbool and Babri (2007). They define a number of clustering algorithms groups and compare their performance using different open source projects. The results show which approach works good

for which application but no conclusions regarding the overall effort necessary to correct the automatic clustering are drawn. Contrary to all these approaches our DSL-based approach is semi-automatic, enables the checking of design constraints during the abstraction process, provides traceability between source code and models and focuses on the evolution of the architecture (having an "up-to-date" architecture that reflects the source code) rather then the recovery of architecture. Also, our approach provides quality checking of the generated architectural abstractions based on the corresponding empirical evaluations.

Egyed (2004) proposes an approach for model abstraction based on traceability information and abstraction rules. The author identified 120 abstraction rules for the example of UML class models, which need to be extended with a probability value because the rules may not always be valid. Our approach is based on architectural abstraction specifications that enable creating architectural models on different levels of abstraction, starting from the system implementation.

3 Empirical study description

For the planning of our study, data collection, and analysis and interpretation of the results, we have followed the experimental process guidelines proposed by Kitchenham et al. (2002). In particular, for the planning phase, the next guidelines are followed: experimental context setting guidelines (examining the related work, defining hypotheses, and considering the circumstances in which an empirical study takes place) and study design guidelines (defining the population of the study, administering the treatments, considering the methods for reducing bias). For data collection, and the analysis and interpretation of the results, the next guidelines are followed: data collection guidelines (defining measures used in the study, ensuring their accurate calculation, considering which data should be excluded), analysis guidelines (choosing the appropriate statistical techniques, performing the data sensitivity analysis), interpretation guidelines (defining the population and the circumstances for which the results apply, specifying study limitations and threats to validity).

3.1 Goals

As mentioned above, this article aims at further elaborating on the concepts and metrics related to the empirical evaluations of the understandability of components that we studied in our previous work. Namely, we examine the usefulness of the hierarchical understandability metrics proposed in the work by Hwa et al. (2009) as well as the participants' experience and try to improve the prediction efficiency of our previous prediction models.

In the following couple of paragraphs we provide the notation and the definitions of the metrics we used in our previous work as well as the metrics from the discussed hierarchical model.

The metrics that we studied in our previous studies include: metrics adapted from the corresponding package level metrics defined by Martin (2003) (studied in Stevanetic and Zdun 2014b) and metrics on graphs that have been previously defined by Allen (2002) and Allen et al. (2007) (studied in Stevanetic and Zdun 2014a).

The metrics adapted from the package-level metrics defined by Martin are shown in Table 1. The first three metrics are adapted from the corresponding package level metrics (number of classes for a package, package afferent coupling and package efferent coupling) defined by Martin (2003). We consider the dependencies between the components in terms of the dependencies between the classes while in the work by Martin the dependencies

Table 1 Metrics adapted from the package level metrics defined by Martin (2003)

Metric's name	Number of classes (NC)	Number of incoming dependencies (NID)	Number of outgoing dependencies (NOD)	Number of internal dependencies (NIntD)
Metric's definition	Total number of classes inside a component.	Total number of classes inside a component.	Total number of dependencies between the classes outside of a component and the classes inside a component that are used by those outside classes.	Total number of dependencies between the classes inside a component and the classes outside of a component that are used by those inside classes.
Measured Property	Design Size/Complexity	Coupling	Coupling	Complexity

Table 2 Graph based metrics definitions (adapted from Allen 2002 and Allen et al. 2007) (please note that $Size(S_i)$ is in principle calculated in the same way as $Size(m_k|S)$, just a different graph (in this case the one defined as S_i) is observed)

Metrics definitions	
Information theory based metrics	Counting based metrics
$\text{Size}(m_k\|S) = \sum_{i \in m_k} (-\log p_{L(i)})$	$\text{CSize}\,(m_k\|\text{S}) = n_k$
$\text{Length}(m_k\|\text{S}) = \max_{i,j \in m_k} \left(\min_r (Size(P_r(i,j))\|S)\right)$	$\text{CLength}\,(m_k\|\text{S}) = \max_{i,j \in m_k} \left(\min_r (CSize\,(P_r(i,j))\,\|S)\right)$
$\text{Complexity}(m_k\|\text{S}) = \sum_{i \in m_k} Size\,(S_i) - \text{Size}\left(m_k\|S^{\#}\right)$	$\text{CComplexity}\,(m_k\|\text{S}) = n_{e_k}$
$\text{Coupling}\,(m_k\|\text{MS}) = \text{Complexity}\left(m_k\|MS^*\right)$	$\text{CCoupling}\,(m_k\|\text{MS}) = \text{CComplexity}\left(m_k\|MS^*\right)$
$\text{Cohesion}\,(m_k\|\text{MS}) = \frac{\text{Complexity}(m_k\|MS^o)}{\text{Complexity}\left(m_k{}^{(n_k)}\|MS^o\right)}$	$\text{CCohesion}\,(m_k\|\text{MS}) = \frac{\text{CComplexity}(m_k\|MS^o)}{\text{CComplexity}\left(m_k{}^{(n_k)}\|MS^o\right)}$

between packages are considered through the number of packages that are related to the given package[1]. The first three metrics characterize the coupling and the size of a component and the fourth metric is introduced to model the internal complexity of the component in terms of the number of dependencies between classes within a component.

Regarding the metrics defined by Allen (2002) and Allen et al. (2007), a graph composed of nodes and edges is considered as an abstraction of a software system and a sub-graph represents a software module. With respect to our case, nodes correspond to the source code classes while edges correspond to the relationships between those classes. Components (that group source code classes) in our case correspond to the modules in the work by Allen (2002).

In this paragraph we provide the metrics' definitions together with some explanations. The definitions of the graph based metrics are shown in Table 2. The notation used for the metrics definitions is the following (adapted from the work by Allen 2002): S – the whole system graph (all nodes and edges), $S^{\#}$ – edges–only graph (edges in S and end points), S_i – node sub–graph (nodes in $S^{\#}$ and edges incident to node i ($i = 0$ for the environment node, $i = 1, ..., n$ for system nodes)), MS – S partitioned into modules, m_k – module k (nodes in a module and their incident edges), MS^* – nodes in MS and intermodule edges, MS^0 – nodes in MS and intramodule edges, $P_r(i, j)$ – path between nodes i and j (nodes and edges on the path between the nodes i and j , including i and j), $p_{L(i)}$ – the proportion of the i-th row pattern in the nodes $\times$ edges table, n_k – the number of nodes in a module, n_{e_k} – the number of edges incident to nodes in a module, and $m_k^{(n_k)}$ – module as a complete graph consisting of nodes in a module and all possible edges between those nodes. The definitions of the length metrics are based on the notion of size, applied to paths (each path is considered to be a module in that case) (Allen 2002). The definitions of the coupling and the cohesion metrics are based on the definition of complexity whereby different graph abstractions are considered. Namely for the complexity metrics a whole system graph is considered while for the coupling and cohesion metrics an intermodule edges graph and an intramodule edges graph are considered, respectively. For instance, the counting coupling metric for a module is equal to the number of edges incident to nodes in a module but only intermodule edges are taken into account unlike the counting complexity metric where edges in a whole system graph are taken into account.

[1]Please note that the relationships between the classes consider dependencies between the classes affected by method calls, data reference or inheritance relationships. The same dependencies are considered for all sets of metrics.

The metrics from the hierarchical understandability assessment model consider six design properties which affect understandability of the modular design of a system. Hwa et al. (2009) systematically examined which properties can affect the understandability and the six of them they found are: design size, complexity, encapsulation (i.e. information-hiding), coupling, cohesion and modular abstraction. Complexity, encapsulation, coupling and cohesion come from general properties which should be managed for software quality (Ghezzi et al. 2002; Booch 1994; Bansiya and Davis 2002) and modular abstraction is a new design concept introduced by the module/package hierarchy (Lungu et al. 2006). Table 3 represents the metrics definitions together with the corresponding notation. Please note that modules in the work by Hwa et al. correspond to components in our case. Please also note that the DMH metric (Depth in Module Hierarchy) might not be always directly applicable for components since e.g. one (big) component might contain classes located in several modules/packages with similar functionalities. In that case, similarly to Hwa et al., we can find an average depth in a hierarchy for all classes in a component with respect to the location of the class in a module/package hierarchy.

3.2 Variables

The variables used in our study can be divided into two sets. The first set is related to the variables that are collected from the participants and the second set is related to the variables that are collected from the studied system. All the variables can also be divided into dependent and independent variables. The first set of variables includes 7 variables, from which 5 are independent variables related to the participants' demographic information: programming experience, Java programming experience, commercial programming experience, experience in programming computer games, and Android programming experience, and the remaining two are the time required to study a component and the percentage of the correct answers on the given questions. The time variable is used to measure the effort required to understand a component and it represents a dependent variable. The percentage of the correct answers variable is introduced to help in estimating the time variable, in the case that the participants do not spend enough time to fully examine the given components in order to achieve a high percentage of correctness (see below for more explanations).

The second set of variables are related to the metrics that we aim to explore (see Tables 1, 2 and 3) and they are calculated from the studied system. All the metrics are treated as independent variables.

The dependent variables and their scale types, units, and ranges are shown in Table 4 while the independent variables together with their scale types, units, and ranges are shown in Table 5.

3.3 Hypotheses

We expect that the given hierarchical understandability metrics can be used as good predictors of the understandability. In addition, we expect that the participants' experience is also significant in predicting the understandability effort. In other words, we expect that the prediction models that use the participants' experience can provide better prediction than using the median as an estimate. In case of the experience variables, we do not expect that they can capture the variability of the measured understandability as good as the metrics related to the system itself. For example, if we have two components to be studied, one with 3 and the other one with 15 classes, it is hard to believe that participants with the same experience would need the same effort to understand them. A bigger component would require

Table 3 Notation for the hierarchical understandability metrics and their definitions (adapted from Hwa et al. 2009)

Notation	Description
MD	The set of modules in the system
C	The set of classes in the system
$C(md)$	The set of classes in a module md ($C = \bigcup_{md \in MD} C(md)$)
$rel_c(c_1 c_2)$	true – if a class $c_1$depends on another class c_2 by method calls, data reference or inheritance relationship false – otherwise
$rel_{md}(md_1 md_2)$	true – if $\exists c_1, c_2 [c_1 \in C(md_1) \cap c_2 \in C(md_2) \cap rel_c(c_1 c_2)]$ false – otherwise
$MD_a(md)$	The set of all ancestor modules of a module md in the module hierarchy

Metric's name	Metric's definitions	Measured Property
Module Size in Classes (MSC)	$\mathrm{MSC}(md) = \lvert C(md) \rvert$	Design Size/Complexity
Number of API Classes (NAC)	$NAC(md) = \lvert \{c_1 \in C(md) \mid \exists c_2 \in C(md2) [rel_c(c_2, c_1) \wedge md_2 \in MD \wedge md_2 \neq md]\} \rvert$	Encapsulation
Direct Module Coupling (DMC)	$DMC(md) = \lvert \{md_2 \in MD \mid rel_c(md, md_2) \vee rel_c(md_2, md), md \neq md_2\} \rvert$	Coupling
Number of Disjoint Clusters (NDC)	$NDC(md) = \left\lvert \left\{ \begin{array}{l} cl \subseteq C(md) \\ \forall c_i \in cl \\ \left[\begin{array}{l} (\lvert cl \rvert = 1 \vee \exists c_j \in cl \left(rel_c(c_i, c_j) \vee rel_c(c_j, c_i)\right)) \wedge \\ \nexists c_k \in C [(c_k \notin cl \wedge (rel_c(c_i, c_k) \vee rel_c(c_k, c_i)))] \end{array} \right] \end{array} \right\} \right\rvert$	Cohesion
Cohesion by Rest of World (CRW)	$CRW(md) = \sum \frac{\lvert SRC(c) \rvert}{\left\lvert \bigcup_{c \in C(md)} SRC(c) \right\rvert}, SRC(c) = \{c_2 \in C \mid (c_2 \in (C - C(md))) \wedge (rel_c(c, c_2) \vee rel_c(c_2, c))\}$	Cohesion
Depth in Module Hierarchy (DMH)	$DMH(md) = \lvert MD_a(md) \rvert$	Abstraction

Table 4 Dependent variables and their scale types, units and ranges (reused from (Stevanetic and Zdun 2014a)

Description	Scale type	Unit	Range
Time	Ratio	Minutes	Positive natural numbers including 0

much more effort than a smaller one that is caused by the variation in their sizes. Therefore we do not expect that the corresponding prediction models for the experience variables are highly accurate. At the end, by combining the system related metrics (the graph-based, package-level, and hierarchical understandability metrics) with the participants' experience, we expect that more efficient prediction models can be obtained compared to those that consider separately the graph based metrics, the package-level metrics, the hierarchical understandability metrics and the participants' experience.

Based on previous considerations we formulate the following set of hypotheses:

Hypothesis (H_1): The hierarchical quality model metrics can be successfully utilized to predict the effort required to understand a component with high practical significance.

Hypothesis (H_2): Prediction models created using just the participants' experiences as predictors have at least one predictor with a non-zero coefficient, i.e. they can predict the understandability effort significantly well.

Table 5 Independent variables and their scale types, units and ranges

Description	Scale type	Unit	Range
Programming exp.	Ratio	Years	Positive rational numbers incl. 0
Java programming exp.	Ratio	Years	Positive rational numbers incl. 0
Commercial programming exp.	Ratio	Years	Positive rational numbers incl. 0
Computer games programming exp.	Ratio	Years	Positive rational numbers incl. 0
Android programming exp.	Ratio	Years	Positive rational numbers incl. 0
NC (Number of Classes)	Ratio	Class	Positive natural numbers incl. 0
NID (Number of Incoming Dependencies)	Ratio	Dependency	Positive natural numbers incl. 0
NOD (Number of Outgoing Dependencies)	Ratio	Dependency	Positive natural numbers incl. 0
NIntD (Number of Internal Dependencies)	Ratio	Dependency	Positive natural numbers incl. 0
Size (inform. and count.)	Ratio	bit/node	Positive real/integer numbers incl. 0
Complexity (inform. and count.)	Ratio	bit/edge	Positive real/integer numbers incl. 0
Coupling (inform. and count.)	Ratio	bit/edge	Positive real/integer numbers incl. 0
Length (inform. and count.)	Ratio	bit/node	Positive real/integer numbers incl. 0
Cohesion (inform. and count.)	Ratio	–	Positive real/rational numbers incl. 0
Percentage of the correct answers	Ratio	–	[0,100]%
MSC (Module Size in Classes)	Ratio	class	Positive integer numbers incl. 0
NAC (Number of API Classes)	Ratio	class	Positive integer numbers incl. 0
DMC (Direct Module Coupling)	Ratio	module	Positive integer numbers incl. 0
NDC (Number of Disjoint Clusters)	Ratio	–	Positive integer numbers incl. 0
CRW (Cohesion by Rest of World)	Ratio	class	Positive rational numbers incl. 0
DMH (Depth in Module Hierarchy)	Ratio	–	Positive integer numbers incl. 0

Hypothesis (H_3): Combining both the system related metrics and the participants' experiences leads to a significantly increased efficiency of the obtained prediction models compared to the prediction models that use just the graph based metrics.

Hypothesis (H_4): Combining both the system related metrics and the participants' experiences leads to a significantly increased efficiency of the obtained prediction models compared to the prediction models that use just the package-level metrics.

Hypothesis (H_5): Combining both the system related metrics and the participants' experiences leads to a significantly increased efficiency of the obtained prediction models compared to the prediction models that use just the participants' experiences.

Hypothesis (H_6): Combining both the system related metrics and the participants' experiences leads to a significantly increased efficiency of the obtained prediction models compared to the prediction models that use just the hierarchical understandability metrics.

3.4 Study design

3.4.1 Subjects

The participants of the study are 49 master students. The study took place within the Advanced Software Engineering (ASE) lecture at the University of Vienna in the Winter Semester 2013.

3.4.2 Objects

The object of our study was the Soomla Android store [2] system, version 2.0. It is an open source cross platform framework that supports virtual economy in mobile games, and encourages better game design and faster development. We choose the given system because of the following factors:

- The system is open source which enables us to carry out the study and communicate its results.
- The system is written in Java which the participants are familiar enough with.
- The application domain of the system is probably known to the participants from similar game applications.
- The system has industrial relevance since it is used in many real-world games.
- The source code of the system contains of 54 classes within 8 packages. The system has in total 3623 LOC (excluding blank and commented lines) and therefore it is probably understandable within a study session, but also not too simple.

3.4.3 Instrumentation

Architectural documentation about the Soomla Android store system A UML component diagram representing the architecture of the system, its conceptual description and the traceability links that relate the architecture to the system implementation (class design) are handed in to the participants.

The architecture of the system is shown in Fig. 2. There are in total seven architectural components: *Security (C1)*, *CryptDecrypt (C2)*, *PriceModel (C3)*, *GooglePlayBilling*

[2] all versions: https://github.com/soomla/android-store, studied version: https://swa.univie.ac.at/soomla/

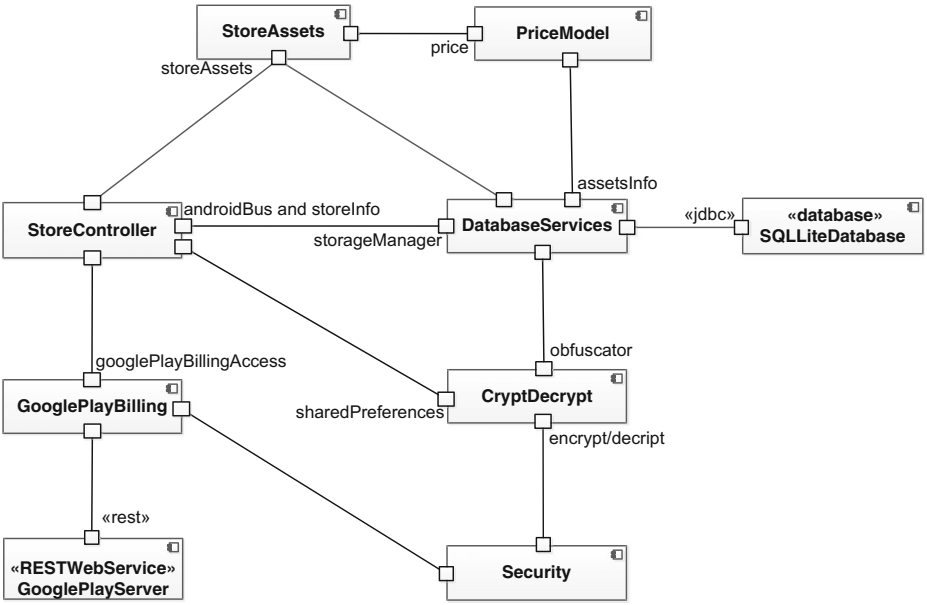


Fig. 2 Architectural description of the Soomla Android store system in the form of a UML component diagram (reused from Stevanetic and Zdun 2014b)

(C4), *StoreController (C5)*, *DatabaseServices (C6)*, and *StoreAssets (C7)*. In addition there exist two more external components: *GooglePlayServer*, the REST Web Services running at Google, and *SQLLiteDatabase*, the database accessed using JDBC. The architectural representation of the system is constructed by two experienced software architects. They fully studied the given system and its documentation and extracted its architecture together with the traceability links to the system implementation. Table 6 shows a short description of the roles that the components play in the system.

Table 6 Soomla Android store architectural components and their roles in the system (reused from Stevanetic and Zdun 2014b)

Component	Component's role
Security (C1)	Verifies the information during the purchasing process
CryptDecrypt (C2)	Provides encrypt/decrypt services to obfuscate the billing information and to encrypt/decrypt the data stored to or retrieved from the database
PriceModel (C3)	Describes the model that explains how the prices of virtual items are formed
GooglePlayBilling (C4)	Simplifies in-app billing API which is a Google play service that lets you sell virtual goods from inside your applications
StoreController (C5)	Provides the runtime functionality of the Android store and contains up-to-date store information
DatabaseServices (C6)	Performs the initialization of the database and implement retrieve, add, and remove operations for store assets in the database
StoreAssets (C7)	Describes the virtual items used in the application (virtual currency, virtual goods, and their classification)

Source code access The access to the source code of the system was browser-based, on prepared computers. Namely we enabled the participants to easily navigate through the components and open the source code of their realized classes by grouping the classes into the corresponding components.

A questionnaire to be filled-in by the participants The first part of the questionnaire is related to the rated participants' experiences including. The second part contains the understandability questions related to the 7 architectural components. Four true/false questions were provided to be studied for each component, and the participants had to check the right answers among them. In order to correctly answer the questions, the participants had to fully understand the functionalities of each component by examining the relationships (as well as the roles of those relationships) among the classes inside a component and the relationships among the classes inside a component and the classes outside of that component. In the case of bigger components, answering the questions requires to analyse more classes and their relationships than in the case of smaller components. Table 7 shows an example of two questions, one for Component GooglePlayBilling (Q1) and the other one for Component Security (Q2). Component GooglePlayBilling (has 11 classes) is bigger than Component Security (has 2 classes) and therefore the corresponding question(s) require to examine more classes and their relationships than the question(s) for Component Security. The order in which the seven components are studied is changed for different participants so that 7 random combinations of components are generated and assigned to the participants (the order of questions within the components remained the same). For example, one participant studied the components in one order, e.g.: C2, C6, C1, C3, C5, C7, and C4 while another one studied them in some other randomly generated order, e.g.: C1, C5, C7, C3, C4, C6, and C2. The randomization enables us to get more/less balanced data for all the components in terms of equalizing the fatigue effects or the lack of time needed to complete all required tasks.

In order to measure the time that the participants spent on analysing each of the components, we provided a table with the time slots. Each slot contains a start and a stop time. The start time indicates the time when the participants started analysing a component while the stop time indicates the time when they finish it. Several slots were provided for each

Table 7 An example of two questions (one for Component GooglePlayBilling and one for Component Security)

Q1 (*GooglePlayBilling*).
The usual data flow in the component *GooglePlayBilling* can be represented using the next sequence of relationships: class *BillingRequest* – Android Market – class *BillingReceiver* – class *BillingService* – class *ResponseHandler* – class *PurchaseObserver*. The sequence can be explained as follows: The class *BillingRequest* sends messages to Android Market using *MarketBillingService*, then the class *BillingReceiver* receives and forwards all received messages for handling the further communication with Android Market to the *BillingService* class, then *BillingService* notifies the application about purchase state changes using the *ResponseHandler* class which at the end updates the UI using the received information from the Android Market (posting appropriate events, updating currency balances, items, etc.).
a) True □ b) False □
Q2 (*Security*).
The class *Security* uses the class *AESObfuscator* in order to obfuscate (make unclear) of values before saving to database (DB) and when retrieving from DB.
a) True □ b) False □

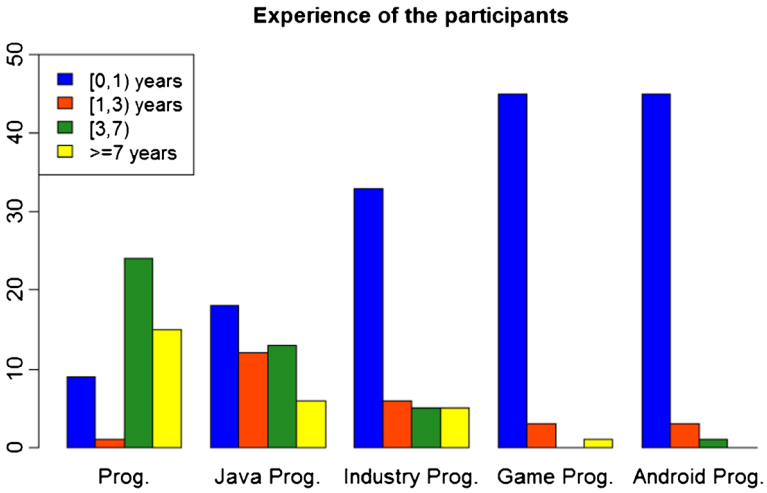


Fig. 3 Participants' demographic information

component in case that the participants want to analyse a component several times. The format used for writing the time is *hour : minute*. The time limit for the whole study was 90 minutes. None of the participants has been studied the system before so that a potential bias that some participants spent additional time (beside the time written in the time slots) on examining the system is negligible. To ensure that there will be enough time to analyse all the components within the study session of 1.5 hours, we tried the same study with several our colleagues before we tried it within the course. All of them agreed that the given tasks are appropriate for the given time limit. All the above explained instruments are available on the following Web address[3]. The file containing our results to be assessed by others is available on the same page.

3.5 Execution

3.5.1 Data collection

Figure 3 shows the data related to the participants' demographic information.

Based on the information from the figure we can say that the programming experience of the participants is medium to high. Most of them have more than 3 years of programming experience. Many of the participants also have industrial programming experience but only a few of them have game programming experience and experience with Android.

The descriptive statistics (mean, median, and standard deviation) related to the time and the percentage of the correct answers variables are shown in Fig. 4. From our results we excluded the participants who have less than one year of programming experience (9 of them). Some of the participants did not specify both start and stop time for all studied components, so that we also excluded those results from the analysis (just for the components where the start and stop time were not specified). The total number of collected data samples for all components is $7(components) \times 49(students) = 343$ and the number of excluded data samples is 103.

[3]https://swa.univie.ac.at/soomla-architectural-components/

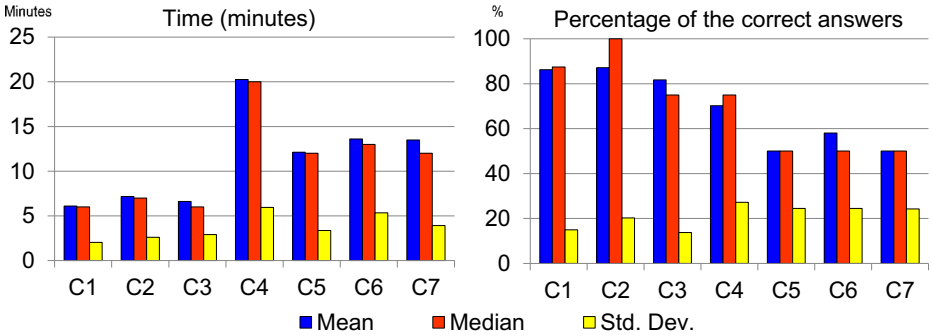


Fig. 4 Descriptive statistics for the time and the percentage of the correct answers variables (reused from Stevanetic and Zdun 2014a)

The data related to the metrics we aim to explore are shown in Tables 8, 9 and 10. The graph based metrics are automatically calculated from the corresponding graph abstractions of the system. The graph abstraction of the whole system is also utilized for the calculation of the package based and hierarchical understandability metrics. The metrics are independently calculated by two architects who studied the system in order to avoid misinterpretation of their calculations. The accuracy of the graph based metrics calculations is additionally tested on the examples provided by Allen (2002).

Looking at Fig. 4 we can say that the obtained time for the first three components (*C1*, *C2* and *C3*) is significantly lower than the time for the remaining four components. This observation is expected since the first three components contain smaller number of classes in comparison to the other four. Another observation is related to the component *C4*. The average time needed to analyse this component is significantly higher than the time needed to analyse the components *C5*, *C6* and *C7*. Consequentially the percentage of the correct answers for the components *C5*, *C6* and *C7* is decreased with respect to the component *C4* which has more/less similar values to the smaller components (*C1*, *C2* and *C3*). Even though it seems expected that the percentage of the correct answers decreases for the components that have many classes simply because of the higher amount of information that need to be handled which increases the probability of missing some relevant information parts, it seems also that the participants spent a bit less time for analysing the components *C5*, *C6* and *C7* than it is necessary (or at least for the component *C7* which has the same number of classes as the component *C4*) in order to score better and achieve the higher percentage of

Table 8 Package based component level metrics (reused from Stevanetic and Zdun 2014b)

Component level metrics	Number of classes	Number of incoming dependencies	Number of outgoing dependencies	Number of internal dependencies
Security (C1)	2	3	4	1
CryptDecrypt (C2)	5	9	0	5
PriceModel (C3)	3	1	4	2
GooglePlayBilling (C4)	11	4	3	12
StoreController (C5)	8	5	15	5
DatabaseServices (C6)	8	8	8	13
StoreAssets (C7)	13	9	3	14

Table 9 Graph based component level metrics (reused from Stevanetic and Zdun 2014a)

Component level metrics	Size		Complexity		Coupling		Cohesion		Length	
	Info	Count	Info	Count	Info	Count	Info	Count	Info	Count
Security (C1)	11.23	2	62.72	8	44.15	7	1.00	1.00	11.23	2
CryptDecrypt (C2)	28.07	5	138.2	14	54.82	9	0.49	0.50	16.84	3
PriceModel (C3)	16.84	3	66.72	7	31.34	5	0.61	0.67	16.84	3
GooglePlayBilling (C4)	61.76	11	222.3	19	42.11	7	0.27	0.27	28.07	5
StoreController (C5)	44.92	8	205.7	25	125.2	20	0.31	0.33	16.84	3
DatabaseServices (C6)	44.92	8	298.2	29	101.5	16	0.47	0.46	16.84	3
StoreAssets (C7)	61.76	11	274.3	24	64.73	10	0.40	0.39	22.46	4

the correct answers. With respect to this and the discussion in Section 3.2 the percentage of the correct answers variable is used to help in estimating the time required to fully analyse a component and achieve maximal correctness of 100%.

3.5.2 Validation

To prevent the participants from using forbidden materials and talking to each other at least one observer was present in the lab during the study execution. It also enabled the participants to pose clarification questions. The materials given to the participants are collected before any of them left the lab. There were no cases where the participants behaved unexpectedly.

4 Analysis

The following statistical tests are used for analysing the data.

- Variance Inflation Factor (VIF) (O'brien 2007) and Condition Number (CN) (Belsley 1991) - Collinearity Analysis
- Multiple Regression Analysis (MRA) (Rubinfeld 2000)

VIF and CN are commonly used to detect the multicollinearity problems (see below). MRA is commonly used to examine the relationship between one dependent variable and more than one independent variables or predictors. The relationship is assumed to be linear, which makes a model easy to interpret. Furthermore, the "true" relationship is often at least

Table 10 Hierarchical understandability component level metrics (reused from (Stevanetic and Zdun 2016))

Component level metrics	MSC	NAC	DMC	NDC	CRW	DMH
Security (C1)	2	1	2	0	1	1
CryptDecrypt (C2)	5	5	3	0	1.8	2
PriceModel (C3)	3	1	2	0	1.25	3
GooglePlayBilling (C4)	11	4	2	0	3.5	1
StoreController (C5)	8	2	4	0	1.33	1.87
DatabaseServices (C6)	8	5	4	0	2.29	1.87
StoreAssets (C7)	11	6	3	0	1.43	2.64

Table 11 Condition number and variance inflation factor – information theory graph based metrics

Variable	VIF	VIF (w/o Length)	VIF (w/o Size)	VIF (w/o Size, Length)	VIF (w/o Complexity, Length)
Size	52.61	12.95	N/A	N/A	3.69
Complexity	12.62	10.44	5.66	2.97	N/A
Coupling	12.90	2.59	7.60	1.64	1.29
Cohesion	12.54	3.92	9.01	2.09	3.49
Length	54.54	N/A	13.43	N/A	N/A
Percentage of the correct answers	1.39	1.38	1.36	1.34	1.38
Condition number (CN)	33.78	18.10	13.68	7.73	7.27

approximately linear over the range of values that are of interest to us. Even if it is not, the variables can be transformed in such a way as to linearise the relationship. The analyses are performed using the programming language R (R Development Core Team 2008).

4.1 Collinearity analysis

Collinearity analysis aims at indicating the variables that are highly correlated with some other variables. Those variables should be excluded from the set of all possible predictors potentially considered for the prediction. To test for possible correlations within the studied metrics sets, we calculate the Condition Number (CN) and the Variance Inflation Factor (VIF). The VIF values greater than 10 suggest high correlation, i.e. multicollinearity problems among the tested variables. The CN values greater than 30 suggest the same (Belsley et al. 1980).

Regarding the information theory and counting graph based metrics we consider them as two separate sets of predictors because we already saw that they are highly correlated in our case. Therefore, all potential predictors considered for the prediction models generation include either the information theory based metrics or the counting based metrics and the percentage of the correct answers (see discussion in Section 3.5.1). The VIF and the CN values for the information theory graph based metrics and the package based metrics are shown in Tables 11 and 12 respectively.

Regarding the information theory graph based metrics, as we can see from Table 11, the greatest VIF value when all metrics (predictors) are included (column "VIF") is the value for the Length metric ($54.54 > 10$). The VIF value for the Size metric is very close to it (52.61). Therefore, in the first step we can exclude either the Length or the Size metric from the set of predictors. The results for the VIF values and the CN value after excluding these metrics are shown in the third and the fourth column of the figure. After excluding the Length metric there are two predictors that can be further excluded, the Size or the Complexity metric (they are both greater than 10 and have similar VIF values[4]). After excluding the Size metric,

[4]In principle the predictors with the highest VIF values are step-by-step excluded from the set until the highest VIF value becomes less than 10. In our case we have two predictors that have high VIF values that are close to each other (both in the first and in the second step of the analysis) and therefore we can exclude either one or another predictor. The performances of the obtained linear regression models in all the cases show very tiny differences between each other (see Section 4.2).

Table 12 Condition Number and Variance Inflation Factor – package based metrics (reused from Stevanetic and Zdun 2014b)

Variable	VIF	VIF (without NIntD)
Percentage of the correct answers	1.4405	1.4391
Number of Classes (NC)	7.3977	1.6092
Number of Incoming Dependencies (NID)	1.5776	1.4213
Number of Outgoing Dependencies (NOD)	1.2432	1.2135
Number of Internal Dependencies (NintD)	7.9627	N/A
Condition number (CN)	5.72	4.94

only the Length metric has the VIF value greater than 10. Therefore, we obtained two final sets of possible predictors that are used for creating the prediction models for information theory based metrics. Excluded predictors are either the Size and the Length metrics or the Complexity and the Length metrics. The final sets of predictors have acceptable VIF and CN values (see for example those in Table 11). Using the same procedure for the counting based metrics we obtain three final sets of possible predictors, i.e. the sets exclude either the Size and the Length metrics, the Complexity and the Length metrics, or the Size and the Cohesion metrics.

For the package based metrics, as we can see from Table 12, the VIF coefficients in the case when all predictors are included are less than 10 where the greatest VIF value is 7.96 (for NIntD). Therefore, we can say that there is a slight tendency of multicollinearity between the variables. Hence, we decided to exclude the NIntD from the set of all predictors after which we get acceptable results for both VIF and CN values (see Fig. 12).

Regarding the hierarchical understandability metrics they are no multicollinearity problems in that set of metrics. The highest VIF value has the MSC metric (3.87) and the CN value for this set of predictors is 13.11. The participants' experiences also do not express multicollinearity problems. The highest VIF value has the programming experience variable (1.62) and the CN for the whole set of variables is 8.29. As mentioned above, we would like to examine the model where all the studied variables are taken into account, i.e. the hierarchical understandability metrics, the participants' experiences, the package based metrics and the counting or information theory graph based metrics. Combining all those 4 sets together introduces multicollinearity problems since there are metrics in multiple sets that measure the same concepts (size, coupling, and cohesion) even if different metrics for those concepts are used. After examining the VIF and CN values all graph based and package based metrics can be excluded from the set. After excluding these metrics, the highest VIF value has the MSC metric (3.98) and the CN value for the remaining set of variables is 16.76.

4.2 Multiple regression analysis

In this part of the analysis we create multiple regression models that can be used for predicting the time variable. They are also used to test our hypotheses described in Section 3.3. To prevent the over-fitting of the data, i.e. to enable more efficient generalization of the results we perform the Mallows' Cp calculation for creating the prediction models (Kobayashi and Sakata 1990). If p is the number of predictors including the constant predictor, if it exists, all the models that satisfy the equation $Cp \leq p$ must be considered as reasonable good fits with respect to preventing data over-fitting.

Before we move to the regression analysis, we shortly explain the role of the percentage of the correct answers variable. In Section 3.2, we mentioned that this variable is used as an independent variable to help estimating the time as a dependent variable. Namely, there might exist a dependency between the time and the percentage of the correct answers because if the participants spend less than some minimum time required to analyse a component, the percentage of the correct answers will probably decrease because of an incomplete insight into all relevant component parts. Therefore with the help of the percentage of the correct answers variable we can estimate the time required to fully understand a given component, i.e., to achieve 100% of the correct answers.[5] If we replace the value for the percentage of the correct answers in the obtained prediction models (see below) with the constant value of 100%, the effort required to fully understand a component is obtained, that further depends only on other factors included in the model. Please note also that predicting the time for 100% of the correctness is not the most realistic requirement because of the lack of the data that are available for that. For example we can also estimate 75% of the correctness which would be more accurate because there exist more data for that. However in our case we use 100% because of the negligible difference in the prediction.

To check the accuracy of the obtained prediction models we calculated a goodness of fit measure using the following equation based on the absolute deviation of the median (Kampenes et al. 2007) (assuming X_i is the prediction and Y_i is the actual value):

$$A(accuracy) = \frac{\sum_i |Y_i - X_i|}{\sum_i |Y_i - median(Y_i)|}$$

The smaller the value of A the better prediction. If the value is greater than 1, the estimation is not working, i.e. there is no evidence that the prediction is better than using the median as an estimate. The value (1-A) represents the proportion of the variation in the Y variable explained by the predictions. (1-A) is a robust analogue of R^2, so the following guidelines based on those proposed by Kampenes et al. (2007) can be used for the effect size calculation: the (1-A) values in the range of 0 to 0.0372 represent a small effect size, the values in the range of 0.0372 to 0.208 represent a medium effect size while the values in the range of 0.208 to 0.753 represent a large effect size. Furthermore, for good prediction models the residuals have to be normally distributed which is the case with our data. The influential points are the points whose removal will cause a large change in the fit, and they can be detected using Cook's distance contour lines (Cook 1977). When some points have a distance that is larger than 1, it suggests that the model might be poor or might have outliers. Our models do not have influential points. We further provide the significance of the coefficient of determination (R^2) for the obtained models that is measured by the F-statistic (Dalgaard 2004).

In order to test our hypotheses described in Section 3.3, we first generate the prediction models that consider: 1) the package based understandability metrics, 2) the graph based understandability metrics, 3) the hierarchical understandability metrics, 4) the participants experiences, and 5) both system related metrics (the package based, graph based and hierarchical understandability metrics) and the participants experiences, using the above explained analysis. The obtained models are then compared if there is a significant

[5]Please note that predicting the percentage of the correct answers variable is also possible but since we focus on estimating the time as a measure for the understandability effort we consider the percentage of the correct answers as an auxiliary variable that helps in predicting the time variable.

Table 13 Models' parameters – package based metrics

Coefficients	Intercept	Percentage of the correct answers	Number of Classes	Number of Incoming Dependencies	Number of Outgoing Dependencies
Model 1	0	4.8597	1.5162	−0.5349	0
Model 2	0	4.5754	1.4628	−0.5175	0.1150
Model 3	2.4250	2.8902	1.4200	−0.5795	0
Models' characteristics	F-statistic: p-value	Accuracy	Effect size (1-Accuracy)		
Model 1	< 2.2e−16	0.6436	0.3563		
Model 2	< 2.2e−16	0.6447	0.3552		
Model 3	< 2.2e−16	0.6387	0.3612		

difference in their prediction capabilities. The best 3 models in terms of the given accuracy measure (A) for the above given cases that fit the explained criteria ($Cp \leq p$) are shown in Tables 13 (the package based metrics), 14 (the counting graph based metrics), 15 (the hierarchical metrics), 16 (the participants' experiences), 17 (the participants' experiences together with the system related metrics). For all shown models except the ones that consider the participants' experiences as predictors, the effect size is in the range of 32% to 40% which represents a large effect size. Those results suggest that the obtained prediction models have high practical significance. Please note that the percentage of the correct answers variable is taken into account for the construction of the prediction models as independent variable based on the discussion provided in Section 3.1. With regard to that, we have to check if this variable alone captures the most of the variance in the measured understandability effort in which case the studied metrics and participants' experiences variables do not play an important role. It is not the case, since the prediction model that considers only the percentage of the correct answers variable has the accuracy measure (A) greater then 1 and does not provide better prediction then using just the median as an estimate.

Another useful technique for overcoming the over-fitting problem is the cross-validation analysis (Field et al. 2012). Beside the Mallows' Cp analysis, we also applied 10-fold cross-validation technique on our data.[6] The results of the cross-validation analysis corroborate the results of the Mallows' Cp analysis and confirm their validity.

Regarding the hypothesis $\mathbf{H_1}$ that consider the prediction models for the hierarchical understandability metrics, with respect to the analysis undertaken, we can say that the hypothesis $\mathbf{H_1}$ is supported, i.e. the hierarchical quality model metrics can be successfully utilized to predict the effort required to understand a component with high practical significance.

Regarding the hypothesis $\mathbf{H_2}$ that consider the participants' experiences as predictors, we see from Table 16 that the effect size of the obtained models (around 4%) is on the border between small and medium. Compared to the other obtained models that consider the system related metrics, we can say that these models are much less accurate and efficient.

[6]"Cross Validation techniques in R: A brief overview of some methods, packages, and functions for assessing prediction models".

Table 14 Models' parameters – counting graph based metrics

Coefficients	Size	Complexity	Coupling	Length	Cohesion	Percentage of the correct answers
Model 3	1.3890	0	0	0	0	2.7413
Model 4	1.4519	0	−0.0665	0	0	3.0246
Model 5	1.3893	0	0	0	0.0334	2.7157
Models' characteristics	F-statistic: p-value	Accuracy	Effect size (1-Accuracy)			
Model 3	< 2.2e−16	0.6770	0.3230			
Model 4	< 2.2e−16	0.6781	0.3219			
Model 5	< 2.2e−16	0.6768	0.3232			

These results comply with the discussions provided in Section 3.3 that the participants' experiences cannot capture the variability as good as the metrics related to the software model itself. Therefore it has been demonstrated that the hypothesis $\mathbf{H_2}$ is supported, i.e. the prediction models for the effort required to understand a component created using just the participants' experiences as predictors have at least one predictor with a non-zero coefficient, i.e. they can predict the understandability effort significantly well. Please note that this does not mean that the obtained models are well-fitted (accurate), it just means that they can predict the significant amount of variance in the model comparing to the remaining unexplained variance (Field et al. 2012). Based on the obtained result, we can say that the participants' experiences are important and can significantly improve the understandability but they are not able to appropriately capture the variance in the data caused by the variation of system's structural properties (like size, coupling, cohesion, etc.). The result is as mentioned above expected.

Finally, to test the hypotheses $\mathbf{H_3}$, $\mathbf{H_4}$, $\mathbf{H_5}$, and $\mathbf{H_6}$, we compare the efficiency of the obtained prediction models that use both the system related metrics and participants' experiences on one side and the models that use separately the package based, graph based, and hierarchical understandability metrics, as well as the participants' experiences. For that purpose, we calculate two parameters, the difference between the AICc (second-order corrected Akaike Information Criterion) values (ΔAICc) for the models to be compared and the corresponding evidence ratios (w). These parameters are commonly used for model comparisons

Table 15 Models' parameters - hierarchical understandability metrics

Coefficients	MSC	NAC	DMC	CRW	DMH	Percentage of the correct answers
Model 1	1.2979	−0.7210	0	2.0825	−0.1394	1.6652
Model 2	1.1517	−0.7889	0.5555	2.5473	−0.1371	0
Model 3	1.2463	−0.7858	0.4285	2.1331	−0.3535	1.2900
Models' characteristics	F-statistic: p-value	Accuracy	Effect size (1-Accuracy)			
Model 1	< 2.2e−16	0.6053	0.3947			
Model 2	< 2.2e−16	0.6054	0.3946			
Model 3	< 2.2e−16	0.6024	0.3976			

Table 16 Models' parameters - participants' experiences

Coeff.	Prog.	Java Prog.	Comm. Prog.	Game Prog.	Android Prog.	Percentage of the correct answers
Mod 1	0	0	0	0	0.8977	12.7565
Mod 2	0	0	0	0.3978	0.7729	12.6131
Mod 3	0	0	0.2173	0	0.9456	12.0908

Models' characteristics	F-statistic: p-value	Accuracy	Effect size (1-Accuracy)
Mod 1	$< 2.2e-16$	0.9608	0.0392
Mod 2	$< 2.2e-16$	0.9622	0.0378
Mod 3	$< 2.2e-16$	0.9614	0.0384

in case of non-nested models[7] (Burnham and Anderson 2002). If the obtained difference (ΔAICc) is lower than 4 we can say that there is no significant difference in the prediction capabilities (power) of the given two models (Burnham and Anderson 2002). If the difference is in the range [4,7], we can say that there is a significant difference in the prediction capabilities, and, if the difference is greater than 10, a very strong difference exists (Burnham and Anderson 2002). The evidence ratio is a value of one model being more likely than the other model (for example a model with AICc=120 is nearly 150 times more likely than a model with AICc=130). We compare the best models from each group in terms of the AICc measure. The results of the analysis are shown in Table 18.

Column ΔAICc 1 shows the difference between the AICc values of each model and the model that includes both the hierarchical understandability metrics and the participants' experience variables (Model 1). The corresponding evidence ratios are shown in column w1. From the obtained ΔAICc 1 values we see that there is a large significant difference (values greater then 10) in prediction capabilities between Model 1 and the last 3 listed models Models 3, 4, and 5. Regarding the difference in prediction between Model 1 and Model 2 (that just includes the hierarchical understandability metrics) no significant difference exists (ΔAICc 1 $= -1.88$). Based on the obtained results we can say that the Hypotheses $\mathbf{H_3}$, $\mathbf{H_4}$, and $\mathbf{H_5}$ are supported while the Hypothesis $\mathbf{H_6}$ is not supported, i.e. combining both the system related metrics and the participants' experience variables leads to a significantly increased efficiency of the obtained prediction models compared to: 1) the prediction models that use just the graph based metrics, 2) the prediction models that use just the package based metrics, and 2) the prediction models that use just the participants' experiences. The model with both the hierarchical understandability metrics and the participants' experience is not significantly better in prediction compared to the model that includes just the hierarchical understandability metrics. It is even a little bit worse (the AICc value is increased by 1.88)[8]. As a consequence of this last fact we can also conclude that the model that includes just the hierarchical understandability metrics is significantly better than the last 3 listed models, i.e. Models 3 ,4, and 5 (the differences of the AICc values are increased by 1.88 in comparison with the ΔAICc 1). Columns ΔAICc 2 and w2 show the differences of the AICc values and the corresponding evidence ratios between each model and Model 2.

[7]Nested models are those where all predictors from one model are also contained in the other model. Our models use different sets of predictors and therefore they are non-nested.

[8]The reason for that is that Model 2 has a lower number of predictors which is more preferable for the AICc criterion.

Table 17 Models' parameters - participants' experiences together with the system related metrics

Coeff.	Prog.	Java Prog.	Comm Prog.	Game Prog.	MSC	NAC	CRW	DMC	DMH	Percentage of the correct answers
Mod 1	0	0	0	0.229	1.188	−0.70	2.625	0	0.230	0
Mod 2	0	0	−0.017	0.225	1.188	−0.69	2.632	0	0.243	0
Mod 3	−0.019	−0.09	0	0.293	1.305	−0.74	2.119	0	0	1.790
Models' characteristics	F-statistic: p-value	Accuracy	Effect size (1-Accuracy)							
Mod 1	$< 2.2e-16$	0.6092	0.3908							
Mod 2	$< 2.2e-16$	0.6079	0.3921							
Mod 3	$< 2.2e-16$	0.5936	0.4063							

Table 18 Model comparisons

Model	Description	ΔAICc 1	ΔAICc 2	w1	w2
Model 1	The hierarchical understandability metrics plus the participants' experiences	0	1.88	1	2.56
Model 2	The hierarchical understandability metrics	−1.88	0	0.39	1
Model 3	The participants' experiences	> **10.00**	> **10.00**	> **e+05**	> **e+05**
Model 4	The graph-based metrics	> **10.00**	> **10.00**	> **e+05**	> **e+05**
Model 5	The package-level metrics	> **10.00**	> **10.00**	> **e+05**	> **e+05**

ΔAICc 1 = AICc (Model i) - AICc (Model 1); **w1** (evidence ratio) = exp(0.5*ΔAICc 1)

ΔAICc 2 = AICc (Model i) - AICc (Model 2); **w2** (evidence ratio) = exp(0.5*ΔAICc 2)

To summarize the obtained results we can say the following. The introduced hierarchical understandability metrics can be used to predict the understandability effort of a component with high practical significance. On the one hand, those prediction models are significantly better in predicting the understandability effort than the models obtained using the graph based metrics, the package based metrics or the participants' experiences. On the other hand, those models are not significantly different or worse in the prediction from the models that combine both the system related metrics (the graph based, package based and hierarchical understandability metrics) and the participants' experiences. The participants' experience can predict a significant amount of variance in the data but the obtained models are not as accurate as the models that use the metrics related to the system itself (concretely the hierarchical understandability metrics).

With respect to the discussions in Section 3.2 we can now calculate the effort required to fully understand a component by replacing the percentage of the correct answers variable in the obtained prediction models with the constant value of 100%. In Fig. 5 the predicted time variable using the model with the highest effect size value (Model 3 from Table 15) and the time variable obtained from the participants are shown. The predicted time variable significantly differs from the time variable obtained from the participants just for the component *StoreAssets (C7)*. It can be interpreted that the participants needed a bit more time

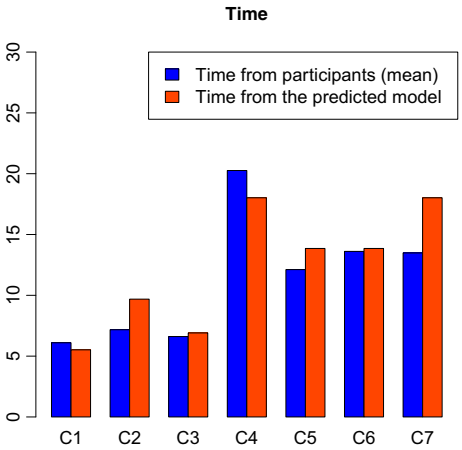


Fig. 5 The time from the participants and the time from the predicted model where the correctness of the answers is set to 100 %

for analysing the component *StoreAssets (C7)* in order to be able to answer all the questions correctly. It really makes sense because the component *StoreAssets (C7)* has 11 classes as there are in the component *GooglePlayBilling (C4)* and therefore we expect that they require similar times in order to be fully studied.

Before we move to the next section let us examine one more interesting aspect. Namely in the context of process model understandability (Canfora et al. 2005; Aguilar et al. 2007) (see Section 2 for more details) different empirical validations showed that size is not enough to fully determine phenomena of understanding: additional metrics like structuredness help to improve the explanatory power significantly (Mendling 2008). We confirm this in the context of architectural components by comparing the prediction power of the model that considers just the size metric (MSC metric) and the best obtained model in terms of the accuracy measure that considers the hierarchical understandability metrics. The obtained ΔAICc value is 59.707 and the corresponding evidence ratio is w=9.2e+12. These results confirm that there is a strong significant difference in the prediction power between the mentioned models.

5 Validity evaluation

In this section we discuss how we tried to minimize the threats to validity. The following threats are taken into account:

Conclusion validity The conclusion validity indicates to which extent the conclusions are statistically valid. The sample size is one of the possible threats for the statistical validity. In our case 49 students answered the questions for the 7 components.

While the number of participants we used is quite fair the dataset consisting of 7 components is limited to the relatively small-size dataset due to the limited time of the study session. However after performing the power analysis in R (Kabacoff 2011) we found that the statistical power obtained for our sample with the medium effect size of 0.15, which corresponds to the expected R^2 around 0.4 (we assumed the effect size suggested by Cohen (1988)) is 0.99. It means that the likelihood of finding a prediction model when there is one with the given effect size is 99 %. Therefore the total sample size is not considered to be a threat for the conclusion validity. Anyway we plan to increase the number of studied components in our future work.

Construct validity The construct validity describes the degree to which the used variables are accurately measured by the appropriated instruments.

A possible threat to the construct validity might be related to the instruments for measuring the time variable. Namely the participants might have forgotten to write the time in the time slots appropriately, i.e. right before they start analysing a given component and right after they finish it. To minimize that threat we put a reminder before the text related to each component to remind the participants to write the time appropriately.

For the future reproduction studies in a browser-based environment it might be useful to set up a script that monitors the website being viewed and automatically collects the information per student. Another option would be to use IDE tracking tools, e.g. an Eclipse plugin.

The true/false questions might seem to be not good choice for measuring the understandability since the participants could get the right answer 50% of the time. However, the maximal likelihood that any number of participants from the given range (1–49) get the correct answers on 2 or more question (2, 3, or 4) is just around 14%. Therefore the

likelihood of obtaining a substantially higher score by guessing alone is very small (Ebel and Frisbie 1991).

The component level metrics are calculated automatically with the help of the tool ObjectAid UML Explorer[9]. The dependencies between the source code classes are visualized in the tool and based on those visualizations the corresponding graph abstractions used for the graph based metrics calculations are manually generated. The hierarchical understandability metrics are directly calculated from the provided visualizations. The accuracy of the graph based metrics calculations is additionally tested on the examples provided in the work by Allen (2002). In any case, all metrics are independently calculated by two architects who also created the architecture of the studied system. Therefore, the threat that the metrics calculations are not valid is highly reduced.

Internal validity The internal validity relates to the degree to which conclusions can be drawn about cause-effect of independent variables on the dependent variables. The following threats are considered:

- **Participants competences and experiences.** The participants' competence might influence the study results. In our case all participants have knowledge about software development and software architecture, as well as of software traceability. Most of them have at least medium experience in programming. Regarding the participants' experiences we considered the experience years (see Table 5). Some other potential variables related to the participants' demographic information may affect the obtained results to a certain extent. For example, in addition to the considered variables we examined possible differences in our results in case we add the final participants' grades in the course and if the participants successfully passed two other courses that might be relevant for the studied problem Software Engineering, and Information Systems and Technologies. After considering these variables the accuracy measure for the best prediction model that considers the experience variables only slightly changed. This result does not affect any of our hypotheses and considerations and therefore we decided not to report about it in detail. In our future work we plan to include experts who have many years of professional experience and to test whether some different prediction models can be obtained.
- **Fatigue effects.** Total time limit for the whole study was 1.5 hours so fatigue was not very relevant. Also, the randomization of the tasks helped to cancel out these effects.
- **Question design.** The fact that we used more complex questions in case of larger components might cause additional difficulties to answer them. It is because in practice, people have a limit to the number of things they can keep in mind at a time. However, please note that each smaller question within the bigger one can be separately studied.

External validity The external validity is related to the degree to which the results of the study can be generalized to the broader population. The greater the external validity, the more the results of an empirical study can be generalised to actual software engineering practice. We dealt with the following facts:

- **Components and their metrics.**
  With respect to the time limitation of our study, we tried to find the components that vary in the size and the other studied metrics to the extent possible in order to make our

[9] www.objectaid.com

results more generalizable. Therefore we intentionally took the components that vary in the size and the other studied metrics in order to cover different metrics values. There is a very low threat for the statistical validity of our results (see Section 5). The obtained prediction models are validated to prevent over-fitting of the data (see Section 4.2), i.e. to enable a reasonably well-fitting prediction in case of new data. However, to examine more fine-grained distributions of the components' metrics and especially the components whose metrics' values significantly vary from the studied metrics' values (i.e. are significantly bigger than the studied metrics' values), more components need to be examined. In case of bigger components it would be interesting to see to which extent the obtained prediction models would be affected. In that case the participants would require much more time to analyse the components. Furthermore, an architectural representation would probably require hierarchical organization of the components, i.e. components having sub-components at different abstraction levels (it starts from a set of high-level components that model high level functionalities and results in a set of low-level components that combine to perform the high-level functionalities). This representation complies with the guides for software architecture definition in the series of guides for software engineering produced by the Board for Software Standardisation and Control (BSSC) of the European Space Agency (Mazza et al. 1996), for instance. Having in mind the above discussion we are aware that our results (obtained prediction models) might vary to a certain extent for new data. According to that our tool support (see Section 6 for more details) is designed to consider the predicted component's understandability values as more relative values (rather than evaluating the design by giving absolute values), i.e. in comparison to the understandability of other components in the system, that is used for identifying critical components which require more effort to be understood compared to other components in the system.

– **Studied system and its representations.**

  Regarding the studied system we chose the system that is written in Java (that the participants are familiar with), that has industrial relevance, and whose application domain is relatively known to the participants (see more details in Section 3.4.2). The architecture of the system is represented in the form of a UML component diagram that the participants are also familiar with (see Section 5). Having in mind these facts, we can say that our results might be more or less different for other potential systems depending on the extent to which the assumptions related to the chosen system are violated. For example, the results might differ for a system written in some other language that the participants are not familiar with, or some domain specific systems that the participants are totally unfamiliar with, etc. Also, architectural descriptions of software systems using component models could be created in different ways, starting from the simple descriptions of the system like box-and-line diagrams (Rozanski and Woods 2005), over semi-formal models (e.g. UML models) (Björkander and Kobryn 2003; Medvidovic et al. 2002; Robbins et al. 1998) to formal models in architecture description languages (ADLs) (Medvidovic and Taylor 2000) or domain-specific languages for architecture description (Völter 2010). More studies are necessary to examine how different architectural representations affect the understandability of components with respect to their concrete implementation.

– **Varying class sizes within components.**

  As we already mentioned above in order to generalize our results we plan to increase the number of studied components. Beside that we consider one more threat in this context, it is the size of the classes in a component. Considering general case there might be some classes that are much bigger than other classes in the system. In that case

the number of classes in a component will not appropriately capture the component size (in our case as it is mentioned in Section 3.4.2 no big deviations in the sizes of the classes exist). However that case might also be considered as inappropriate design, i.e. big classes can be divided into smaller classes that consist of one or a set of closely related functionalities. Anyway the given observation can be further examined in order to see how the deviations in the size of classes affect the obtained results.

– **Subjects.**
  It has been shown in previous research that software engineering students may provide an adequate model for the professional population (Weber et al. 2014). Even though our participants have substantial experience including the industrial background certain changes in the obtained results might be expected with experts. Studies with experts would enable us to conduct more robust analysis.

To summarize the cases in which our findings, i.e. predictions, would appropriately work, taking into account the given threats to validity, we can say the following:

– The studied system needs to be object-oriented and its application domain relatively known to the participants.
– The architectural components need to have up to 15-20 classes that do not have big deviations in their size (e.g. one very big class and several very small ones)
– The participants need to have at least a couple of years of appropriate programming experience as well as basic knowledge in the software architecture and software engineering field so that they can easily understand the code of the system together with its architecture.

In other cases, the obtained results can vary from ours to a lesser or greater extent.

6 Tool support

6.1 Background

In our previous work (position paper Stevanetic et al. 2014), we presented an integration of a semi-automated DSL-based abstraction of architectural component models and understandability related software metrics. An overview of our approach is shown in Fig. 6. The black part refers to the semi-automated DSL-based architectural abstraction while the red part marked with dashed lines refers to the understandability metrics.

Regarding the black part, we defined a DSL that enables architectural abstractions from class models, which can be automatically extracted from the source code, into architectural component models. First, a class model from the system's source code is extracted. Starting from a class model, a UML component model is generated using the architectural abstraction specification defined in the DSL code. In this way the traceability information that links the class models and component models can be preserved. Furthermore, the approach supports consistency checks that are based on the automatically generated traceability information that link the DSL, the class model, and the component model of the system. For instance, the source code classes that are not covered by the architecture abstraction specification or connectors that are defined in the architecture specification but where no relation exists in the source code classes are checked. This enables having an "up-to-date" component model that reflects the source code (i.e. all source code classes are mapped

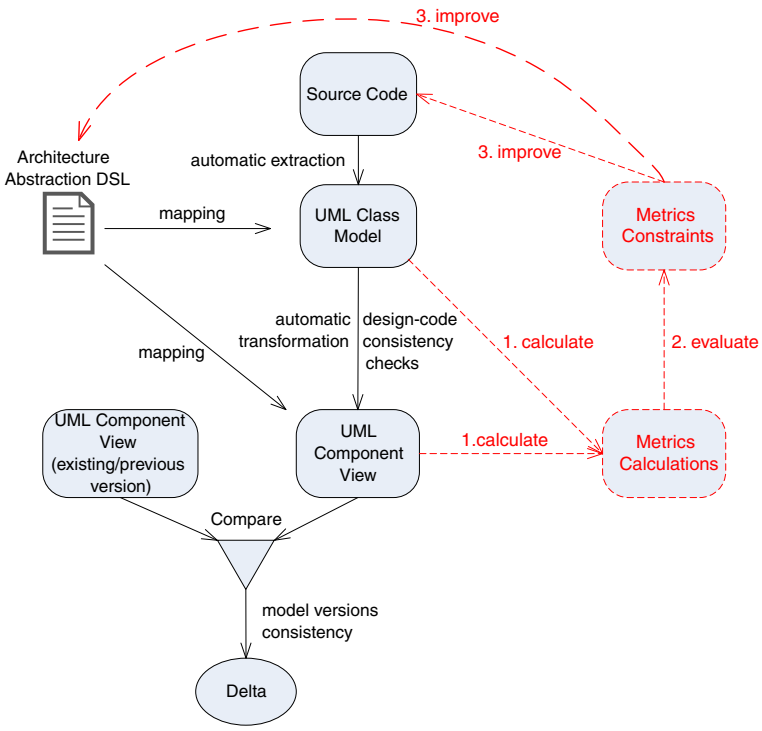


Fig. 6 Integration of the understandability related metrics in the DSL-based architecture abstraction approach (reused from Stevanetic et al. 2014)

to their respective components). The given approach also supports the software architect throughout the evolution of a software system by allowing him/her to compare different component models (see the bottom of the figure) and to maintain them in correspondence with the source code over time.

The red part marked with dashed lines describes the integration of the understandability metrics for generated component models. For example, they provide an indicator whether a component model is growing too large or other similar guidelines. Firstly, the metrics calculations are extracted from both the class model and the component model. The obtained metrics values are then evaluated with regard to different metrics constraints. Metrics constraints represent a set of rules defined on metrics values that need to be satisfied. In our case they are defined based on our empirical evaluations and also take into account some additional considerations (see Stevanetic et al. 2014). In case that some metrics values do not satisfy the corresponding constraints the architectural abstraction DSL or the source code can be improved in order to resolve the inconsistencies that occurred.

In this paper, we investigate how our empirical findings can be combined with existing empirical evaluations and how we can provide a corresponding tool support. In that context, we have found the work by Bouwers et al. (2011), who studied the analyzability of component models. Taking into account the findings from Bouwers et al., who found that the components should be balanced in size in order to facilitate the system's analyzability, we have argued that balanced values for the components' understandability effort can

facilitate the analyzability of the whole system. In contrast to our previous position paper (Stevanetic et al. 2014), where the idea about the balanced understandability of components is just mentioned, in this article we further elaborate on concrete calculations of the components' analyzability based on the integration of our new understandability effort prediction models (i.e. the ones that use the hierarchical understandability metrics, see Section 4.2) and the metrics provided by Bouwers et al.

6.2 Architecture analyzability metric

In this section, we briefly explain our new analyzability metric for component models that is based on the integration of our understandability related prediction models in the analyzability metric defined by Bouwers et al. (2011). Furthermore, we elaborate on calculations related to how much each of the rules used to specify architectural abstractions contribute to the overall understandability of components.

Namely, Bouwers et al. (2011) defined a metric for quantifying the analyzability of software architectures. The metric is called Component Balance (CB) and is defined as the product of two metrics: System Breakdown (SB), which measures whether a system is decomposed into a reasonable number of components and Component Size Uniformity (CSU), which measures whether the components are all reasonably sized. The SB metric is based on the number of components in the system and it is driven by logic that both high and low number of components hinders analyzability. For example, having only one component is bad since the structure of the code does not provide any hints as to where functionality is implemented. On the other hand, many small components do not provide a software engineer with sufficient clues as to which component should be chosen to inspect.

The CSU metric captures how uniformly the volume of the system is distributed over its components and it is based on the Gini coefficient (see Bouwers et al. 2011). To provide maximal discriminative power to a software engineer, a system should be decomposed into a limited number of components of roughly the same size.

Now, we can explain our idea of combining the metric given above with our empirical findings. Namely one of the main drawbacks of the given metric is that it captures the system's structural decomposition only using the size metric of a component. Dependencies between components are not taken into account which is important because the size is not enough to fully determine phenomena of understanding (see Section 4.2). To improve the situation, we propose to use our "understandability metric" related to the obtained prediction models instead of a simple size metric. In that case both the internal structure of a component and its dependencies to other components are captured (see Section 3.4.3)

Therefore, instead of using the size metric for the CSU metric, we can use the metric obtained from our prediction models as follows:

$$\begin{aligned} Understandability(c) &= 1.1517 \times MSC(c) - 0.7889 \times NAC(c) + 0.5555 \times DMC(c) \\ &\quad + 2.5473 \times CRW(c) - 0.1371 \times DMH(c) \\ CSU(C) &= 1 - Gini(\{Understandability(c) : c \in C\}) \end{aligned}$$

We picked the second prediction model from Table 15 but any of the models can be used since they have almost the same explanatory power (accuracy). Using the adapted CSU metric, we can now calculate the adapted CB metric as a new analyzability metric.

6.3 Integration of the metrics in the tool

In this Section, we explain how our concepts are embodied in the tool using a concrete example. In particular, we demonstrate how to create component models with reasonable analyzability level by incrementally improving an initial component model of the system. In addition, we show how the tool can be used in detecting the changes that exist between different component models that affect their different analyzability levels. For the calculations given below we used the prediction model 1 given in Table 15 (any other model provided in Table 15 can be used).

The calculation of all required metrics are added in the Metrics Calculation part of our tool (see Fig. 6). The Metrics Calculation part is developed using the custom validation features for Xtext based projects (for more information please refer to http://eclipse.org/Xtext/documentation/). Particularly, the DSL specification for the component models abstraction is written in a file. The file can then be explicitly validated via the menu option in which case the validator class that pursues the metrics calculations is called. The calculated metrics are written in a file which is then processed using the R programming language script. The final output represents the barplot diagram of the CSU metric together with the hierarchical metrics used in the prediction models for all components in the system. Additionally, the CB and SB metrics are calculated for the whole system. Furthermore, our tool supports the calculations on how much each of the architectural rules, used to specify a DSL-based architectural abstraction specification, contributes to the understandability of a given component.

To demonstrate our tool we use the example of the Frag system. Frag[10] is a dynamic programming language implemented in Java, specifically designed for being a tailorable language, building Domain-Specific Languages (DSLs), supporting Model-driven Software Development, and for being easily embeddable in Java. To generate the initial component model of the system, we studied the source code of the system and the corresponding class model that is automatically generated from the system's source code. To ease this task, we imported the source code in an Eclipse IDE. The understanding of the system was facilitated by the fact that one of the authors of the paper is also the author of Frag. After initial examinations, we created the initial architecture abstraction specification of the system consisting of 6 components (*Core*, *Interpreter*, *Command Objects*, *Parser*, *Exceptions*, and *MDSD*) is generated. The given 6 components are found to represent the major functionalities and/or concerns in the system. The DSL specification of the initial component model is shown in Fig. 7. Figure 8 shows the distribution of the calculated metrics values for the initial component model. The CB metric value for the model is 0.32.

From the DSL specification (developed using Xtext2), we can see that different architectural rules are used to write the architecture abstraction. For example, rules operating on source code artefacts relate different source code artefacts packages, classes, and interfaces to an architectural component (e.g. *Package* rule shown in Fig. 7 which selects everything inside a specific package), rules utilizing relationships between source code artefacts relate an architectural component to the source code artefacts that have specific relationships to the given source code artefact like sub- and super-type relations for classes and interfaces, interface realizations, and other dependencies (e.g. *Uses* rule shown in Fig. 7), etc. Complex rules definitions are supported through the implementation of the three set operations

[10] http://frag.sourceforge.net/

```
Component Parser consists of {Package (root.frag.parser) and not
      {Uses (root.frag.core.Interp) and Package (root.frag)}}
Component CommandObjects consists of
      {{Package (root.frag.objs) or Package (root.frag.files)}
      and not {Uses (root.frag.core.Interp) and Package (root.frag)}}
Component Interpreter consists of {Class (".*Interp") or
      {{Uses (root.frag.core.Interp) and Package (root.frag)}
      and not Class (root.frag.core.Dual)}}
Component Core consists of {Package (root.frag.core) and not
      {Class (".*Interp") or
      {Uses (root.frag.core.Interp) and Package (root.frag)}}}
Component MDSD consists of {Package (root.mdsd)}
Component Exception consists of {Package (root.frag.exceptions)}
```

Fig. 7 Initial component model - DSL specification

union (or), intersect (and), and difference (and not) which are all used in Fig. 7. Detailed information about all rules and how they are derived can be found in Haitzer and Zdun (2014).

Here, we shortly explain how the metrics for each architectural rule in the DSL specification are calculated. In particular, each rule specifies a set of source code elements to be added to a given component. Hence, for each rule we can calculate the above mentioned set of metrics as we do for the system's main components. Let us explain how exactly these calculations are done for Component Interpreter in the example from Fig. 7. The rules are evaluated in the following way: first the highest level rule is evaluated. In our example, the highest level rule is the *or* composition rule which actually specifies all classes in the component. Next the left part of the previously analysed composition rule is evaluated, which is here the *Class* rule. Then the right part of the *or* rule is analysed. Since it consists of

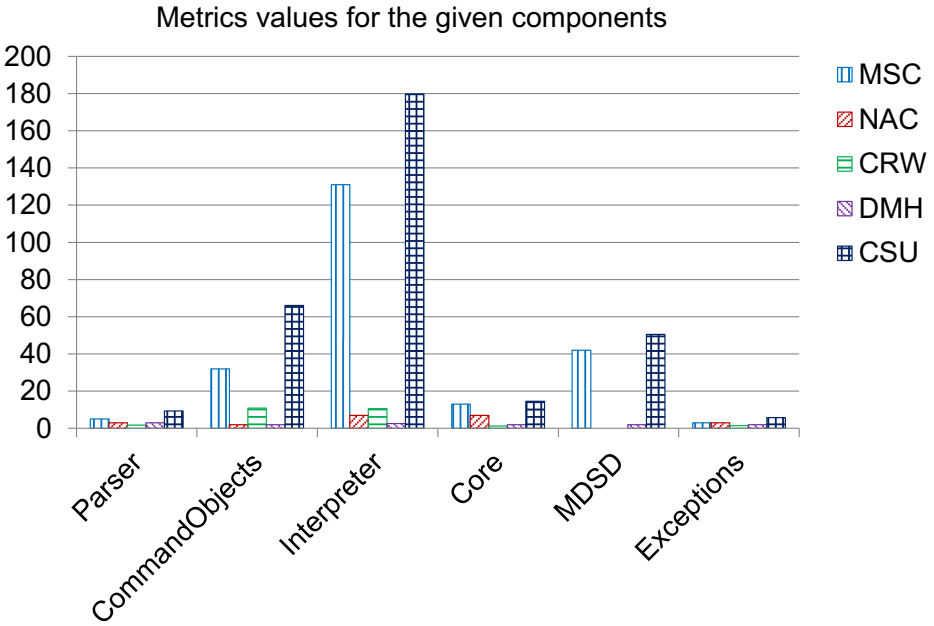


Fig. 8 Initial component model - metrics

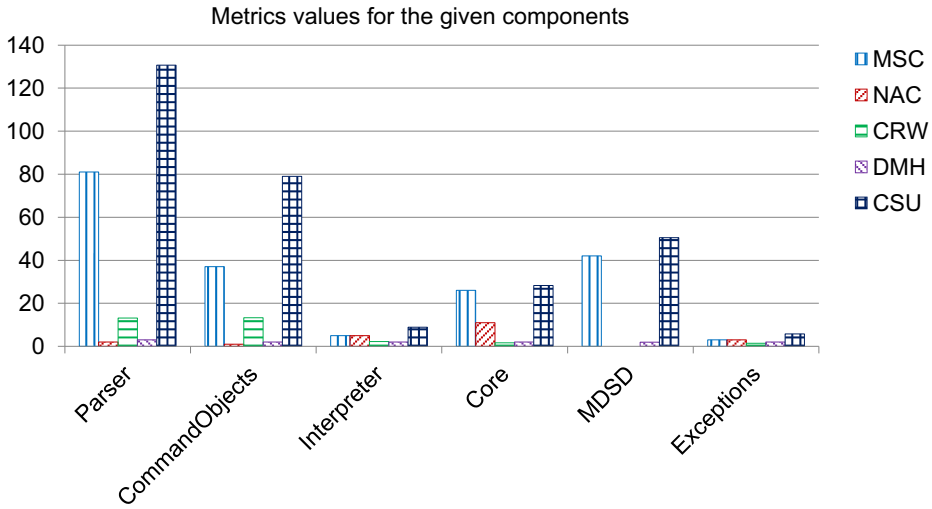


Fig. 9 Changed component model 1 - metrics

further composition rules the next highest level rule will be picked. In the given example it would be the *and not* composition rule which specifies all classes contained in both root.frag.core.Interp and root.frag packages without the class root.frag.core.Dual. Next the left part of the *and not* rule would be evaluated, which corresponds to the *and* composition rule. Then the left and right parts of the *and* rule are examined which corresponds to the *Uses* rule and the *Package* rule, respectively. At the end the right part of the (*and not*) rule is evaluated which corresponds to the *Class* rule.

From Fig. 8 we can see that Component *Interpreter* has significantly higher CSU metric than the other components, mainly because of the high number of classes that it contains (see the MSC metric in the figure). Namely, we want that Component *Interpreter* includes all classes which name ends with "Interp" or those that are tightly coupled to the class *Interp* from the *core* package (see the DSL specification of Component *Interpreter* in Fig. 7). However, by examining how much each architectural rule in Component *Interpreter* affects the CSU metric value for the whole component we find that the *Uses* and *Package* rules and therefore the *and* rule that connects them have the high CSU metric values because of the high number of classes that those rules produce (see Fig. 10). Consequently, a lot of classes are assigned to Component *Interpreter* that should not belong there. To improve the situation we change the DSL specification for Component *Interpreter* so that the tightly coupled classes to the class *Interp* now include those that both use the given class and are used by it. Furthermore, the *Package* rule now limits finding the tightly coupled classes to the *core* package. The metrics for the new component model are shown in Fig. 9. The CB metric value for the new component model is 0.37 (Fig. 10).

By looking at Fig. 9 we can see that now Component *Parser* has significantly higher CSU metric value than the other components. The high CSU metric value for Component *Parser* is mainly affected by the relatively high number of classes that this component contains (see the MSC metric values in Fig. 9) compared to the other components. Therefore, dividing it into several smaller components would probably improve the situation, i.e. increase the overall CB metric value for the system. From the point of the SB metric value which decreases if the number of components is greater or smaller than 8 (see Section 6.2),

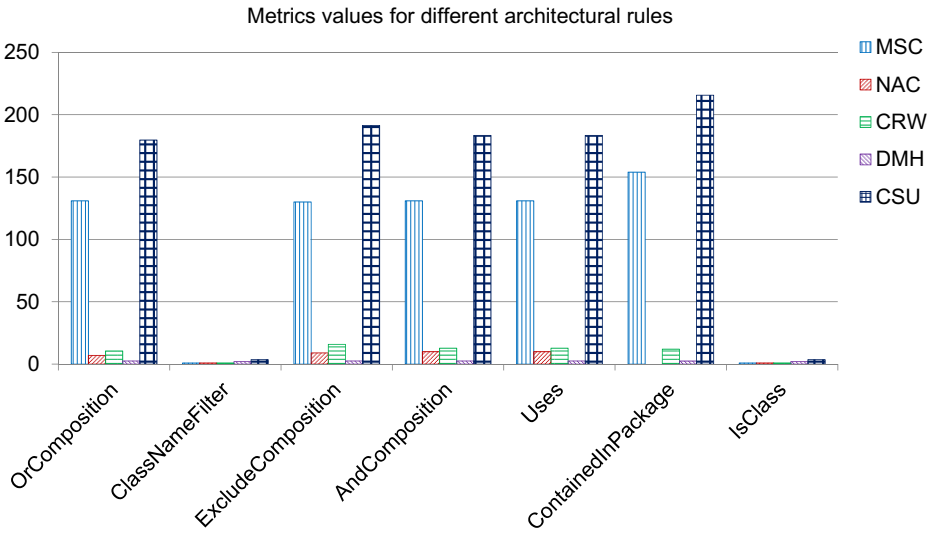


Fig. 10 The impact of each architectural rule on understandability – component interpreter from Fig. 7

dividing the *Parser* component into 2 or 3 smaller components would increase its value since the current number of components in the system is 6.

By examining the *Parser* component we have found that it would make sense to divide the component into two components *ParserRules* and *ParsedObjects*. Namely, the parser used in Frag uses lexical parsing approach based on the composition of rule definitions that are similar to EBNF. A rule is a description of the situation when the rule matches (a matcher) plus an action that is taken when the rule applies. The result is a tokenized list of parsed elements. Since the concept of rules is important it makes sense to create a separate component for handling the parsing rules (Component *ParserRules*). The second new component *ParsedObjects* relates to the list of the parsed elements that roughly corresponds to the Abstract Syntax Tree (AST) in other parsing approaches. Having a separate component for the AST of the parsed code makes sense because the AST structure can contain additional information that need to be managed, i.e. the information related to the subsequent processing, e.g. contextual analysis, etc. The calculated metrics for the new component model are shown in Fig. 11. The CB metric for the new component model is 0.55 and it is increased by 0.18 compared to the previous component model.

The distribution of the metrics values for the new components can further be examined in order to make additional possible improvements. For example, we can examine if it would make sense to divide Component *CommandObjects* that has the highest CSU metric value in the newly generated component model. By examining the corresponding classes of the *CommandObjects* component we have found that it can be further divided into two components *FileCommands* and *NonFileCommands* that correspond to the commands for handling files and other non-file related commands. The calculated metrics for the generated component model that encompasses all mentioned changes are shown in Fig. 12. The CB metric for the component model that consists of all mentioned changes is 0.64 and it is increased by 0.09 compared to the component model it is adapted from.

The example given above shows how we can gradually improve the analyzability of the initial component model by making changes in the DSL and judge the analyzability of the

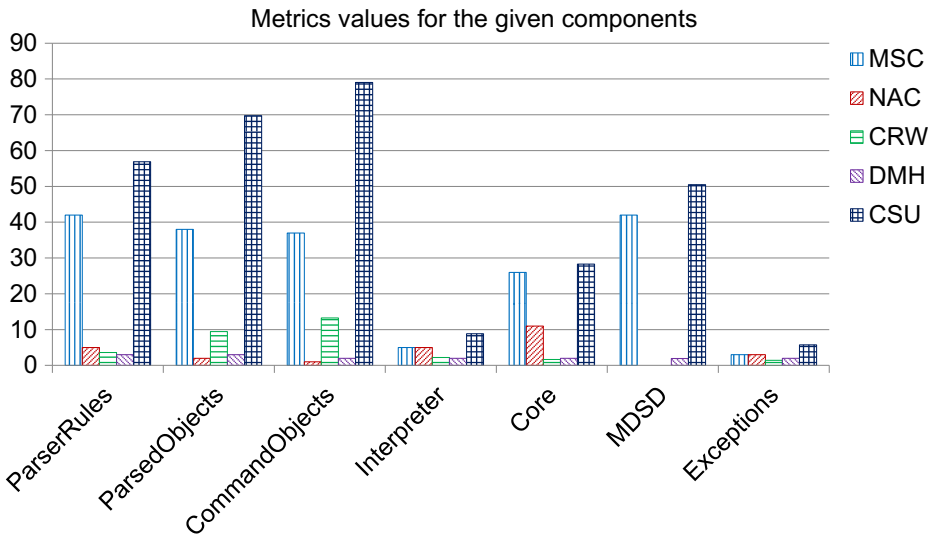


Fig. 11 Changed component model 2 - metrics

component model created with the DSL using the given metrics calculations. It would be also possible to make source code changes and observe their effect on the analyzability of the generated component model.

Let us now explain how our tool can support finding the changes that affect different analyzability levels of two different component models. To appropriately capture different changes in the system that can affect changes in the given metrics values, our tool supports finding the changes at three different levels in the system: source code changes, component model changes, and DSL changes. Regarding the source code changes, by comparing

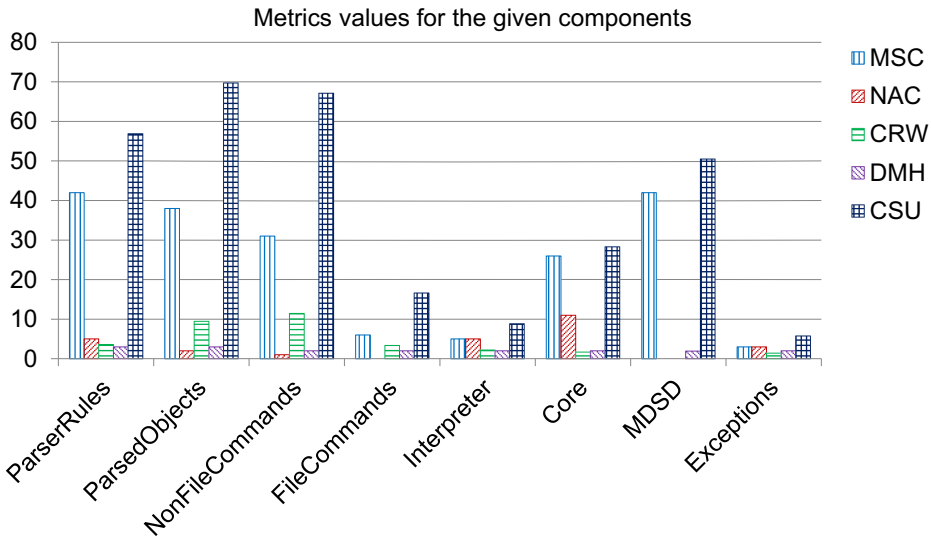


Fig. 12 Changed component model 3 - metrics

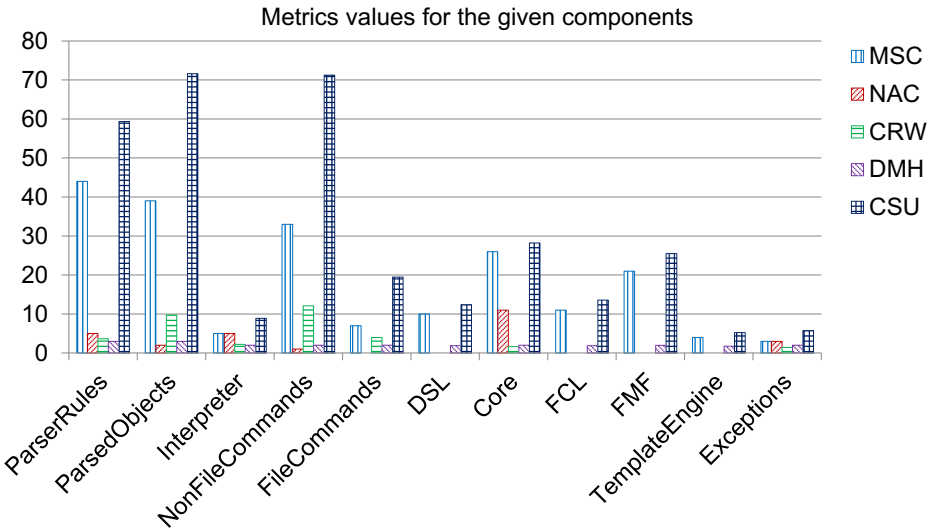


Fig. 13 Component model of the frag version 0.8 - metrics

different source code versions of the system an architect or developer can find which source code elements are added to the system, deleted from it, or changed. By comparing the component models, a user can find the differences in the realized classes contained in each of the components, i.e. if some classes are added to a given component or removed from it (which cannot be seen using the source code comparisons). Furthermore, by comparing the component models it is possible to find new components in the system or those that are deleted from it. The DSL comparisons show the differences in the architectural rules used to specify which system parts are assigned to a component. By combining the given three kinds of comparisons a user can precisely determine the differences in the compared system's versions as well as in each of the compared components. To realize the mentioned kinds of comparisons we used and extended the Eclipse IDE's features for the comparisons of different resources. Furthermore, to enable finding the component model changes we created the corresponding serialized representations of the examined component models that contain the fully qualified names of all source code artefacts related to each component.

We compare two component models of the Frag system, one that corresponds to the 0.7 version of the system and another one that corresponds to the 0.8 version. The metrics values for the first component model are shown in Fig. 12 while for the second component model they are shown in Fig. 13.

To find which changes in the system caused the different metrics levels of the given component models we compared their DSLs, their source code, and the classes they contain. From the DSL comparison we can see if there is a difference in the architectural rules used to specify the given views, if new components are added, or if some of them are deleted. By comparing the two DSLs we have found that Component *MDSD* from the first view (version 0.7) is replaced with 3 new Components *DSL*, *FCL*, and *FMF* in the second view (version 0.8). Otherwise, no changes in the DSL of the other components have been found. By comparing the components' contained classes we have found that the package

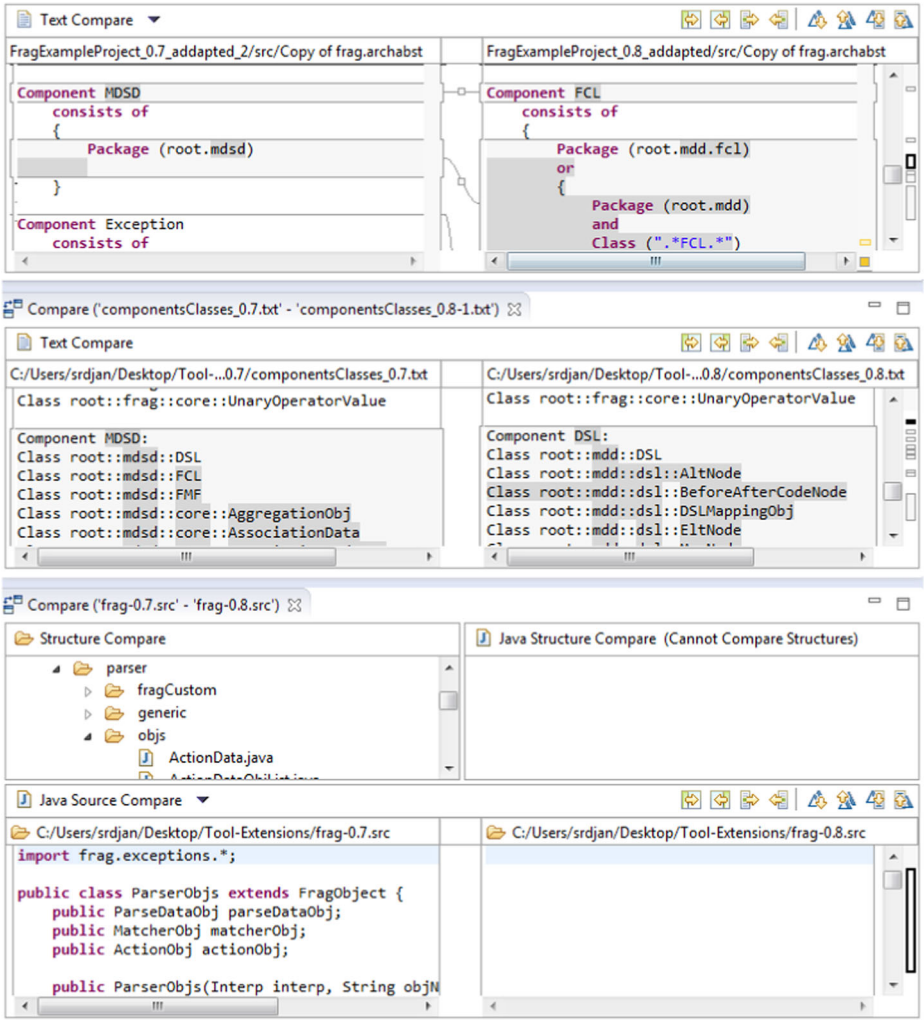


Fig. 14 Change impact analysis comparisons

core in the first view is renamed to the package *fmf* in the second view.[11] Using the same comparison we have found which classes are newly added to the components or which classes are deleted from them. Finally by comparing the source code folders we can find which classes changed their source code. In our example we have found that all classes have been changed to a certain extent. Figure 14 shows the visualization of the given comparisons views (DSL, classes, and source code comparisons) based on the Eclipse features for the resource comparisons. Table 19 provides an overview of all found changes.

[11]The classes contained in the components are compared using their full qualified names that include the names of all packages that contain a given class.

Table 19 Overview of all found changes

	Added classes	Deleted classes	Changed classes	Version 0.7	Version 0.8
ParserRules	2	0	41	yes	yes
ParsedObjects	2	1	37	yes	yes
NonFileCommands	2	0	31	yes	yes
FileCommands	1	0	6	yes	yes
Interpreter	0	0	5	yes	yes
MDSD	0	0	43	yes	no (divided into the DSL, FCL, and FMF components)
Core	0	0	27	yes	yes
Exceptions	0	0	3	yes	yes
DSL	0	0	10	no	yes
FCL	0	0	11	no	yes
FMF	0	0	11	no	yes
TemplateEngine	0	0	4	no	yes

By comparing the corresponding metrics values for the given two component models, we can say that, except newly introduced components, they show very small differences. This is not in accordance with the number of changes that we found (see Table 19). However, after examining the given changes we have found that the only real changes in terms of adding, deleting or changing some functionality in the system or its part are related to the added or deleted classes in the components (Columns "Added Classes" and "Deleted Classes" in Table 19). The changes in other classes are mostly syntactic changes or code refactoring related changes that do not affect classes' external behaviour.

The integrated metrics benefit from the architecture abstraction tool in the way that the later provides an "up-to-date" architectural component model that reflects the source code (i.e. all source code classes are mapped to their respective components) that is necessary for the metrics calculations. This way, as we demonstrated, the architects or developers can gradually improve the architecture by making the changes in the source code or in the architecture abstraction DSL and judge the analyzability of the architecture created with the DSL. To perform such improvements the architect or developer can partially benefit from the given metrics calculations provided for each component. For example, as we demonstrated, components that have a large number of classes (i.e. the high MSC metric value) can be broken into several new components. Similarly, components with high coupling can be modified by rearranging their classes with other classes to which they have a strong coupling or by refactoring the classes source code to reduce their coupling to the classes in other components. Modification steps, of course, require manual effort and human expertise.

The metrics calculations for each component as a whole provide useful information on what is its understandability level and what is it affected from. The metrics calculations for each architectural rule related to a given component help an architect or developer to grasp how much different source code artefacts that constitute a given component contribute to its understandability, which rules contribute the most to the limited understandability, etc. (as demonstrated for Component *Interpreter* above). It can help during performing changes in the DSL of a component in terms that an architect can assess in which direction and

approximately how much the understandability of a component will change if some rules are changed.

7 Conclusions and future work

In this article, we provide an extended description of the analysis and results obtained in our previous work (Stevanetic and Zdun 2016) consisting of a more detailed description of the studied metrics, applied statistical techniques, and obtained findings. In addition, we present a new metric for measuring the analyzability of component models based on the integration of our empirical findings and the existing observations related to them, i.e. concretely the existing work on the analyzability related software metrics proposed by Bouwers et al. (2011). Furthermore, we present significant tool extensions compared to our previous work (Stevanetic et al. 2014) including the realization of the new analyzability metric by integrating our previous tools for supporting software evolution using a DSL-based architecture abstraction with the obtained empirical findings. Our tool extensions enable the calculations of how much each of the architectural rules used to specify a DSL-based architectural abstraction specification contributes to the understandability of components and also enable change impact analysis, i.e. the identification of changes in the system that affect different analyzability levels of the component models.

Regarding our empirical findings, we studied the understandability of architectural components using a number of component level metrics including the package based metrics defined by Martin (2003), information theory graph based metrics, and the corresponding counting-based graph based metrics defined by Allen (2002) and Allen et al. (2007), and hierarchical understandability metrics introduced in the work by Hwa et al. (2009), as well as the personal factors of participants like experience and expertise, and the combinations of both personal factors and component level metrics. The understandability of component models is measured through the time that the participants spent on understanding the components, and then predicted using the above given component level metrics and participants' experiences. On the one hand, the prediction models that consider the hierarchical understandability metrics are significantly better in predicting the understandability effort than the models obtained using other component level metrics or the models that include the participants' experiences. On the other hand, those models are not significantly different in the prediction from the models that combine both the component level metrics and the participants' experiences. This means that from all studied models it is enough to consider the hierarchical understandability metrics for the prediction. This result is from our point of view intuitive, as those metrics are originally designed to assess the understandability of the modular design of a system. The participants' experience is also important and can predict a significant amount of variance in the data but the obtained models are not as accurate as the models that use the component level metrics, i.e., the metrics related to the system itself.

The obtained empirical findings are integrated in the tool that supports the synchronized evolution of the architecture and source code of the system. While the DSL-based architecture abstraction approach enables users to keep source code and architecture consistent, the given metrics extensions enables them, while working with the DSL or source code, to continuously judge and improve the analyzability of the architectural component model they create with the DSL. To further support users in performing adequate changes in the DSL or source code and understanding their impacts on the understandability of a given component, we calculate the given metrics for each architectural rule used to define the DSL-based specification of that component. In that way users can grasp how much different source code

artefacts that constitute a given component affect its understandability. Beside improving the analyzability of component models, our approach also supports change impact analysis, i.e. finding which changes in the system's source code or the DSL-based architectural specification correspond to the changes in the observed metrics values. The applicability of our approach is shown using a case study of an existing open source system.

From the academic point of view we believe that our study can serve as a good starting point for future studies on the understandability of architectural components and component models, but also other kinds of software models. The used instruments and applied statistical techniques provide insight in how the understandability can be appropriately measured and predicted which can help in devising new empirical studies and experiments. From the practitioner's point of view, the results of our study show which factors and metrics are important for assessing the understandability of architectural components in relation to the system implementation and in how far those metrics can predict the understandability. The understandability effort (time) for new architectural components can be assessed based on the complexity of their implementation. Absolute values for the measured understandability effort for new components are considered to be accurate only for the systems similar to the studied one. In other cases the assessment can vary to a lesser or greater extent. Our tool support facilitates the application of the obtained empirical finding in practice. It is designed to consider the predicted component's understandability values as more relative values (rather than evaluating the design by giving absolute values), i.e. in comparison to the understandability of other components in the system, that is used for identifying critical components which require more effort to be understood compared to other components in the system. In that way, it can be more or less successfully applied for the systems and components which size and complexity differ from the studied one.

In our future work, we plan to include experts with many years of experience and compare the results with the ones obtained here. We also plan to examine more components, including bigger ones, that would enable us to construct more robust prediction models. However, tackling these challenges is not an easy task since it requires a lot of resources in terms of time and money.

Acknowledgements This work was supported by the Austrian Science Fund (FWF), Project: P24345-N23. We thank Dr. Nina Senitschnig from the Department of Statistics and Operations Research, for valuable suggestions and help related to the statistical analysis pursued.

References

Aguilar ER, García F, Ruiz F, Piattini M (2007) An exploratory experiment to validate measures for business process models. In: Rolland C, Pastor O, Cavarero J-L (eds) RCIS, pp 271–280

Albrecht A, Gaffney JE (1983) Software function, source lines of code, and development effort prediction: a software science validation. IEEE Trans Softw Eng SE-9:639–648

Allen EB (2002) Measuring graph abstractions of software: an information-theory approach. In: IEEE METRICS. IEEE Computer Society, Washington, p 182

Allen EB, Gottipati S, Govindarajan R (2007) Measuring size, complexity, and coupling of hypergraph abstractions of software: an information-theory approach. Software Quality Control 15:179–212

Babar MA, Lago P (2009) Editorial: design decisions and design rationale in software architecture. J Syst Softw 82:1195–1197

Bansiya J, Davis CG (2002) A hierarchical model for object-oriented design quality assessment. IEEE Trans Softw Eng 28:4–17

Barnes JM, Garlan D, Schmerl BR (2014) Evolution styles: foundations and models for software architecture evolution. Softw Syst Model 13(2):649–678

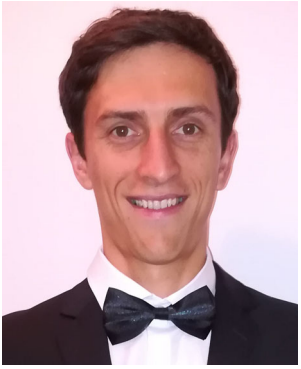
Basili V, Briand L, Melo W (1996) A validation of object-oriented design metrics as quality indicators. IEEE Trans Softw Eng 22:751–761
Bass L, Clements P, Kazman R (1998) Software architecture in practice. Addison-Wesley Longman Publishing Co. Inc., Boston
Belsley D (1991) Conditioning diagnostics, collinearity and weak data in regression. Wiley-Interscience, Hoboken
Belsley DA, Kuh E, Welsch RE (1980) Regression diagnostics: identifying influential data and sources of collinearity (Wiley Series in Probability and Statistics). Wiley-Interscience, Hoboken
Bhattacharya P, Iliofotou M, Neamtiu I, Faloutsos M (2012) Graph-based analysis and prediction for software evolution. In: ICSE'12, pp 419–429
Björkander M, Kobryn C (2003) Architecting systems with UML 2.0. IEEE Softw 20:57–61
Boehm B (1978) Characteristics of software quality. North-Holland Pub. Co., TRW series of software technology
Booch G (1994) Object-oriented analysis and design with applications, 2nd edn. Benjamin-Cummings Publishing Co. Inc., Redwood City
Bouwers E, Correia JP, Deursen A, Visser J (2011) Quantifying the analyzability of software architectures. In: 2011 Ninth working IEEE/IFIP conference on software architecture. IEEE, Piscataway, pp 83–92
Briand L, Labiche Y, Di Penta M, Yan-Bondoc H (2005) An experimental investigation of formality in uml-based development. IEEE Trans Softw Eng 31:833–849
Burnham K, Anderson D (2002) Model selection and multimodel inference: a practical information-theoretic approach. Springer, Berlin
Canfora G, García F, Piattini M, Ruiz F, Visaggio C (2005) A family of experiments to validate metrics for software process models. J Syst Softw 77(2):113–129
Cardoso J (2006) Process control-flow complexity metric: an empirical validation. In: IEEE international conference on services computing, 2006. SCC '06, pp 167–173
Chidamber S, Kemerer C (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20:476–493
Clements P, Garlan D, Bass L, Stafford J, Nord R, Ivers J, Little R (2002) Documenting software architectures: views and beyond. Pearson Education, London
Clements P, Bachmann F, Bass L, Garlan D, Ivers J, Little R, Nord R, Stafford J (2003) Documenting software architectures: views and beyond. Addison-wesley, Boston
Cohen J (1988) Statistical power analysis for the behavioral sciences. Lawrence Erlbaum, New Jersey
Cuesta CE, Navarro E, Perry DE, Roda C (2013) Evolution styles: using architectural knowledge as an evolution driver. Journal of Software: Evolution and Process 25(9):957–980
Dalgaard P (2004) Introductory statistics with r. Springer, Berlin
Dromey RG (1995) A model for software product quality. IEEE Trans Softw Eng 21:146–162
Dromey RG, McGettrick AD (1992) On specifying software quality. Softw Qual J 1:45–74
Dugerdil P, Niculescu M (2014) Visualizing software structure understandability. In: 23rd Australian software engineering conference, ASWEC 2014, Milsons Point, April 7-10, 2014, IEEE Computer Society, Sydney, pp 110–119
Ebel R, Frisbie D (1991) Essentials of educational measurement. Prentice Hall, Upper Saddle River
Egyed A (2004) Consistent adaptation and evolution of class diagrams during refinement. In: Fundamental approaches to software engineering, 7th international conference, FASE 2004, ETAPS 2004 Barcelona, Spain, vol. 2984 of Lecture Notes in Computer Science. Springer, Berlin, pp 37–53
Elish MO (2010) Exploring the relationships between design metrics and package understandability: a case study. In: ICPC. IEEE Computer Society, Washington, pp 144–147
Cook RD (1977) Detection of Influential Observation in Linear Regression, Technometrics, 19(1):15–18. https://doi.org/10.1080/00401706.1977.10489493
Fenton NE, Pfleeger SL (1998) Software metrics: a rigorous and practical approach, 2nd edn. PWS Publishing Co., Boston
Fenton NE, Ohlsson N (2000) Quantitative analysis of faults and failures in a complex software system. IEEE Trans Softw Eng 26:797–814
Field A, Miles J, Field Z (2012) Discovering statistics using r. SAGE Publications, Thousand Oaks
Genero Bocco M, Moody DL, Piattini M (2005) Assessing the capability of internal metrics as early indicators of maintenance effort through experimentation: Research articles. J Softw Maint Evol 17:225–246
Ghezzi C, Jazayeri M, Mandrioli D (2002) Fundamentals of software engineering, 2nd edn. Prentice Hall PTR, Upper Saddle River
Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Trans Softw Eng 26:653–661

Gupta V, Chhabra JK (2009) Package coupling measurement in object-oriented software. J Comput Sci Technol 24:273–283
Gupta V, Chhabra JK (2012) Package level cohesion measurement in object-oriented software. J Braz Comp Soc 18(3):251–266
Haitzer T, Zdun U (2014) Semi-automated architectural abstraction specifications for supporting software evolution. Sci Comput Program 90(Part B, 0):135–160. Special Issue on Component-Based Software Engineering and Software Architecture
Harrison R, Counsell SJ, Nithi RV (1998) An evaluation of the mood set of object-oriented software metrics. IEEE Trans Softw Eng 24:491–496
Hofmeister C, Nord R, Soni D (2000) Applied software architecture. Addison-Wesley Professional, Boston
Hwa J, Lee S, Kwon Y-R (2009) Hierarchical understandability assessment model for large-scale oo system. In: Software engineering conference, 2009. APSEC '09. Asia-Pacific, pp 11–18
Jansen A, Bosch J (2005) Software architecture as a set of architectural design decisions. In: Proceedings of the 5th working IEEE/IFIP conference on software architecture, WICSA '05. IEEE Computer Society, Washington, pp 109–120
Kabacoff R (2011) R in action: data analysis and graphics with r. Manning Pubs Co Series, Manning
Kampenes VB, Dybå T, Hannay JE, Sjøberg DIK (2007) Systematic review: a systematic review of effect size in software engineering experiments. Inf Softw Technol 49:1073–1086
Kitchenham BA, Pfleeger SL, Pickard LM, Jones PW, Hoaglin DC, El Emam K, Rosenberg J (2002) Preliminary guidelines for empirical research in software engineering. IEEE Trans Softw Eng 28:721–734
Kobayashi M, Sakata S (1990) Mallows' Cp criterion and unbiasedness of model selection. J Econ, Elsevier 45(3):385–395. <https://ideas.repec.org/a/eee/econom/v45y1990i3p385-395.html>
Konersmann M, Durdik Z, Goedicke M, Reussner RH (2013) Towards architecture-centric evolution of long-living systems (the advert approach). In: Kruchten P, Koziolek A, Nord RL (eds) QoSA. ACM, New York, pp 163–168
Kruchten P (1995) The 4 + 1 view model of architecture. IEEE Softw 12:42–50
Lindland OI, Sindre G, Sølvberg A (1994) Understanding quality in conceptual modeling. IEEE Softw 11:42–49
Losavio F, Chirinos L, Lévy N, Ramdane-Cherif A (2003) Quality characteristics for software architecture. Journal of Object Technology 2(2):133–150
Lungu M, lanza M, Girba T (2006) Package patterns for visual architecture recovery. In: Proceedings of the 10th European conference on software maintenance and reengineering 2006. CSMR 2006, pp 10–196
Ma Y, He K, Du D, Liu J, Yan Y (2006) A complexity metrics set for large-scale object-oriented software systems. In: Proceedings of the Sixth IEEE international conference on computer and information technology, CIT '06. IEEE Computer Society, Washington, p 189
Malaiya YK, Denton J (2000) Module size distribution and defect density. In: Proceedings of the 11th international symposium on software reliability engineering, ISSRE '00, IEEE Computer Society, p 62
Maqbool O, Babri H (2007) Hierarchical clustering for software architecture recovery. IEEE Trans Softw Eng 33:759–780
Martin RC (2003) Agile software development: principles, patterns, and practices. Prentice Hall PTR, Upper Saddle River
Mazza C, Fairclough J, Bryan M, Daniel P, Adriaan S, Richard S, Michael J, Alvisi G (1996) Software engineering guides. Prentice-Hall International, UK
McCabe TJ (1976) A Complexity Measure. IEEE Trans Softw Eng 2(4):308–320. https://doi.org/10.1109/TSE.1976.233837
Medvidovic N, Taylor RN (2000) A classification and comparison framework for software architecture description languages. IEEE Trans Softw Eng 26:70–93
Medvidovic N, Rosenblum DS, Redmiles DF, Robbins JE (2002) Modeling software architectures in the unified modeling language. ACM Trans Softw Eng Methodol 11(1):2–57
Mendling J (2008) Metrics for process models: empirical foundations of verification, error prediction, and guidelines for correctness, 1st edn. Springer Publishing Company, Incorporated, New York
Mohagheghi P, Conradi R, Killi OM, Schwarz H (2004) An empirical study of software reuse vs. defect-density and stability. In: Proceedings of the 26th international conference on software engineering, ICSE '04. IEEE Computer Society, Washington, pp 282–292
Moody DL (1998) Metrics for evaluating the quality of entity relationship models. In: Proceedings of the 17th international conference on conceptual modeling, ER '98. Springer, London, pp 211–225
Moody DL (2003) Measuring the quality of data models: an empirical evaluation of the use of quality metrics in practice. In: Ciborra CU, Mercurio R, de Marco M, Martinez M, Carignani A (eds) ECIS, pp 1337–1352
Morasca S (1999) Measuring attributes of concurrent software specifications in petri nets. In: Software metrics symposium, 1999. Proceedings. Sixth International, pp 100–110

Nissen ME (1998) Redesigning reengineering through measurement-driven inference. MIS Q 22:509–534
Oreizy P, Gorlick MM, Taylor RN, Heimbigner D, Johnson G, Medvidovic N, Quilici A, Rosenblum DS, Wolf AL (1999) An architecture-based approach to self-adaptive software. IEEE Intell Syst 14:54–62
Otero MC, Dolado JJ (2004) Evaluation of the comprehension of the dynamic modeling in uml. Inf Softw Technol 46(1):35–53
O'brien RM (2007) A caution regarding rules of thumb for variance inflation factors. Qual Quant 41(5):673–690
Pacione MJ, Roper M, Wood M (2004) A novel software visualisation model to support software comprehension. In: 11th working conference on reverse engineering, pp 70–79
Patig S (2008) A practical guide to testing the understandability of notations. In: Proceedings of the Fifth Asia-Pacific conference on conceptual modelling - vol 79, APCCM '08. Australian Computer Society, Inc., Darlinghurst, pp 49–58
Purchase HC, Colpoys L, McGill M, Carrington D, Britton C (2001) Uml class diagram syntax: an empirical study of comprehension. In: Proceedings of the 2001 Asia-Pacific symposium on information visualisation - vol 9, APVis '01. Australian Computer Society, Inc., Darlinghurst, pp 113–120
Reijers H, Mendling J (2011) A study into the factors that influence the understandability of business process models. IEEE Trans Syst Man Cybern Part A Syst Humans 41:449–462
Robbins JE, Medvidovic N, Redmiles DF, Rosenblum DS (1998) Integrating architecture description languages with a standard design method. In: Proceedings of the 20th international conference on software engineering, ICSE '98, IEEE Computer Society, pp 209–218
Rozanski N, Woods E (2005) Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley Professional, Boston
Rubinfeld DL (2000) Reference guide on multiple regression, 2nd edn. Federal Judicial Center, Washington
R Development Core Team (2008) R: a language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0. http://www.R-project.org
Sarkar S, Kak A, Rama G (2008) Metrics for measuring the quality of modularization of large-scale object-oriented software. IEEE Trans SSoftw Eng 34:700–720
Sartipi K (2001) A software evaluation model using component association views. In: IWPC, pp 259–268
Sengupta S, Kanjilal A, Bhattacharya S (2011) Measuring complexity of component based architecture: a graph based approach. SIGSOFT Softw Eng Notes 36:1–10
Sharma A, Grover PS, Kumar R (2009) Dependency analysis for component-based software systems. SIGSOFT Softw Eng Notes 34:1–6
Soo LG, Jung-Mo Y (1992) An empirical study on the complexity metrics of petri nets. Microelectron Reliab 32(3):323–329
Stevanetic S, Zdun U (2014a) Exploring the relationships between the understandability of architectural components and graph-based component level metrics. In: Proceedings of the 14th international conference on software quality (QSIC), QSIC 2014. IEEE Computer Society, Dallas
Stevanetic S, Zdun U (2014b) Exploring the relationships between the understandability of components in architectural component models and component level metrics. In: Proceedings of the 18th international conference on evaluation and assessment in software engineering (EASE), EASE 2014. ACM Computer Society, London
Stevanetic S, Zdun U (2015) Software metrics for measuring the understandability of architectural structures – a systematic mapping study. In: EASE 2015 - 19th international conference on evaluation and assessment in software engineering
Stevanetic S, Zdun U (2016) Exploring the understandability of components in architectural component models using component level metrics and participants' experience. In: The 19th international ACM Sigsoft symposium on component-based software engineering (CBSE 2016)
Stevanetic S, Haitzer T, Zdun U (2014) Supporting software evolution by integrating dsl-based architectural abstraction and understandability related metrics. In: Proceedings of the 2014 European conference on software architecture workshops, ECSAW '14. ACM, New York, pp 19:1–19:8
Sun D, Wong K (2005) On evaluating the layout of uml class diagrams for program comprehension. In: Proceedings. 13th international workshop on program comprehension, 2005. IWPC 2005, pp 317–326
Vanderfeesten I, Reijers HA, Mendling RJ, Aalst WM, Cardoso J (2008) On a quest for good process models: the cross-connectivity metric. In: Bellahséne Z, Léonard M (eds) Proceedings of the 20th international conference on Advanced Information Systems Engineering (CAiSE '08). Springer, Berlin, Heidelberg, pp 480–494. https://doi.org/10.1007/978-3-540-69534-9_36
Vanhatalo J, Völzer H, Leymann F (2007) Faster and more focused control-flow analysis for business process models through sese decomposition. In: Proceedings of the 5th international conference on service-oriented computing, ICSOC '07. Springer, Berlin, pp 43–55
van der Aalst WMP, Bisgaard Lassen K (2008) Translating unstructured workflow processes to readable bpel: theory and implementation. Inf Softw Technol 50:131–159
Völter M (2010) Architecture as language. IEEE Softw 27:56–64

Weber B, Zeitelhofer S, Pinggera J, Torres V, Reichert M (2014) How advanced change patterns impact the process of process modeling. In: Bider I, Gaaloul K, Krogstie J, Nurcan S, Proper H, Schmidt R, Soffer P (eds) Enterprise, business-process and information systems modeling vol 175 of Lecture Notes in Business Information Processing. Springer, Berlin, pp 17–32

Zimmermann O, Gschwind T, Küster J, Leymann F, Schuster N (2007) Reusable architectural decision models for enterprise application development. In: Proceedings of the quality of software architectures 3rd international conference on software architectures, components, and applications, QoSA'07, Springer, pp 15–32



Dr. techn. Srdjan Stevanetic is a software architect and research scientist at Siemens A.G., Austria. Before that, he received his PhD at the Software Architecture Research Group, Faculty of Computer Science, University of Vienna, Austria. His main working areas are: empirical software engineering, software architecture research and review, software engineering and design in various fields including big data applications, intelligent autonomous systems (robots, cyber-physical systems), SOA based applications, and NFR engineering. Srdjan has published several articles in peer-reviewed journals, conferences, and workshops.

Prof. Dr. Uwe Zdun is a full professor for software architecture at the Faculty of Computer Science, University of Vienna. Before that, he worked as assistant professor at the Vienna University of Technology and the Vienna University of Economics respectively. He received his doctoral degree from the University of Essen in 2002. His research focuses on software design and architecture, empirical software engineering, distributed systems engineering (service-based, cloud, mobile, IoT, and process-driven systems), software patterns, domain-specific languages, and model-driven development. Uwe has published more than 210 articles in peer-reviewed journals, conferences, book chapters, and workshops, and is co-author of the books "Remoting Patterns – Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware", "Process-Driven SOA – Proven Patterns for Business-IT Alignment", and "Software-Architektur." He has participated in 26 R&D projects. Uwe is editor of the journal Transactions on Pattern Languages of Programming (TPLoP) published by Springer, Associate Editor of the Computing journal published by Springer, and Associate Editor-in-Chief for design and architecture for the IEEE Software magazine.