Supporting Software Evolution by Integrating DSL-based Architectural Abstraction and Understandability Related Metrics

Srdjan Stevanetic, Thomas Haitzer, and Uwe Zdun
Software Architecture Research Group
University of Vienna, Austria
srdjan.stevanetic|thomas.haitzer|uwe.zdun@univie.ac.at

ABSTRACT

Software architecture erosion and architectural drift are well known software evolution problems. While there exist a number of approaches to address these problems, so far in these approaches the understandability of the resulting architectural models (e.g., component models) is seldom studied. However, reduced understandability of the architectural models might lead to problems similar to architecture erosion and architectural drift. To address this problem, we propose to extend our existing DSL-based architecture abstraction approach with empirically evaluated understandability metrics. While the DSL-based architecture abstraction approach enables software architects to keep source code and architecture consistent, the understandability metrics extensions enables them, while working with the DSL, to continuously judge the understandability of the architectural component models they create with the DSL. We studied the applicability of our approach in a case study of an existing open source system.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.8 [**Software Engineering**]: Metrics

General Terms

Experimentation, Measurement, Design

Keywords

DSL, Architectural Abstraction, Architectural Component Views, Software Evolution, Understandability, Software Metrics, Empirical Evaluation

1. INTRODUCTION

Software systems must evolve constantly or they will become obsolete [14]. During the evolution of software systems, software architectures tend to erode as requirements change or new features are implemented [21] (also known as *architectural erosion*). In addition, the intended, documented architecture and the implemented architecture of a system often drift apart during the system's evolution [12] (also known as *architectural drift*).

To address these problems, many approaches have been proposed [18, 19, 7]. For instance, in our previous work [11] we proposed a semi-automatic approach for keeping the architecture and source code consistent throughout the software evolution. In this approach, we used a Domain Specific Language (DSL) that allows architects to specify architectural abstraction specifications. These architecture abstraction specifications enable the architects to define architectural components based on the source code. Based on the architecture abstraction specifications we then automatically generate an architectural component view of the system and its current state. While this approach addresses the consistency of architecture and source code, it does not offer any solutions for preventing the architectural designs in the component models from degrading over time and become less and less understandable. For instance, some architecture design models tend to grow in size over time, as new features are added to the system, until they become at some stage hard to understand.

Clements et al. [5] stated that it is essential that an architecture is documented well in order to communicate it. Reduced understandability hampers the possibility to communicate the architecture well and thus probably leads to further architectural erosion and drift. This is why we consider the understandability of an architecture as essential to the future evolution of a software system.

In this paper we propose to integrate our DSL-based architecture evolution approach with empirically evaluated understandability metrics. We suggest to use understandability metrics for the architectural component view as a whole as well as understandability metrics that focus on single architectural components. This way, while using our architecture abstraction DSL to create component model abstraction, the architect is automatically informed when the understandability of the architecture in the component models that are created through the DSL is reduced during the evolution of the software system and can take measures to improve the architecture's understandability. The metrics we use are empirically evaluated in our previous work with regard to the understandability of either the whole component view or the individual components. A precondition for the application of the metrics (i.e., for the accurate and successful metrics calculations) is an "up-to-date" component view that reflects the source code of the examined system. As this is provided by our previously mentioned architecture abstraction approach, our approach with metrics depends on the integration of both approaches. The main contributions of this paper are the conceptual integration of the two approaches, the integration into our DSL-based tool support, and the derivation of a set of metrics-based guidelines for component model design from our previous empirical studies.

The remainder of this paper is organized as follows: In Section 2 we give an overview of the Architecture Abstraction DSL in its original form. We give an overview of the proposed integrated approach in Section 3. Section 4 describes the details of the given integrated approach. We present a case study in which we have studied the applicability of our approach in Section 5. In Section 6 we discuss relevant related works, and we conclude in Section 7.

2. BACKGROUND: ARCHITECTURE ABSTRACTION DSL

Our previously defined DSL-based approach supports the semi-automated architectural abstraction of architectural component views throughout the software life-cycle. The approach supports the software architect throughout the evolution of a software system by allowing him/her to compare the abstracted model with a previously defined architectural model and to maintain that model in correspondence with the source code over time.

We introduced a DSL that defines architectural abstractions from class models, which can be automatically extracted from the source code, into architectural component views. Once an architectural abstraction specification is defined, it can automatically generate the architectural component view. The workflow of the generation process is shown in Figure 1. First, the extraction of a class model from the source code of the system is pursued. Further analysis uses a class model which decouples the approach from a specific source language. Second, a model transformation is used to generate a UML component view. This model transformation uses the architectural abstraction specification defined in the DSL code (for more details see Section 4.2) and the class model as inputs and generates a UML component view. The model transformation also generates and stores traceability information that links the class models and the architectural component views.

During the software system evolution multiple architectural component views can be generated, one for each version of the system. All those component views can be compared to each other and to component views generated manually by the software architects during early software architecture design stages. This way the approach can help software architects to identify where the implementation differs from the original design or from previous versions.

Beside the already mentioned consistency checking between different versions of software the approach does automatic consistency checking of the different artifacts of the same software version. These checks are based on the automatically generated traceability information that link the

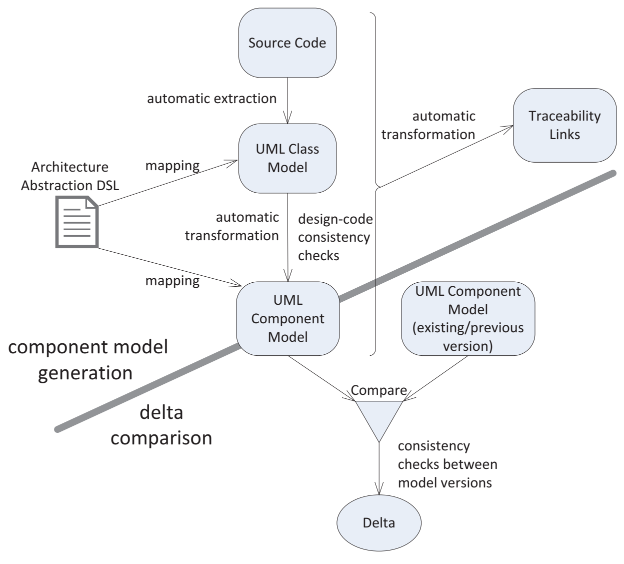

Figure 1: Overview of the architecture evolution approach proposed by Haitzer and Zdun [11]

DSL, the class model, and the component view of the system and check, for instance, for 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 of these components and vice versa: if the source code classes of two components have relations with each other but no connector is defined in the architecture abstraction. The approach also allows architects to keep track of which parts of the source code correspond to which architectural components by utilizing the traceability links that are created and stored during the transformation. The traceability links can then be used for navigating from the architectural model to the source code and vice versa.

In our previous work [11] we performed a number of case studies on open source systems of different sizes and application types that showed that in most of the cases it was possible to create architectural abstractions that are stable during the implementation process and only need to be changed when architectural changes occur (e.g., leading to significant restructuring of the architectural design).

Using this approach, architects can easily maintain an architecture documentation by providing an "up-to-date" architectural component view that reflects the source code. However, the quality (in terms of understandability) of the abstracted and then generated component views is not yet addressed by the approach. For instance, there is no indicator whether the component model is for instance growing too large too be understandable by humans or other similar guidelines.

3. INTEGRATED APPROACH OVERVIEW

The approach that we present in this paper represents an extension of the previously explained approach for supporting semi-automated architectural abstractions of a software system from the source code using a DSL that we call *Architecture Abstraction DSL*. The proposed extension of the approach is related to the integration of software metrics that can support the understandability of architectural component views generated using the previously explained approach. The understandability related software metrics are empirically evaluated in our previous work [24, 25] and can further support the maintainability of the continuously evolved architecture.

Namely, we did a series of studies where we tried to empirically evaluate and prove the usefulness of software metrics in assessing the understandability of architectural component views. The goal was to produce a set of guidelines as best practices for architectural component view design. The metrics that we show are collected at the level of individual components as well as at the level of the whole architecture. They include three simple size metrics related to the number of components, the number of connectors and the total number of elements (summing up the number of components and the number of connectors) in the architecture and four metrics related to individual components the number of classes in a component, the number of incoming dependencies of a component, the number of outgoing dependencies of a component, and the number of internal dependencies of a component.

Regarding the three architecture level size metrics, we showed that middle values of those metrics significantly increase the architectural understandability compared to high or low values [24]. The indicated thresholds/guidelines for using the metrics are roughly predicted and need to be investigated further (they are defined below in Section 4). More precisely we showed that the component diagrams (visual representations of the component views) with very high numbers of elements usually suffer from mixing of several concerns which might lead to ambiguity and less precision. Very low numbers of components, links, and elements are not sufficient to model all relevant concerns of the architecture [24]. The four metrics at the level of individual components are shown to be useful in predicting the effort required to understand an individual component, measured through the time that participants spent on studying a component [25]. They have shown either a statistically significant correlation with the effort required to understand a component or can be used in the prediction models obtained using the multivariate regression analysis, to predict the given effort.

The integration of the given metrics in the workflow of the previously explained approach is shown in Figure 2. In order to more easily distinguish the part related to the integration of the given metrics we marked it red in the figure. Firstly, the metrics calculations are extracted from both the class model and the component view. The obtained metrics values then need to be evaluated with regard to different metrics constraints, i.e., it should be checked if the metrics values satisfy required 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 previous empirical evaluations and also take into account some additional reasonable considerations. Namely, for the architectural level metrics the obtained middle values that increase the architectural understandability can be realized as constraints (thresholds/guidelines are shown in Section 4). For the metrics at the level of individual components we did not examine any specific values/thresholds that can be specified as constraints but the information related to the obtained prediction models and the statistically significant correlations can be useful in providing the relative values that might be used for identifying critical components which require more effort to be understood (see Section 4 for more details). All given constraints and considerations can be further refined with regard to the architects' and developers' specific experience and more specific requirements in the certain domain. 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.

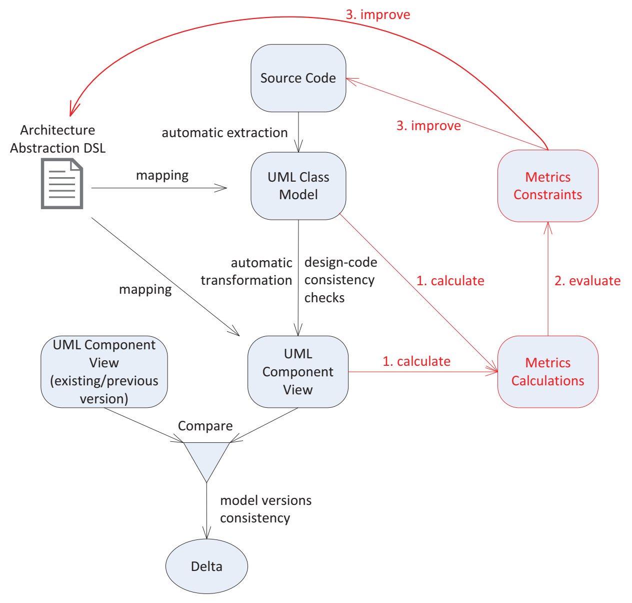

Figure 2: Integration of the understandability related metrics in the DSL-based architecture abstraction approach

4. INTEGRATED APPROACH DETAILS

In this section, we explain the technical details of our approach. In Section 4.1 we discuss the metrics we use in our approach and in Section 4.2 we present the details about the DSL-based architecture abstraction approach and its integration with the given metrics.

4.1 Understandability Related Metrics

Regarding the empirical studies for supporting the understandability of architectural component views we report the results from three studies [1]. The first two studies examine to which extent the software architecture could be conveyed through architectural component views (16 different component diagrams were studied) and they are base on the participants' subjective ratings while the third one examines the relationships between the effort required to understand an individual component, measured through the time that participants spent on studying a component, and some com-

[1]Some more studies were conducted in the meantime and they are currently under review. The results from those studies will be incorporated in our approach afterwards because we would need to explain them in the paper in details and the space limitations do not allow us to do that. For the published studies we can simply refer the reader to the corresponding articles

ponent level metrics that describe component's size, complexity and coupling.

The statistical evaluation of the results from the first two studies shows that metrics such as the number of components, number of connectors, number of elements, and number of symbols used in the diagrams can significantly decrease architectural understandability when they are above and below a certain, roughly predicted threshold. Also, our results indicate that architectural understandability is linearly correlated with the perceived precision and general understandability of the diagrams (please refer to [24] for more details about the terms precision, general understandability, and architectural understandability). The conclusions form these two studies are summarized below:

- Any measures that increase the general understandability and precision of architectural component views directly help to improve the architectural understandability.
- Measures to increase the domain knowledge are helpful to increase the understanding of architectural component views in general.
- From a certain size on (in terms of number of elements), architectural component views get hard to understand in general because of the high cognitive load and human perception limits.
- Middle values of the number of components, links, elements, and symbols in the diagram significantly increase the architectural understandability compared to high or low values. The diagrams with very high numbers of elements usually suffer from mixing of several concerns which might lead to ambiguity and less precision. Very low numbers of components, links, and elements are not sufficient to model all relevant concerns of the architecture. These dependencies might also deserve to be investigated further, especially it would be interesting to indicate the thresholds of maximum (minimum) numbers of components, links, elements, and symbols that should be depicted in one diagram more precisely. So far, we consider the thresholds we found as rough indicators.

From these 2 studies we consider three metrics the number of components, the number of connectors and the total number of elements (summing up the number of components and the number of connectors) in the architecture. As it is mentioned above we observed that the middle values of those metrics significantly increase the architectural understandability. Therefore the corresponding metrics' constraints can be realized (based on the thresholds that are roughly indicated in our previous study [24]). Table 1 summarizes the considered architecture level metrics together with the corresponding constraints. The number of symbols is not considered because it is related to the visual representation of the component views that we do not support at the moment. Also we do not consider the first two items in the above mentioned conclusions because we did not examine the appropriate measures for it. Those items are related to the measures of the precision, the general understandability, and the domain knowledge contained in the component views. Some aspects of these measures are automatically taken into account when the architecture abstraction DSL is specified like for example the names of the components. Informative and coherent names can increase the precision and convey the domain semantics of the system. However, more studies are necessary to define and examine the appropriate measures and the corresponding constraints for these aspects.

Regarding the third study four metrics related to individual components the number of classes in a component, the number of incoming dependencies of a component, the number of outgoing dependencies of a component, and the number of internal dependencies of a component are considered. The results of the analysis show a statistically significant correlation between three of the metrics, number of classes, number of incoming dependencies, and number of internal dependencies, on one side, and the effort required to understand a component, on the other side. In a multivariate regression analysis we obtained 3 reasonably well-fitting models that can be used to estimate the effort required to understand a component [2].

For the metrics at the level of individual components we did not examine any specific values/thresholds that can be specified as constraints. The information related to the obtained correlations and prediction models can be used to provide more relative values (rather than evaluating a design by giving absolute values) that might be used for identifying critical components which require more effort to be understood. Those components can be further simplified and/or reorganized together with other components in the system to satisfy the given understandability requirements. For example, Bouwers et al. found that the components should be balanced in size in order to facilitate the system's analyzability (location of possible failures/bugs in the system) [4]. In our case the similar reasoning can be applied. Balanced values for the components' understandability effort can facilitate the analyzability of the whole system in terms that all components require the same effort to be understood which can facilitate the location of possible bugs/failures in the system (see Section 5 for an illustrative example). Furthermore for the component level metrics the architects/developers can adopt the specific ranges for them based on their concrete experiences and requirements.

The component level metrics together with the prediction models and the identified correlations to the measured understandability effort are shown in Table 2. The Spearman's correlation coefficients are shown. They are widely used for measuring the degree of relationship between two variables and take a value between -1 and +1. A positive correlation is one in which the variables increase (or decrease) together. A negative correlation is one in which one variable increases as the other variable decreases. The coefficient for the number of outgoing dependencies metric is not shown because it is not statistically significant.

4.2 Architecture Abstraction Approach and Metrics Integration

As we mentioned above the given understandability metrics are integrated with the Architecture Abstraction DSL that supports semi-automated architectural abstractions of a software system from a source code. The Architecture Ab-

[2]Please note that the given prediction models do not include the percentage of the correct answers variable which is replaced with 100 % according to the discussion in our previous work [24].

Metic	Description	Metric's constraint
Number of Components (**NCOM**)	Total number of components in the architecture	5 < **NCOM** < 15
Number of Connectors (**NCONN**)	Total number of connectors in the architecture (regardless whether the connector is one-way or two-ways)	3 < **NCONN** ≤ 17
Number of Elements (**NELEM**)	Total number of elements in the architecture (summing up the number of components and the number of connectors)	11 < **NELEM** ≤ 25

Table 1: Architecture level metrics

Metic	Description	Spearman's correlation coefficient
Number of Classes (**NC**)	Total number of classes inside a component	r=0.74
Number of Incoming Dependencies (**NID**)	Total number of dependencies between the classes outside of a component and the classes inside a component that are used by those outside classes	r=0.26
Number of Outgoing Dependencies (**NOD**)	Total number of dependencies between the classes inside a component and the classes outside of a component that are used by those inside classes	-
Number of Internal Dependencies (**NIntD**)	Total number of dependencies between the classes within a component	r=0.66

Prediction Models
Model 1:~4.85+1.52***NC**-0.53***NID**
Model 2:~4.58+1.46***NC**-0.52***NID**+0.12***NOD**
Model 3:~5.32+1.42***NC**-0.58***NID**

Table 2: Component level metrics and the obtained prediction models

straction DSL is developed using Xtext2 that allows specifying architectural components and connectors and their relations to source code using a number of different rules. Those rules can be grouped into 4 different categories:

- **Rules working on source code artefacts:** These rules allow relating different source code artefacts packages, classes, and interfaces to an architectural component. One example for this category is the *Package* rule shown in the example in Figure 3 which selects everything inside a specific package.
- **Rules utilizing relationships between source code artefacts:** These rules allow relating an architectural component to the source code artefacts that have specific relationships to the given source code artefact (sub- and super-type relations for classes and interfaces, interface realizations, and other dependencies). Figure 3 shows the *ChildOf* rule as an example for this category. This rule matches all classes that extend the referenced class.
- **Rules operating on the names of the source code artefacts:** These rules related an architectural components to source code artefacts based on regular expressions of the names of those artefacts. An example of the rule from this group is the rule Class(”.*No.*”) (see Figure 3). This rule matches a regular expression over all class names.
- **Rules composition:** More complex rules definitions are supported through the implementation of the three set operations union (*or*), intersect (*and*), and difference (*and not*) which are all used in Figure 3.

The given set of rules is incrementally refined by studying five open source projects (see [11] for more details). During the incremental refinement of our DSL design, we started with scenarios from these projects and extended the set of scenarios step-by-step to cover all changes observed in multiple versions of those five projects. First, we automatically generated a UML class model from the source code using our parser and tried to gain an initial understanding of the program. In order to ease this task we imported the source code in an Eclipse IDE. After an initial study of the source code, we created a first, incomplete architecture abstraction specification. The time that we needed to create this initial specification heavily depended on the size and the previously existing architectural knowledge about the studied cases. Then we utilized the consistency checks to further improve the abstraction specification by removing the reported inconsistencies step-by-step. The inconsistency at this point usually were source code elements that had not been considered in the abstraction specification. When we were satisfied with the resulting architecture abstraction specification, we updated the source model to an existed newer version. After that we checked the architecture specification and the new source model for inconsistencies. Any reported inconsistencies were fixed before we continued with the next version of the program.

```
Component Demo
    consists of
    { // brackets for overriding operator precedence
    /* Structure based: include
     * everything inside this package*/
        Package(org.example)
        and not /* compositon - set difference*/{
            /* Name-based: classes with
             * name containing No*/
            Class(".*No.*")
        } or /* composition - union */ {

    ChildOf(org.example2.AbstractSuperTypeClass)
        }
    } and/* composition - intersection */
    InstanceOf(org.example.IExampleInterface)
```

Figure 3: Architecture Abstraction DSL example - Demo component

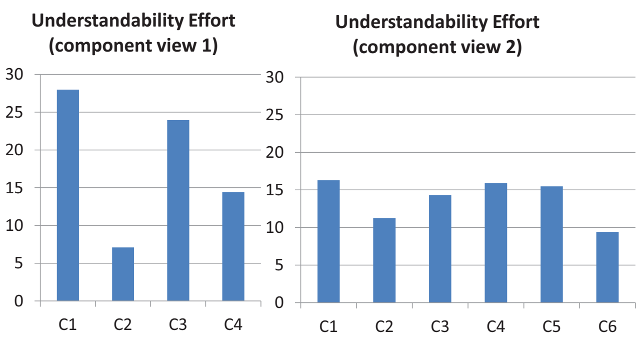

Figure 4: Understandability effort for both component views

Our consistency checking algorithm mentioned in Section 2 supports the evolution of the software system in such a way that it enables consistency checking between different versions of software and also between different artefacts of the same software version (for example between the component view and the corresponding class view). The integrated empirically evaluated metrics provide an additional consistency checking possibility. Namely, according to the discussion above the integrated metrics can provide a valuable support in assessing the understandability of architectural component views which plays a key role in managing and maintaining the overall system. Different versions of the software can be compared using the given metrics set that can be used to argue about the understandability level of both the architectures and the individual components contained in them. Based on the obtained values critical points can be recognized, for example the components that have significantly increased the effort to be understood can be identified. Also different architecture abstractions can be compared in order to generate the one with the reasonable understandability level. The integrated metrics benefit from the architecture abstraction tool in the way that the later provides an "up-to-date" architectural component view 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, the architects/developers can gradually improve the architecture by making the changes in the source code or in the architecture abstraction DSL and judge the understandability of the architecture created with the DSL. The metrics calculations are integrated using the Xtext validation framework which triggers their execution/recalculation whenever the source code or the architecture component view is changed. The corresponding warnings are reported whenever the metrics values violate the respective set of metrics constraints.

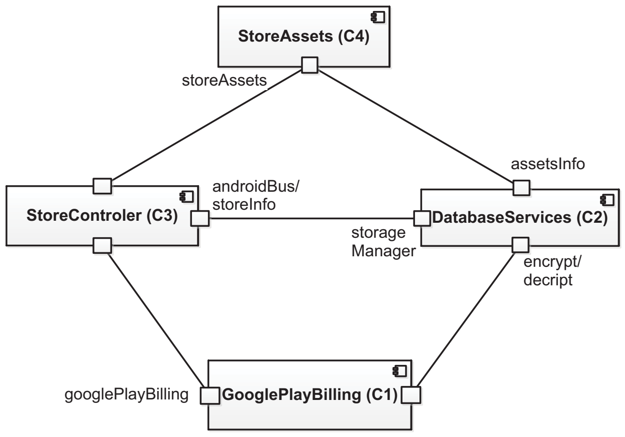

Figure 5: Soomla Android store component view 1

5. CASE STUDY

In this section we present a small case study that illustrates how the previously explained approach can be used to localize possible undesirable effects in the design, in this case the observed fluctuations in the understandability effort of architectural components. The studied system is the Soomla Android store Version 2.0, an open source framework for supporting virtual economy in mobile games [3]. Namely, we show two architectural component views of the system that differ in the number of components and the number of classes that the components contain. In both cases we calculate the understandability effort required to understand each component based on the provided prediction models. In the first case the understandability effort is unevenly distributed over the components, i.e., some components require very low while some others require very high effort to be understood (see Figure 4). After studying the first component view the new component view is generated that better distributes the understandability effort over the components. Thanks to the architecture abstraction tool all source code classes are mapped to their respective components which is a precondition for the accurate and successful metrics calculations. Furthermore the second component view is easily created from an architectural abstraction of the first one by simply relocating the classes in the DSL code from one component to the other. This step, of course, requires human expertize and manual effort. However, please note that the migration to the new view can be done incrementally, by performing small changes in the DSL and observing the change of the metrics with each change in the DSL. Please note also that in general a large, inherently complex system will have lower understandability because the identified metrics (i.e. NCOM etc.) will be higher, than a small, simple system, regardless of the quality of the architecture abstractions used. In that case the aim of the approach is to adapt the inherently high complexity to the extent that is acceptable using the explained incremental changes.

Figure 5 shows the first component view obtained by studying the given software system. The visualization of both component views is separately created in the form of a UML component diagram. Figure 6 shows the second component view created to support better distributed understandability effort between the components in order to facilitate their analyzability (see Section 4 for more details). The understandability efforts are shown in Figure 4.

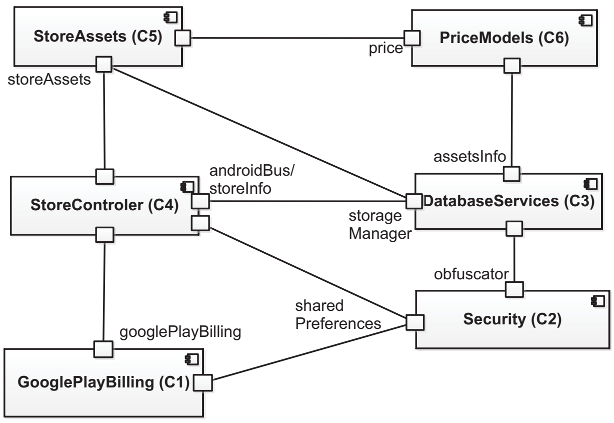

Figure 6: Soomla Android store component view 2

From Figure 4 we see that the Components C1 and C3 of the first component view require a pretty high effort to be understood while the Component C2 requires much less effort. In the second component view the components require more or less balanced effort to be understood, and it is lower than the effort required for the Components C1 and C3 in the first design. This small case illustrates how the given metrics provide a useful feedback in the explained context.

[3]see: http://project.soom.la/

6. RELATED WORK

In this section we compare our approach to other approaches that focus on architecture evolution as well as to other approaches that utilize metrics in a similar way.

Many different software metrics for measuring the system's architecture, components as its constituting parts, and structures similar to architectural component views, such as other higher-level software structures (packages, graph-based structures) have been proposed. Metrics related to components and the corresponding architectures [13, 23, 22] measure size, coupling, cohesion, and dependencies of individual components but also the complexity of the whole architecture when all the components and their interactions are taken into account. Different authors have proposed different package level metrics that measure their size, coupling, stability, and cohesion [17, 10]. Graph-based metrics measure different interactions between the nodes in the graph [3, 15]. Some of the graph-based metrics have been shown to be useful in measuring large scale software systems in the sense that those systems share some properties that are common for complex networks across many fields of science [15]. All the mentioned metrics can be applied or can be more-less easily adapted to be applicable for the component views. However, none of the metrics is empirical evaluated regarding understandability of architectural components or architectural component views so far. In the software architecture literature we find only a very few studies that provide empirical evidence regarding the architectural understandability or the measurement of architectural understandability (see e.g. [9, 8]). Our empirical studies explained before try to provide more evidence in that context. Furthermore the realization of the empirically collected evidence by developing the corresponding tools and the integration of those tools with the existing tools has a great value and can improve the quality of the software systems to a great extent. Our previously explained integrated approach provides one step in that direction.

A number of approaches focuses on the automatic creation of source code abstractions using automatic clustering. Different clustering approaches and clustering measures are reviewed and compared by Maqbool and Babri [16]. They define a number of groups of clustering algorithms and compare the performance of the different groups for different open source software projects. While Maqbool and Babri conclude which approach works best for each of the applications, they do not draw any conclusions regarding the overall effort necessary to correct the automatic clustering. In contrast to all these approaches our approach is semi-automatic, enables the checking of design constraints during the abstraction process, and provides traceability between source code and models and focuses on the evolution of the architecture rather then the recovery of architecture.

Abreu et al. introduce a reengineering approach using cluster analysis [1]. It uses six different affinity schemes and seven clustering methods to produce a series of clustering proposals to verify which one produces the best results. While this approach focuses on architecture recovery, in our approach we focus mainly on architecture evolution and do provide only semi-automatic support for architecture recovery by trying to make it very comfortable for the architect to define architecture abstraction specifications .

Egyed [7] describes an approach for model abstraction by using existing traceability information and abstraction rules. However, 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 uses architectural abstraction specifications that are harder to reuse for other models but easier to define and allow the definition of architectural abstraction specifications on different levels of abstraction.

Another approach for mapping source code models to high-level models is introduced by Murphy et al. [20]. They use software reflexion models which they compute from a mapping between source model and high-level model. However while their approach is similar, it requires a substantial amount of effort, since it requires to define both: the high-level model and the mapping, while our approach requires source code and architectural abstraction specification and the architecture abstraction is generated automatically.

Mens et al. [18] propose intentional source code views that allow grouping of source code by concerns. These views are defined in a logic programming language. Their approach provides generic source views on a low abstraction level while we focus on the architectural aspects and provide an easy way to define our domain specific views.

An approach that focuses on architecture evolution is proposed by Barnes et al. [2]. Their approach is aimed at planing and reasoning about architecture evolution. They support the modeling of different evolution paths and allow reasoning about these different paths. Cuesta et al. [6] propose an approach called AKdES that extends the approach by Barnes et al. by considering evolution as an important aspect of a systems architecture and thus propose the documentation of architecture evolution using architectural knowledge. While these approaches are focused on planing and reasoning about an architecture evolution and they do not consider how the architecture documentation and the source code can be kept synchronized during evolution, our approach is aimed at supporting the architect during the evolution process by supporting the architecture during the evolution in order to evolve source code an architecture documentation in a synchronized fashion.

7. CONCLUSIONS AND FUTURE WORK

In this paper we presented an approach that uses empirically evaluated understandability metrics to support the software architect during architecture documentation and evolution. It is built on previous work on architectural component views that are generated from architecture abstraction specifications. We automatically calculate a number of different metrics whenever the architecture abstraction specification or the source code are updated. If the metrics exceed defined thresholds the prototype notifies the software architect of the potential understandability problem who should revise the architecture abstraction specification and the source code to improve the understandability of the architectural component view. The improved understandability then eases the future evolution of the systems as it reduces the risks of misunderstandings and thus the risk of changes that affect the quality of the architecture in a negative way. Our main contributions lie in the integration of the two approaches and proposing a set of metrics-based guidelines for component model design that are derived from our previous empirical studies. A limitation of our approach is that we currently consider only understandability metrics as a measure for quality. In our future work we plan to integrate other quality metrics that can be used to prevent architecture erosion and drift.

Acknowledgement

This work was supported by the Austrian Science Fund (FWF), Project: P24345-N23.

8. REFERENCES

[1] F. B. e. Abreu, G. Pereira, and P. Sousa. A coupling-guided cluster analysis approach to reengineer the modularity of object-oriented systems. In *Proceedings of the Conference on Software Maintenance and Reengineering*, CSMR '00, pages 13–, Washington, DC, USA, 2000. IEEE Computer Society.

[2] J. M. Barnes, D. Garlan, and B. R. Schmerl. Evolution styles: foundations and models for software architecture evolution. *Software and System Modeling*, 13(2):649–678, 2014.

[3] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *ICSE'12*, pages 419–429, 2012.

[4] E. Bouwers, J. P. Correia, A. Deursen, and J. Visser. Quantifying the Analyzability of Software Architectures. In *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, pages 83–92. IEEE, June 2011.

[5] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[6] C. E. Cuesta, E. Navarro, D. E. Perry, and C. Roda. Evolution styles: using architectural knowledge as an evolution driver. *Journal of Software: Evolution and Process*, 25(9):957–980, 2013.

[7] A. Egyed. Consistent adaptation and evolution of class diagrams during refinement. In *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, ETAPS 2004 Barcelona, Spain*, volume 2984 of *Lecture Notes in Computer Science*, pages 37–53. Springer, 2004.

[8] M. O. Elish. Exploring the relationships between design metrics and package understandability: A case study. In *ICPC*, pages 144–147. IEEE Computer Society, 2010.

[9] V. Gupta and J. K. Chhabra. Package coupling measurement in object-oriented software. *J. Comput. Sci. Technol.*, 24(2):273–283, Mar. 2009.

[10] V. Gupta and J. K. Chhabra. Package level cohesion measurement in object-oriented software. *J. Braz. Comp. Soc.*, 18(3):251–266, 2012.

[11] T. Haitzer and U. Zdun. Semi-automated architectural abstraction specifications for supporting software evolution. *Science of Computer Programming*, 90, Part B(0):135 – 160, 2014. Special Issue on Component-Based Software Engineering and Software Architecture.

[12] A. Jansen, J. van der Ven, P. Avgeriou, and D. K. Hammer. Tool support for architectural decisions. In *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture*, WICSA '07, pages 4–, Washington, DC, USA, 2007. IEEE Computer Society.

[13] A. Kanjilal, S. Sengupta, and S. Bhattacharya. CAG: A Component Architecture Graph. In *TENCON, IEEE Region 10 International Conference*, 2008.

[14] M. M. Lehman. Uncertainty in computer application and its control through the engineering of software. *Journal of Software Maintenance*, 1(1):3–27, Sept. 1989.

[15] Y. Ma, K. He, D. Du, J. Liu, and Y. Yan. 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, pages 189–, Washington, DC, USA, 2006. IEEE Computer Society.

[16] O. Maqbool and H. Babri. Hierarchical clustering for software architecture recovery. *IEEE Trans. Softw. Eng.*, 33:759–780, 2007.

[17] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[18] K. Mens, T. Mens, and M. Wermelinger. Maintaining software through intentional source-code views. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, SEKE '02, pages 289–296, New York, NY, USA, 2002. ACM.

[19] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between source and high-level models. *SIGSOFT Softw. Eng. Notes*, 20:18–28, 1995.

[20] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '95, pages 18–28, New York, NY, USA, 1995. ACM.

[21] D. L. Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[22] K. Sartipi. A software evaluation model using component association views. In *IWPC*, pages 259–268, 2001.

[23] A. Sharma, P. S. Grover, and R. Kumar. Dependency analysis for component-based software systems. *SIGSOFT Softw. Eng. Notes*, 34(4):1–6, July 2009.

[24] S. Stevanetic, M. A. Javed, and U. Zdun. Empirical evaluation of the understandability of architectural component diagrams. In *Companion Proceedings of the 11th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, WICSA 2014, Sydney, Australia, 2014. IEEE Computer Society.

[25] S. Stevanetic and U. Zdun. 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, London, UK, 2014. ACM Computer Society.