Towards a Pattern Language for Construction and Maintenance of Software Architecture Traceability Links

MUHAMMAD ATIF JAVED, SRDJAN STEVANETIC, UWE ZDUN
University of Vienna, Faculty of Computer Science, Software Architecture Research Group, Vienna, Austria

The documentation of software architecture traceability links is the foundation for many important architecture management activities, such as verification and validation, reuse evaluation and impact analysis. In practice, the construction and maintenance of traceability links is mostly manual, which is labor-intensive and error prone. Although the costs of manual traceability in terms of the time, effort and money required can be mitigated by automated construction, the completeness and correctness of traceability links tends to be negatively affected by automation in their creation and maintenance. This paper presents a pattern language for construction and maintenance of software architecture traceability links to requirements and source code. As a foundation for deriving the pattern language, we have performed systematic literature reviews, investigations of traceability links for multiple open-source software systems, and empirical studies. In particular, we studied the nature of the software architecture traceability phenomenon and its driving factors and impacts, as well as the methods that provide the means to control software architecture traceability. The derived pattern language provides support for addressing multiple decision categories for construction and maintenance of software architecture traceability links. To illustrate the patterns, their application is shown in the context of constructing and maintaining traceability links for an open source framework for mobile games.

CCS Concepts: •**Software and its engineering → Software architectures; Documentation;**
Additional Key Words and Phrases: Software Architecture, Traceability Patterns, Grand Challenge 2 (Cost-Effective) and 4 (Trusted) of Traceability.

1. INTRODUCTION

The IEEE standard glossary of software engineering terminology defines traceability as “the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another” [IEEE 1990]. The construction and maintenance of traceability links is mostly manual in practice, which is perceived as costly in terms of the time, effort and money expended [Gotel et al. 2012]. Therefore, much of the current research in software traceability aims at simplification of traceability construction and maintenance by reducing the human effort required. The idea of semi-automatic traceability is to determine where manual intervention is avoidable, reduce the required human involvement, and provide better guidance and tool support. In this

This work is supported by the Austrian Science Fund (FWF) under project P24345-N23.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2015 ACM. 1544-3558/2015/05-ART1 $15.00
DOI: 0000001.0000001

context, model-driven support for eliciting and formalizing [Navarro and Cuesta 2008; Tran et al. 2011] as well as capturing [Haitzer and Zdun 2012] software architecture traceability links can be achieved.

The constructed traceability links need to be maintained (continuously or on-demand) as a project evolves so that up-to-date traceability links would be available when needed. The continuous maintenance of traceability links can be triggered by changes to any of the software artefacts (e.g., architecture components) that, in turn, can be triggered by changes to artefacts within a traceability chain (e.g., underlying requirements and the code classes that implement the component). Semi-automatic support for such event-based maintenance can also be achieved [Hammad et al. 2011; Buchgeher and Weinreich 2011; Mäder and Gotel 2012]. However, the continuous maintenance of traceability links might not be a feasible solution in case of a substantial evolution of a software system, such as a new major version in large real-world project, because the time and effort required might be too high. In this particular case, on-demand reuse and evolution adaptation of traceability links could be performed in the context of different versions of a software project.

Automated information retrieval and machine learning techniques, although helpful to a certain extent, do not completely prevent from insufficient understanding and/or misunderstanding of traceability links, resulting in the unconscious violations during the construction of traceability links. At present, the automated approaches and tools can achieve a completeness of 90% at correctness levels of 5–30% [Antoniol et al. 2002; Chen and Grundy 2011; Hayes et al. 2006; Lucia et al. 2007; Oliveto et al. ; Zou et al. 2010]. Note that the completeness (recall) and correctness (precision) are well-known traceability measures, where recall is defined as the percentage of correct links that are retrieved and precision is defined as the percentage of retrieved links that are actually correct [Baeza-Yates and Ribeiro-Neto 1999; Harman 1992]. Further research has shown that the problems with automated traceability cannot be completely eliminated after validation by a human analyst [Cuddeback et al. 2010; Dekhtyar et al. 2011; Niu et al. 2013].

The architectural level is well suited for construction and maintenance of traceability links, as the software architecture allows (early) reasoning on the quality attributes of the system [Clements et al. 2002] and the software architecture not only describes the high-level structure and behaviour of the system, but also incorporates principles and decisions that determine the system's development and its management [Bengtsson et al. 2004]. To measure the quality attributes of the system, different software measures (metrics) are used. Those metrics take into account the information contained in the architecture and other system artefacts as well as in the corresponding traceability links (see below for details). Therefore, the way in which traceability links are constructed can influence the quality attributes of the system. This paper presents a pattern language for construction and maintenance of software architecture traceability links to requirements and source code. The pattern language is mined from the following sources:

(1) Two systematic literature reviews on (1) software architecture traceability approaches, tools [Javed and Zdun 2014] and (2) architecture level software metrics [Stevanetic and Zdun 2015] have been performed. The results reveal that the research in software architecture traceability has been primarily directed towards new approaches and tools; however, less research focuses on analysing and quantifying the points at which the potential violations in automated traceability would be resolved [Javed and Zdun 2014]. Our systematic mapping study on software metrics for architectural structures provides an overview of the metrics for measuring different quality aspects, such as size, complexity, coupling, cohesion, and modularization of those structures together with the information on how those metrics are used to evaluate external quality attributes like maintainability or understandability [Stevanetic and Zdun 2015].

(2) For identification of potential architectural violation points and their resolutions as well as for the purpose of conducting multiple empirical studies, traceability links for multiple open-source software systems have been constructed. The construction of traceability links was a very time-consuming task

that was completed over the course of three months by reviewing the available literature related to the software systems and then we manually hunting for the features in the source code.

(3) Finally, a number of empirical investigations have been conducted. We have studied human analyst performance in software architecture traceability as well as which software metrics can be used to predict external software qualities with a special focus on understandability [Javed and Zdun a; b; c; Javed et al. 2015; Stevanetic et al. 2014; Stevanetic and Zdun 2014b].

The remainder of this paper is structured as follows: Section 2 introduces a motivating example for traceability of software architecture to requirements and source code. Section 3 describes the pattern language for construction and maintenance of software architecture traceability links. We discuss the related work in Section 4, and conclude in Section 5.

2. MOTIVATING EXAMPLE

The Soomla Android store[1] allows mobile game developers to easier implement virtual currencies (tokens, coins, gems, etc.), virtual goods and in-app purchases. The high-level architecture design of the Soomla Android store Version 2.0 comprises of five components, named as *StoreAssets*, *StoreController*, *DatabaseServices*, *GooglePlayBilling*, and *Security*. In addition, two external components can be modeled: GooglePlayServer, the REST Web Services running at Google, and SQLLiteDatabase, the used database accessed over JDBC. Figure 1 illustrates the traceability links of a component in the Soomla Android store architecture to its underlying requirements and the code classes that implement the component. The use of traceability links is considered critical for rigorous software development. For example, in order to find out whether the software architecture implements all of the specified software requirements, all aspects of the architecture need to be traceable to software requirements. To enable further analysis and make dependencies more explicit, the artefacts from the other activities of the development process, such as the implementation, would also need to be linked to the established specifications.

In our previous work, we have shown that traceability links play an important role in better reasoning and understanding of an architecture using Soomla as an object of study [Javed and Zdun b]. For the *StoreController* component highlighted in the figure, for example, these links help developers and maintainers to identify changes by determining the artefacts that are affected by change and thus estimating the effort for applying a particular change [Javed and Zdun c]. The calculation procedure for reuse evaluation, with a focus on software architecture traceability links, is also carried out in a similar manner to the calculation of the evolution analysis. Traceability links also support the verification and validation of software systems. In this context, they provide the means to check whether the *StoreController* component is complete and consistent with the requirements and the source code. Note that the U.S. federal aviation administration (FAA) and capability maturity model integration (CMMI) require similar traceability practices [Radio Technical Commission for Aeronautics RTCA 1992; McClure 2011].

[1]Soomla is an open source framework that provides a software development kit for implementing every day virtual economy operations in mobile games; see http://soom.la/.

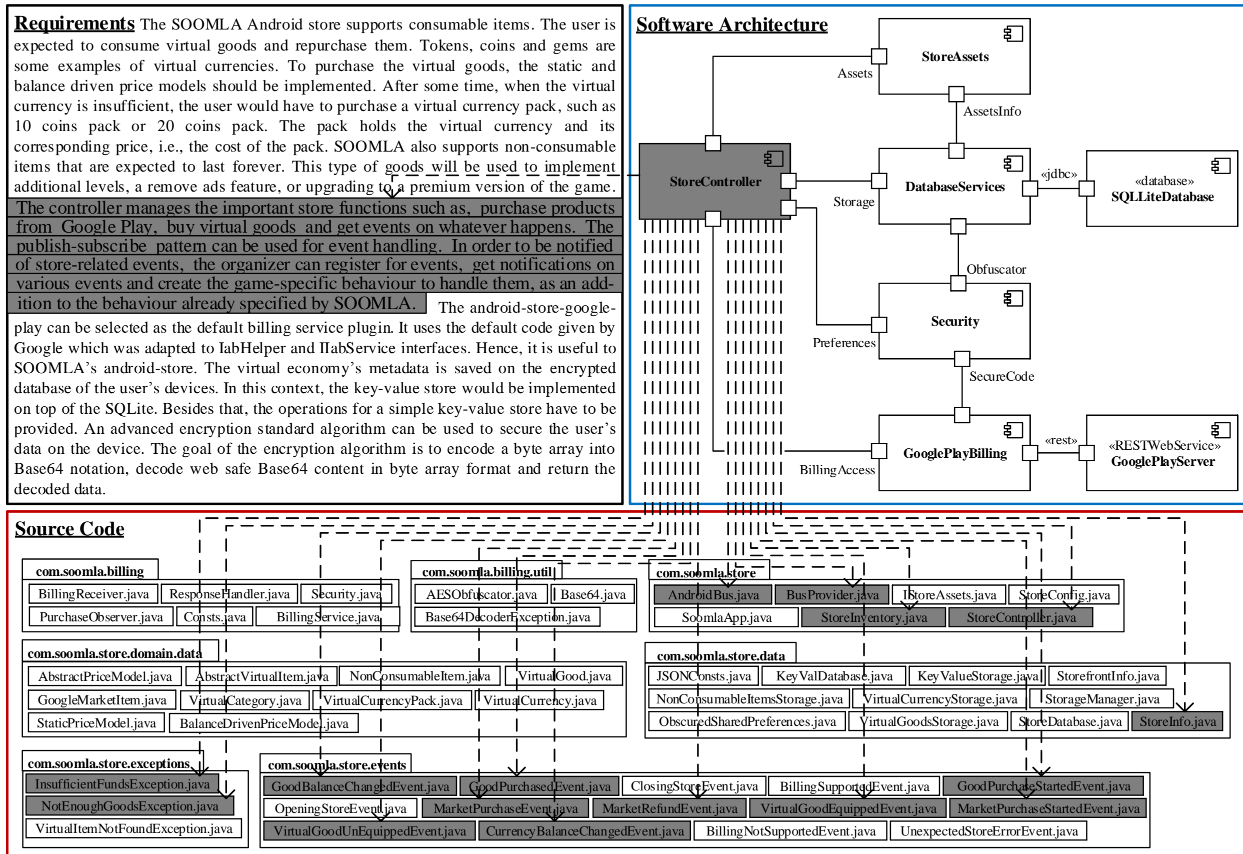

Fig. 1: Traceability Links of a Component in the Soomla Android Store Architecture to its Underlying Requirements and the Code Classes that Implement the Component

3. PATTERN LANGUAGE FOR CONSTRUCTION AND MAINTENANCE OF SOFTWARE ARCHITECTURE TRACEABILITY LINKS

3.1 Pattern Language Overview

In this section, we describe our pattern language by providing a short summary for each pattern and providing a pattern language map. The identified pattern language consists of five patterns:

—Initial traceability construction resolves the problem of establishing clear and straightforward links as a first step towards cost-effective and trusted traceability construction. The link construction not only focusses on textual similarity, but also on the adaptation and reorganization of principle classes based on their relationships, in order to achieve conformance with the architectural components and their interconnections.

—Traceability completion resolves the construction of an entire set of traceability links utilizing an initial set of traceability links. Accordingly, this process focuses on classification scores with an explicit consideration of the architectural conformance.

—Continuous traceability maintenance helps in maintaining traceability links in case of small evolutionary changes captured as change events. The evolutionary changes that require traceability update are recorded and used to find a match with performed maintenance activities.

—On-demand traceability maintenance is used to perform an overall update of previously established traceability links to the new version of a software project. It focuses on component-to-component features for identification and prioritization of previous traceability links constructed using the traceability completion pattern, which are then used to perform reuse and adaptation of traceability links based on the matches and mismatches, respectively.

—Traceability quality checks resolves the problem of reduced external system qualities affected by the way in which the component architecture is partitioned and the traceability links are constructed. The pattern aims at improving the quality of the component architecture partitioning so that desired external system qualities are improved. External system qualities are estimated using software metrics that can be calculated from the system itself taking into account the traceability information. Estimated qualities and metrics are used as a guide to step by step improve the quality of the component model.

Figure 2 shows a pattern language overview diagram. The relationships among the elements of the diagram are represented by the labelled arrows.

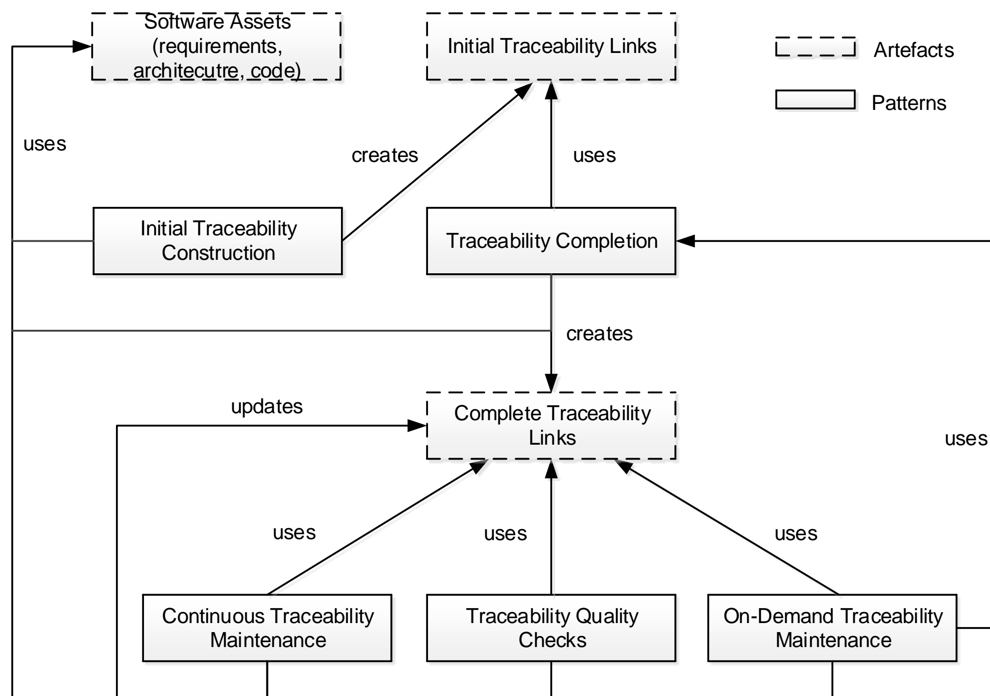

Fig. 2: An Overview of the Pattern Language

3.2 Pattern: Initial Traceability Construction

You want to construct an initial set of traceability links between software architecture and other artefacts produced in the requirements and implementation activities of the development process.

How can you identify an initial set of traceability links for each architectural component in order to provide a basis for the traceability construction process?

The inability to achieve an initial set of correct traceability links leads towards reduced correctness and proliferation of links in automated traceability approaches and tools. Hints for constructing the initial set of traceability links can be detected for instance through the textual information, such as using the same or similar names. However, the violations of architectural conformance (i.e., absences and divergences) need to be identified and resolved.

An initial set of traceability links is established with a focus on constructing the clear and straightforward links as a first step towards cost-effective and trusted traceability construction. The initial traceability construction pattern not only focusses on the textual similarity, but also on the adaptation and reorganization of main classes based on their relationships, in order to achieve conformance with the architectural components and their interconnections.

The functionality required and provided by a component is often specified in the software architecture models and source code classes as interface interactions. However, there is also the possibility that the interfaces are not modelled and/or unavailable in the software implementation. If interfaces are available in the software implementation, the probabilities can be propagated across *implements*-relationships and the interface calls made through the instance references. The classes that implement an interface define their own behaviour within the component (i.e., a provided interface) whereas the classes calling an interface indicates the requiring connected component (i.e., a required interface). If interfaces are not available in the project implementation, the classes with 2-3 times higher relationships than average should be taken into consideration first. In this context, the software architecture needs to be preprocessed to capture constituent elements and their interconnections; whereas the source code is preprocessed to extract the dependency, association, generalization and realization relationships. Besides that, indicator terms in the requirements, software architecture and source code classes are extracted. The INITIAL TRACEABILITY CONSTRUCTION pattern not only focuses on "textual similarity", but also on "adaptation and reorganization of main classes based on their relationships" in order to achieve "conformance with the architectural components and their interconnections". In this context, the possible causes of non-conformance (i.e., absences and divergences) need to be detected and resolved. An absence is the violation that a relation is described in the architectural model, but not reflected in the source code. A divergence is the violation that a relation is not modelled in the architectural model, but exists in the source code. In the context of machine learning, a training dataset can be developed to support the identification and analysis of the initial traceability links. An overview of the INITIAL TRACEABILITY CONSTRUCTION pattern is presented in Figure 3.

We have conducted a controlled experiment on human analyst performance for different kinds of traceability links to find out whether an initial set of correct links for each architectural component is better than an almost complete traceability links with a reduced correctness [Javed et al. 2015]. The participants with the initial set of correct links reveal a focus on a traceability-based assessment process, which was mainly driven by exploring the imports and source code packages of the main classes. However, the participants provided with nearly complete but highly imprecise links generated using an information retrieval based tool focus on finding out the correct traceability links. It was hard for them to further explore the remaining set of relevant links. The evaluation of the experiment showed that the highest completeness and correctness of elements can be achieved using an initial set of correct links compared to the nearly complete but highly imprecise links.

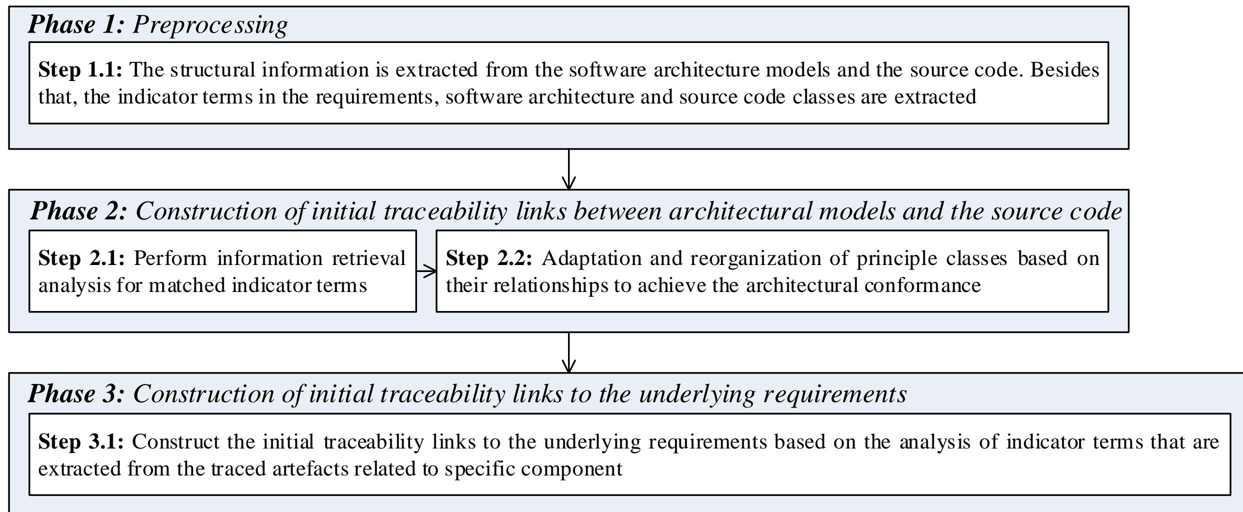

Fig. 3: An overview of the INITIAL TRACEABILITY CONSTRUCTION pattern

Let us consider the example of the Soomla Android store Version 2.0. The construction of an initial set of traceability links between the Soomla Android store architecture and its underlying requirements and the implementation classes can be reflected in three constituent steps:

Step 1: The software architecture is pre-processed to capture the *StoreController*, *GooglePlayBilling*, *Security*, *DatabaseServices* and *StoreAssets* components, and their interconnections; whereas the analysis of relationships between source code classes leads to the selection of nine classes with ten or more relationships: *StoreController*, *StoreInfo*, *BillingService*, *Security*, *AESObfuscator*, *VirtualGoodsStorage*, *StorageManager*, *VirtualCurrencyStorage* and *VirtualGood*. Besides that, the indicator terms from the requirements, software architecture and source code classes are extracted.

Step 2: The selected classes are mapped to the architectural components in a systematic manner to achieve conformance with the architectural components and their interconnections as shown in Figure 4. The information retrieval analysis indicates that the *StoreController*, *BillingService* and *Security* classes can be mapped to the *StoreController*, *GooglePlayBilling* and *Security* components, respectively. The analysis of relationship paths further confirms that the *StoreController* class cannot be mapped to any of the components easily. In this context, the class with highest relationships is mapped to the component connected with all the extracted components. It is detected that the *BillingService* class has relationships with the *StoreController* and *Security* classes, which confirms the mapping of this particular class to the *GooglePlayBilling* component. The *AESObfuscator* class has relationships with the *Security*, *StoreInfo*, *StorageManager*, *VirtualGoodsStorage* and *VirtualCurrencyStorage* classes. As the relationships between classes reflect the architectural components and their interconnections, the *StoreInfo* class is mapped to the *StoreController* component, while the *Security* and *AESObfuscator* classes are mapped to the *Security* component. Furthermore, the *StorageManager*, *VirtualGoodsStorage* and *VirtualCurrencyStorage* classes are mapped to the *DatabaseServices* component due to the mutual term ‘Storage’ and their relationships. Similarly, the *VirtualGood* class is mapped to the *StoreAssets* component.

Step 3: The links to underlying requirement should be constructed based on the analysis of indicator terms in the traced artefacts related to specific component. For example, the ‘Google’, ‘Play’, ‘Billing’ and ‘Service’ terms – extracted from the *GooglePlayBilling* component and *BillingService* class – provide the

means for identification of the following corresponding text: “the android-store-google-play can be selected as the default billing service plugin.”

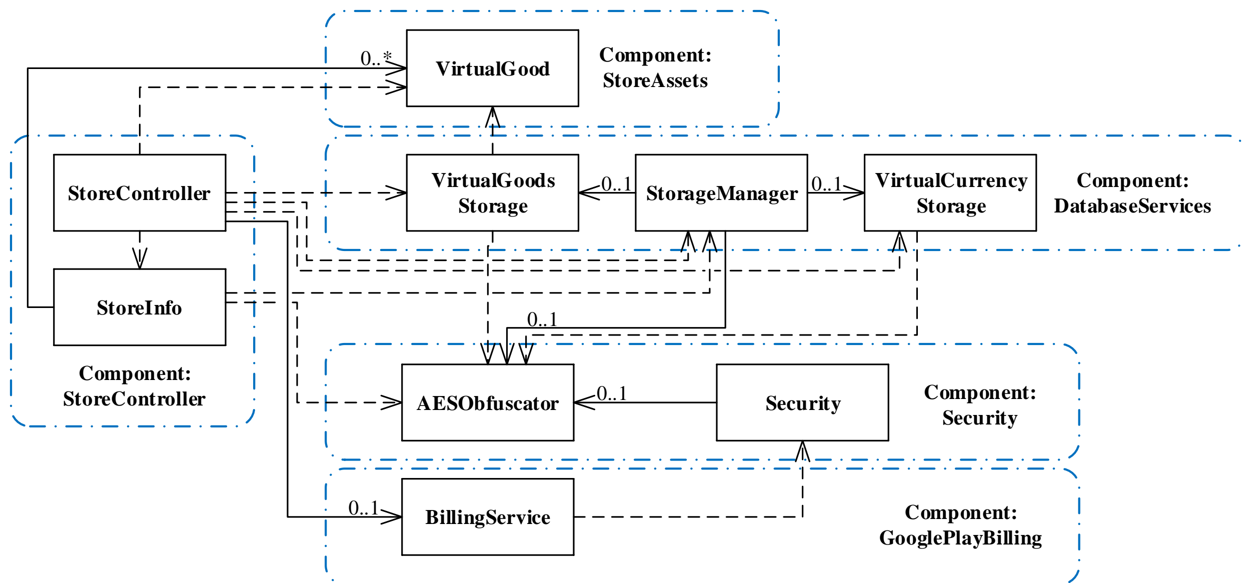

Fig. 4: Alignment of source code classes in terms of the architectural conformance

Multiple traceability approaches and tools require initial traceability links from the developers to produce better results [Nguyen et al. 2005; ?; ?]. In this context, the machine learning techniques require training dataset from the developers, and therefore tends to provide better results than information retrieval techniques. The inability to achieve the initial set of correct traceability links lead towards reduced correctness and proliferation of links in automated traceability approaches and tools.

3.3 Pattern: Traceability completion

You want to construct the entire traceability links between a software architecture and other artefacts produced in the requirements and implementation activities of the development process.

♦♦♦

How can you transform a set of initial traceability links into an entire set of traceability links by determining a preferred candidate solution?

After constructing the initial traceability links, the hints for entire traceability links can be detected through the classification scores and architectural violations. Therefore, the main reason of first constructing the principal traceability links is that the probabilities for the least dependent artefacts could be later changed.

♦♦♦

The initial set of traceability links can be used as an active countermeasure to arbitrarily making traceability decisions and to maintain and preserve the trust in further traceability construction. In particular, it helps in assignment of classification scores with an explicit consideration of the architectural conformance.

The construction of traceability links is not an essential part of project planning and management in general, and therefore often tackled in isolation when needed in projects, rather than built into the software development lifecycle. The exemptions are model-driven development and formal development processes in which the transformations provide essential support for traceability. The initial set of identified links provide a basis for analysing and evaluating the likelihood of unconstructured traceability links. A main reason of first constructing the principal traceability links is that the probabilities for the least dependent artefacts could be later changed.

In the traceability completion process, the classes that tend to be linked to the same components, for example, related by *«extends»* relationships with the traced classes can be first handled. In the subsequent step, the classes with similar indicator terms can be taken into consideration. Finally, the undetected classes are mapped based on higher relationships with a class(es) in specific component. Most importantly, the "classification scores" and "architectural violations" have to be considered for mapping to the corresponding components. As with the INITIAL TRACEABILITY CONSTRUCTION pattern, the indicator terms in the traced artefacts related to specific component are used to identify the underlying requirements. An overview of the TRACEABILITY COMPLETION pattern is presented in Figure 5.

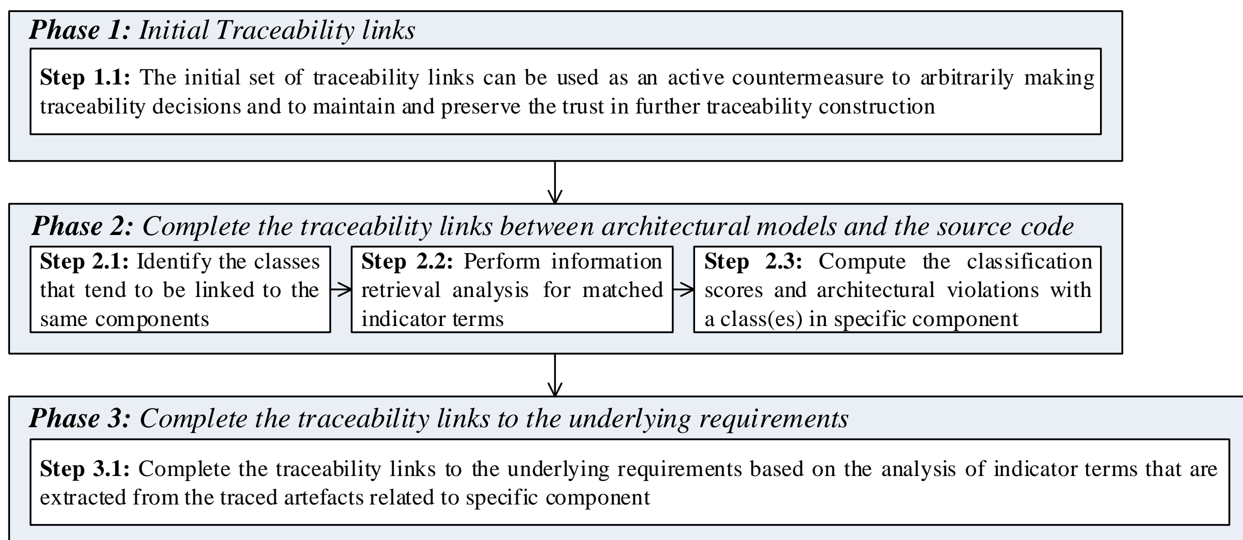

Fig. 5: An overview of the TRACEABILITY COMPLETION pattern

Two of the empirical investigations have shown that architecture-level software understanding [Javed and Zdun b] and evolution analysis [Javed and Zdun c] tasks are significantly better performed when providing traceability links, whereas no significant differences with regard to the experience of the participants are observed [Javed and Zdun b]. Another study has shown that traceability is more important in larger software systems [Javed and Zdun a]. In these particular studies, the completion of traceability links was based on the analysis of initial traceability links.

Let us consider the initial set of traceability links constructed for the Soomla Android store Version 2.0. The completion of traceability links is performed in a particular manner: The *VirtualGood*, *VirtualCurrency*, *VirtualCurrencyPack* and *NonConsumableItem* classes *«extends» AbstractVirtualItem* class. Furthermore, the *StaticPriceModel* and *BalanceDrivenPriceModel* classes *«extends» AbstractPriceModel* class. Based on

the classification scores and no violation of the architectural model, they are mapped to the *StoreAssets* component. The analysis of indicator terms identified fourteen candidate classes, in which the *KeyValueDatabase* and *StoreDatabase* classes, detected based on the 'Database' term are mapped to the *DatabaseServices* component due to the higher score and architectural conformance. In addition, the *KeyValueStorage* and *NonConsumeableItemStorage* classes, detected based on the 'Storage' term are mapped to the *DatabaseServices* component. It is detected that the relationships of six remaining classes are covered with the mapped classes. Besides that, another nine classes are mapped based on the analysis of classification scores and architectural conformance. The identification of underlying requirements is based on the matched indicator terms with the traced artefacts related to a specific component.

Asuncion et al. [Asuncion et al. 2010] propose an architecture-centric approach to support traceability between architecture models in the ArchStudio tool and other architecture artefacts, including Web Sites, PDF files, Word files, Powerpoint presentations, and Excel spreadsheets. This traceability link recording approach is combined with topic modelling, a widely-used machine learning technique for automatically inferring semantic topics from a text corpus. Mirakhorli and Cleland-Huang [Mirakhorli et al. 2012] utilizes information retrieval and machine learning methods to train a classifier (e.g., training with tactic descriptions and code snippets), to detect the tactic-related classes. However, the probabilistic classifier computes the weight scores to evaluate the likelihood of unconstructed traceability link.

3.4 Pattern: Continuous Traceability Maintenance

You want to perform continuous maintenance of software architecture traceability links in response to smaller evolutionary changes in requirements, architecture design or source code.

How can you maintain software architecture traceability links for smaller evolutionary changes by reducing the human effort required?

Traceability links need to be maintained as a project evolves so that up-to-date traceability links would be available when needed. In this context, three types of change can be considered: (i) the changes that have no impact on the related elements, (ii) the changes that have impact on the related elements, but do not require structural changes and (iii) the change that have impact on the related element and, due to changes in the structure, also on traceability links. On the one hand, the automated reconstruction of all traceability links upon modifications in development artefacts lead towards problems caused by overwriting manual changes. On the other hand, the manual maintenance of traceability links is labour-intensive and error-prone; in particular, it comprises of recognizing the maintenance task, navigating the relevant software artefacts and performing the required changes.

The evolutionary changes can be monitored to capture the change events. In particular, they are matched with the predefined rules to direct the update of impacted traceability links. Each rule contains a name, description and alternative ways to perform the particular activity.

The constructed traceability links need to be maintained as a project evolves; otherwise, they get lost or represent false links. A step by step degradation of traceability links lead towards traceability decay. The idea of the CONTINUOUS TRACEABILITY MAINTENANCE pattern is to continuously monitor the evolutionary changes in order to update the impacted traceability links immediately following changes of the traced artefacts. The continuous maintenance of traceability links is triggered by changes to any of the software artefacts

(e.g., architecture component) that, in turn, can be triggered by changes to artefacts within a traceability chain (e.g., underlying requirements and the code classes that implement the component). To update the traceability links, the nature of change should be analysed to determine what updates are necessary. Six types of changes require traceability update: (i) adding an element, (ii) deleting an element, (iii) replacing an element, (iv) merging several elements into one whole, (v) splitting an element into parts, and (vi) modifying an element by adding or removing parts.

This can be realized by observing these change events: In order to perform the continuous maintenance of traceability links, an event generator recognizes the changes in software assets, whereas the rules are used to retrieve the impacted links and define updates for links.

Event-based support for updating traceability links is for instance integrated in ARTiSAN Studio[2] and Sparx Enterprise Architect[3] to handle the necessary elementary change events and permit for the manipulation of traceability relations from outside the tool.

3.5 Pattern: On-Demand Traceability Maintenance

You want to perform on-demand maintenance of software architecture traceability links in the context of different versions of a software project.

How can you perform an overall update of traceability links in response to substantial evolution of a software system?

In case of a substantial evolution of a software system, such as a new major version, the traceability links often need to be constructed again. Just consider the effort in large scale, maybe even highly distributed software projects for such large changes: Redoing the complete work that went into the first construction of an entire traceability link set should be avoided. For instance, in this context, the automated approaches and tools re-generate all traceability links and require validation (assuring credibility) of final traceability links from the developers.

The extracted features of a new component are used to detect the previous traceability links related to the particular component, which are then used to perform reuse and adaptation of traceability links based on the matches and mismatches, respectively.

After a substantial evolution of a software system, such as a new major version, the previous traceability links become outdated. The ON-DEMAND TRACEABILITY MAINTENANCE pattern concerns overall update of traceability links in the context of different versions of a software project. In order to reuse and adapt the traceability links in the context of different versions of a software project, the pre-existing traceability links need to be organized as cases. Each traceability case consists of two parts: problem description and solution. The former describes the components as software architecture elements and their interconnections, while the later contains the traceability links to artefacts produced in the other activities of the development process, such as requirements and implementation. Afterwards, the extracted features of a new component are used to identify and prioritize the similar cases, i.e., a set of candidate components whose characteristics/features match with the new architectural component. For reuse and adaptation of traceability links, three aspects

[2]http://www.atego.com/de/download-center/products/category/artisan-studio/
[3]http://www.sparxsystems.de/sitemap/

are considered: exactly matched traceability links, partially matched traceability links and unmatched traceability links. The already verified traceability links from the past (i.e., reused links) in both matched and partially matched cases might be omitted from validation. That is, only newly constructed traceability links for evolutionary changes would be made available to human analyst for validation. The unchanged traceability links, if available, should be used as an active countermeasure to arbitrarily making traceability decisions, and to maintain and preserve the trust in further traceability construction. The reuse and adaptation is performed in four steps: (i) reuse the traceability links for matched component requirements and code classes, (ii) an information retrieval analysis is performed based on the indicator terms for variations in architectural components, requirements and undetected classes, (iii) the 'function name' and 'global variable' dependencies of reused classes are computed, and (iv) the mutual and tightly coupled links are adapted. The adapted (i.e., newly constructed) traceability links can be verified by a human analyst and stored in the dedicated case base for future problem solving situation. An overview of the ON-DEMAND TRACEABILITY MAINTENANCE pattern is presented in Figure 6.

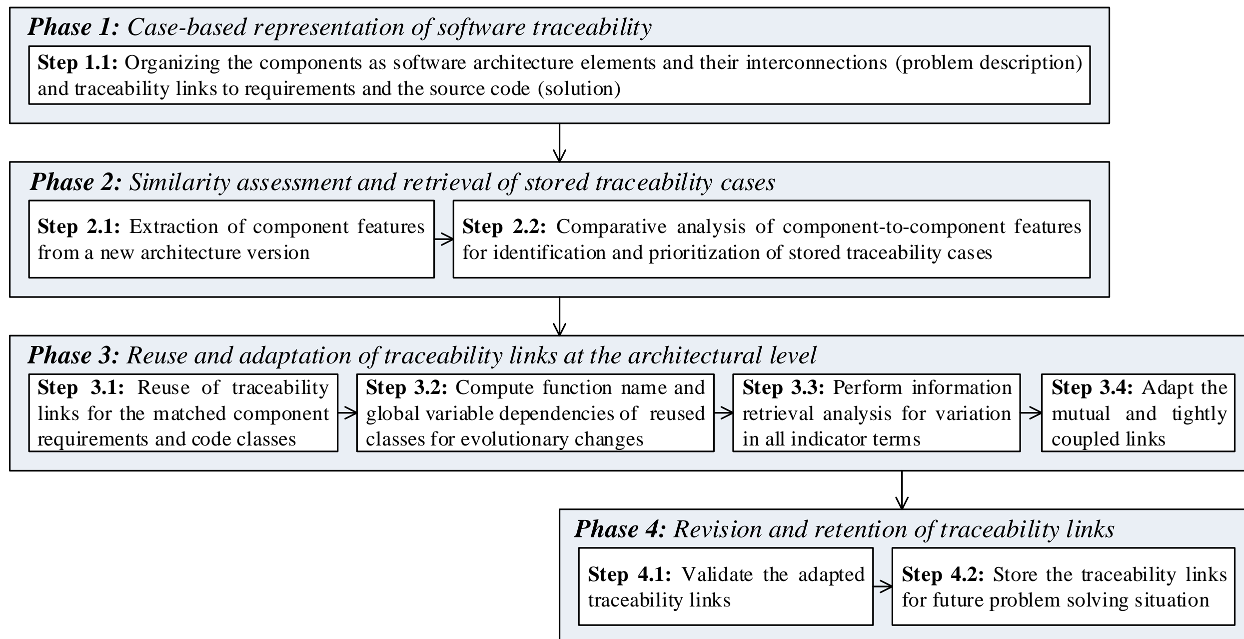

Fig. 6: An overview of the ON-DEMAND TRACEABILITY MAINTENANCE pattern

Let us consider the reuse and adaptation of StoreAssets traceability links of the Soomla Android store Version 2.0 to the Version 3.6.17: The extracted features of the StoreAssets component are used to measure the component-to-component similarities with the previous components located in the case-base. Consequently, a traceability case related to the particular component in Version 2.0 is selected because of the highest similarity score ($Sim = 9.5$) compared to other components. In order to reuse the previous traceability links, the solution part of retrieved case is matched against the textual requirements and the code classes of the Version 3.6.17. Afterwards, for adaptation of traceability links, the function name and global variable dependencies of the reused classes, as well as the information retrieval analysis based on the variation in architectural components, requirements and undetected classes is performed.

The first underlying requirement of the retrieved StoreAssets component is partially matched with the listed textual requirement in Version 3.6.17, the price models are no longer supported; however, a reward feature is introduced. Accordingly, three classes (*AbstractPriceModel*, *StaticPriceModel* and *BalanceDrivenPriceModel*) are not detected and four source code classes (*VirtualCurrency*, *VirtualCurrencyPack*, *VirtualItemNotFoundException* and *JSONConsts*) are reused. In the partially matched case, both reuse and adaptation needs to be performed. The adaptation of traceability links for a particular requirement is performed in three steps. First, the function name and global variable dependencies of reused classes are computed, which leads to the identification of eleven classes. Second, the information retrieval analysis based on the indicator terms in new requirement text is performed, which leads to the identification of nine classes. This covers the variation in new component description. Besides that, the undetected classes are not used for adaptation as the indicator terms in excluded requirement text are matched with the deleted classes. Finally, the mutual and tightly coupled classes (*BadgeReward*, *RandomReward*, *Reward*, *SequenceReward*, *VirtualItemReward*, *Schedule*, *SoomlaEntity* and *JSONFactory*) are linked with the requirement in Version 3.6.17. Note that the adaptation process correctly identified all the classes realizing the particular requirement within a StoreAssets component.

The second requirement of the retrieved StoreAssets component is exactly matched with the listed textual requirement of the Version 3.6.17; whereas two source code classes (*VirtualCategory* and *VirtualGood*) are reused and three classes (*AbstractVirtualItem*, *GoogleMarketItem* and *NonConsumeableItem*) are not detected. In the adaptation process, the function name and global variable dependencies of the reused classes are first computed. This led to the identification of five classes, in which four of the classes are strongly linked as means of «extends» relationship. The information retrieval-based analysis for indicator terms of undetected classes is later performed, which leads to the identification of twelve classes. To perform rather targeted adaptation, the mutual terms in all undetected classes, if available, would be used for the recovery. Finally, the mutual and tightly coupled classes are linked with the second requirement of the StoreAssets component. This process identified all the classes realizing the particular requirement of the StoreAssets component.

3.6 Pattern: Traceability Quality Checks

You want to improve external system qualities (e.g. analysability, maintainability) that are affected by the way the component architecture is partitioned and the traceability links are constructed. You have already created the complete traceability links between the given system's architecture and its source code.

How can you improve the quality of the component architecture partitioning using traceability links so that desired external system qualities are improved?

The above given traceability construction techniques do not take into account the impact of generated traceability links on external software qualities. Therefore, they cannot foresee how external system qualities are affected by created traceability links. Furthermore, the rules used in the given traceability construction techniques often negatively affect external system qualities. For example, a rule that uses the dependencies among the source code classes might assign too many classes to one component and not enough to another one that would poorly affect the architecture level system analysability[4].

[4]The system should be decomposed into a limited number of components of roughly the same size in order to improve its analysability [Bouwers et al. 2011a].

Estimate external system qualities using appropriate software metrics that can be extracted from the system itself. Use the quality estimations as a guide to step by step improve the quality of the component architecture partitioning based on traceability links.

To find relationships between external system qualities and software metrics used to describe the system itself, various techniques can be used. Some of the examples include empirical studies where the subjects performances or ratings are used to measure external qualities that are further correlated with software metrics, careful manual examinations of qualities that are then linked with the metrics values, etc. For an overview please have a look at a mapping study provided by Stevanetic et al. [Stevanetic and Zdun 2015]. This mapping study provides a systematic review of software metrics that can be used to estimate or predict external qualities of higher level architectural structures. Several external qualities like maintainability, bug severity, modularization, integrability, reusability, etc. have been studied (see [Stevanetic and Zdun 2015] for more details). In addition to the given studies there exist several empirical studies on software metrics that can be used to measure the understandability of architectural components [Stevanetic et al. 2014; Stevanetic and Zdun 2014b].

Based on the calculated software metrics, we can estimate a desired external quality. The calculations of metrics as well as external qualities can be fully automated, based on the traceability information. Estimating an external quality implies that a quality can be evaluated by assigning it an absolute or relative value. An example for the absolute value of a quality would be that the analysability level of the system is 0.23 while an example for the relative value would be the distribution of the values of some metric among the components in the architecture so that an analyst can highlight the components with very high or very low metric values that poorly affect a desired external quality.

To apply the TRACEABILITY QUALITY CHECKS pattern, an analyst should examine the metrics values affecting an unacceptably high or low value for the measured quality (e.g. response time for assessing the understandability) and change the component architecture partitioning (e.g. by assigning some classes to a new component or tweaking traceability links by removing unnecessary links, introducing new links or modifying them) in order to improve the situation. Changing the component architecture partitioning cannot be done arbitrarily. For example, moving classes that directly implement the functionalities provided by a certain component to some other component does not make sense. By gradually performing the changes, an analyst can observe the changes in an observed quality and therefore pursue the improvements, until a desired quality level is achieved. Performing the changes involves a manual effort, i.e. the participation of an analyst who should consider which changes would make sense. For example, an analyst can change the component model by dividing a given component into two or pursue some source code changes in order to enable the appropriate creation of traceability links that would improve an observed external quality. Based on what we explained above, this pattern can be semi-automated, i.e. the calculations of the metrics and external qualities can be fully automated while performing the changes and improvements requires a manual effort. An overview of the TRACEABILITY QUALITY CHECKS pattern is presented in Figure 7.

Here we provide an example of how the TRACEABILITY QUALITY CHECKS pattern can be used in case of Soomla Android store Version 2.0. Assume that the initial component view of the system is the one shown in the left side of Figure 9. In applying the TRACEABILITY QUALITY CHECKS pattern, we focus on improving the analysability quality characteristic. Namely, Bouwers et al. found that the components should be balanced in size in order to facilitate the system's analysability (location of possible failures/bugs in the system) [Bouwers et al. 2011b]. In our case, instead of components' size we consider the effort required to understand a component (i.e. the understandability effort) which provides better analysability estimation. For more information on how to calculate the understandability effort for a component, please have a look at the following empirical studies [Stevanetic and Zdun 2014a; 2014b; 2016].

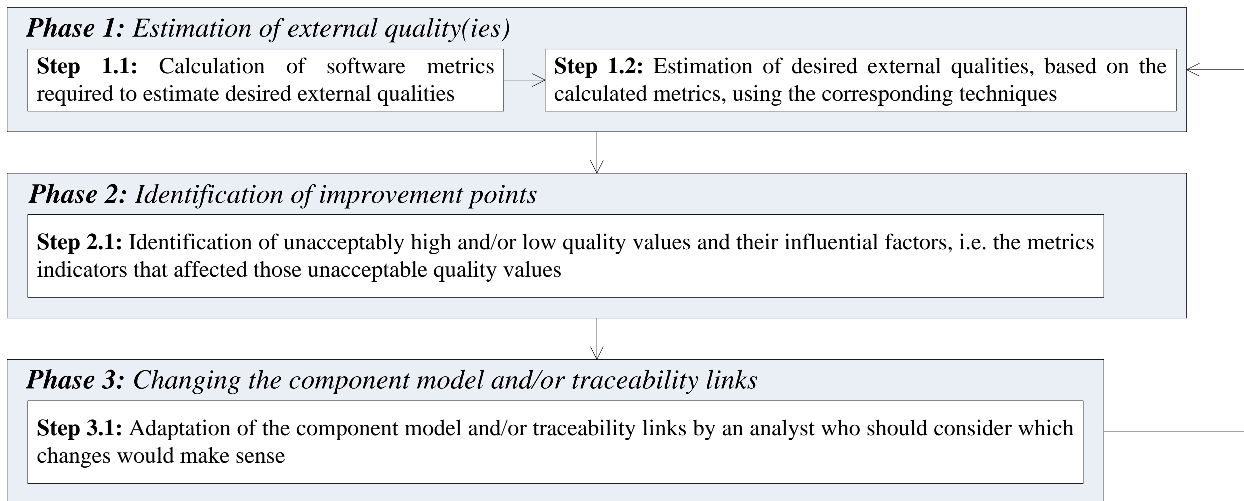

Fig. 7: An Overview of the TRACEABILITY QUALITY CHECKS Pattern

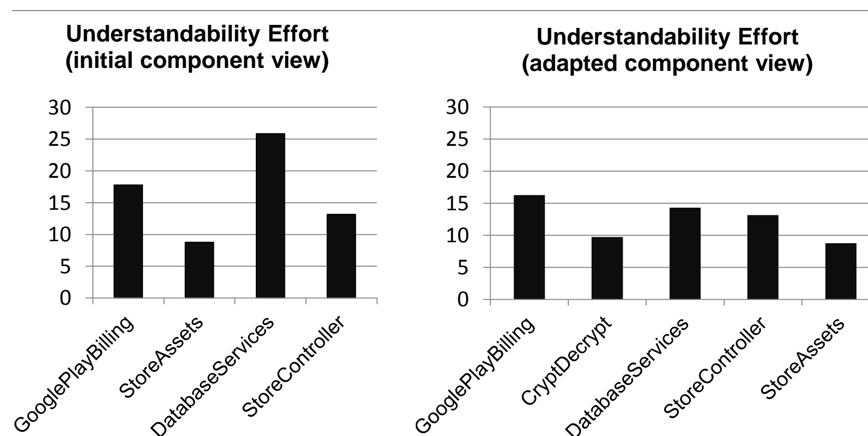

Fig. 8: Understandability Effort for the Initial and Adapted Component Views

To improve the analysability of the initial component view, we have performed the steps described in Figure 7 with respect to the studied example:

Step 1: Using the automatically generated complete traceability links for the initial component view, we calculate the analysability of the system, using the metric proposed by Bouwers et al. [Bouwers et al. 2011b]. As mentioned above, instead of components' size, we calculate the understandability effort required to understand a component, using the metric proposed in [Stevanetic and Zdun 2014b]. The distribution of the understandability effort among the components is shown in Figure 8. The corresponding analysability metric is 0.33.

Step 2: As we can see from Figure 8, in the initial component view the understandability effort is unevenly distributed over the components which decreases the analysability of the system. For instance, Component *DatabaseServices* requires very high effort to be understood compared to Component *StoreAssets*. After examining the traceability links, we find that 18 classes are assigned to Component *DatabaseServices*

compared to 10 classes that are assigned to Component *StoreAssets*. Therefore, possible improvement points would be to reassign the classes from Component *DatabaseServices* to some other components or to divide it into several smaller components.

Step 3: After careful examination of the traceability links, we find that the following changes can be applied in order to improve the analysability: 5 classes from Component *DatabaseServices* (*AESObfuscator*, *Base64DecoderExeption*, *Base64*, *ObscuredSharedPreferences*, and *Editor*) can be regrouped into a new component *CryptDecrypt* and Class *Security* can be moved from Component *GooglePlayBilling* to new Component *CryptDecrypt*. After the given changes, the analyzability of the system is improved to 0.5. The distribution of the understandability effort for the adapted component view is shown in the right side of Figure 8, while the adapted component view is shown in the right side of Figure 9.

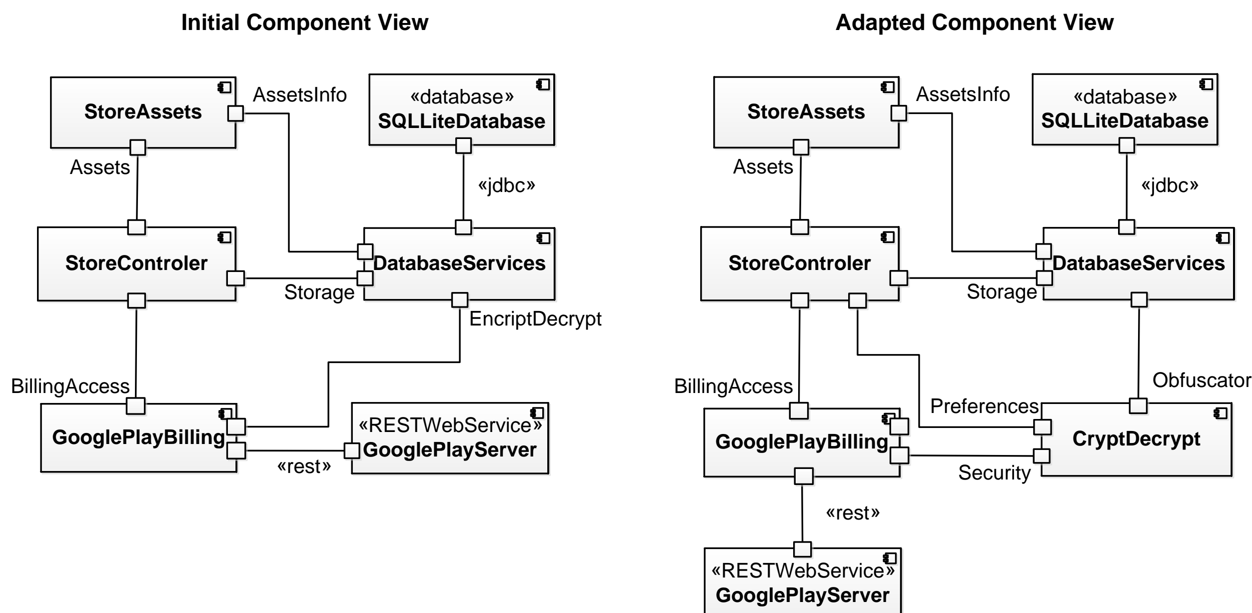

Fig. 9: Initial and Adapted Component Views

In terms of known uses, there exist several approaches and software tools that provide support for the estimation of external system qualities, based on software metrics calculations that utilize the traceability information among architectural structures and system implementation (for an overview please refer to [Stevanetic and Zdun 2015]). For example, Bhattacharya et al. [Bhattacharya et al. 2012] showed how graph-based metrics can be used to predict bug severity, maintenance effort and defect-prone releases. Gupta et al. [Gupta and Chhabra 2009] presented package coupling metrics that show strong correlation with the package understandability. Hwa et al. [Hwa et al. 2009] proposed a hierarchical metrics model to assess understandability of modularization in large-scale object-oriented software. Ma et al. [Ma et al. 2010] proposed a hierarchical set of metrics in terms of coupling and cohesion for measuring the complexity at various levels of granularity, i.e. graph, class (and object) and source code. Empirical evaluations of these metrics indicated that they complement well traditional software metrics and provide more useful information about fault-prone classes. Sarkar et al. [Sarkar et al. 2008] discussed a set of metrics that can be used to characterize large object-oriented software systems with regard to the quality of modularization. The modularization quality

is assessed with respect to the APIs of the modules as well as with respect to the object-oriented intermodule dependencies caused by inheritance, associational relationships, state access violations, fragile base-class design, etc. In a broader context, metrics based quality improvement is very often used to improve different business objectives (e.g. sales, marketing, or production). In that context, business dashboards are today used as the most important part of Business Performance Management (BPM) [Few 2006]. They capture an organization's key performance indicators (KPIs) and enable informed decisions for quality improvements to be made, based on those KPIs [Few 2006].

4. RELATED WORK

There are a few studies that consider patterns for software traceability. The research in [Delgoshaei and Austin 2012] focuses on understanding the role of software patterns (e.g., model-view-controller) and mixtures of graph and tree visualization for ontology-enabled traceability of requirements to elements of finite-state machine behaviour (e.g., actions, states, transitions and guard conditions). The patterns for solving the requirement to component tractability problems in agile development processes are proposed [Ghazarian 2008]. In particular, traceability is achieved as a result of the source code conformance to a set of traceability patterns.

The proposed softgoal traceability patterns can be used during the goal analysis phase in order to support the generation of architectural design elements [Fletcher and Cleland-Huang 2006]. The architect can select a predefined pattern to transform it into a UML class diagram. The generated diagram can be later modified by changing the visual layout, adding additional classes, and even changing existing ones; however, the architect will be warned of the potential modification conflicts that impact the goals of the system. In this context, these patterns provide bidirectional traceability for monitoring the compliance of an architectural design to its stated goals.

5. CONCLUDING REMARKS

This paper presents a pattern language for construction and maintenance of software architecture traceability links to requirements and source code. It was mined from the following resources: First, systematic literature reviews on software architecture traceability approaches, tools and architecture level software metrics have been performed. Second, for identification of potential architectural violation points and their resolutions, software architecture traceability links for multiple open-source software systems have been constructed. The construction of traceability links was a very time-consuming task that was completed over the course of three months. Third, the empirical investigations on human analyst performance in software architecture traceability and metrics have been performed. We have then actively searched for more known uses for each pattern in the literature and tool landscape. That is, our five patterns have been mined from a broader set of sources than only software tools, as many of the existing traceability practices are still a research topic. It was our goal to find those practices in the field that are mature and can be recommended for practical adoption or have been adopted in practices already. For future work we plan to build a catalogue of guidelines as best practices for traceability reuse and adaptation across projects, organizations, domains, product lines and supporting tools.

6. ACKNOWLEDGEMENTS

The authors would like to thank the shepherd Martin Filipczyk for his valuable comments. This work is supported by the Austrian Science Fund (FWF), under project P24345-N23.

REFERENCES

Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. 2002. Recovering Traceability Links Between Code and Documentation. *IEEE Trans. Softw. Eng.* 28, 10 (Oct. 2002), 970–983. DOI:http://dx.doi.org/10.1109/TSE.2002.1041053

Hazeline U. Asuncion, Arthur U. Asuncion, and Richard N. Taylor. 2010. Software Traceability with Topic Modeling. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 95–104. DOI:http://dx.doi.org/10.1145/1806799.1806817

Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. 1999. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

PerOlof Bengtsson, Nico Lassing, Jan Bosch, and Hans van Vliet. 2004. Architecture-level Modifiability Analysis (ALMA). *J. Syst. Softw.* 69, 1-2 (Jan. 2004), 129–147. DOI:http://dx.doi.org/10.1016/S0164-1212(03)00080-3

Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. 2012. Graph-based analysis and prediction for software evolution. In *ICSE'12*. 419–429.

Eric Bouwers, Jose P. Correia, Arie Deursen, and Joost Visser. 2011a. Quantifying the Analyzability of Software Architectures. In *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*. IEEE, 83–92. DOI:http://dx.doi.org/10.1109/wicsa.2011.20

E. Bouwers, J. P. Correia, A. v. Deursen, and J. Visser. 2011b. Quantifying the Analyzability of Software Architectures. In *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*. 83–92. DOI:http://dx.doi.org/10.1109/WICSA.2011.20

Georg Buchgeher and Rainer Weinreich. 2011. Automatic Tracing of Decisions to Architecture and Implementation. In *Proceedings of the 2011 Ninth Working IEEE/IFIP Conference on Software Architecture (WICSA '11)*. IEEE Computer Society, Washington, DC, USA, 46–55. DOI:http://dx.doi.org/10.1109/WICSA.2011.16

Xiaofan Chen and John Grundy. 2011. Improving Automated Documentation to Code Traceability by Combining Retrieval Techniques. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE Computer Society, Washington, DC, USA, 223–232. DOI:http://dx.doi.org/10.1109/ASE.2011.6100057

Paul Clements, Rick Kazman, and Mark Klein. 2002. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

David Cuddeback, Alex Dekhtyar, and Jane Hayes. 2010. Automated Requirements Traceability: The Study of Human Analysts. In *Proceedings of the 2010 18th IEEE International Requirements Engineering Conference (RE '10)*. IEEE Computer Society, 231–240. DOI:http://dx.doi.org/10.1109/RE.2010.35

A. Dekhtyar, O. Dekhtyar, J. Holden, J.H. Hayes, D. Cuddeback, and Wei-Keat Kong. 2011. On human analyst performance in assisted requirements tracing: Statistical analysis. In *Proceedings of the 2011 19th IEEE International Requirements Engineering Conference (RE '11)*. IEEE Computer Society, 111–120. DOI:http://dx.doi.org/10.1109/RE.2011.6051649

Parastoo Delgoshaei and Mark Austin. 2012. Software Patterns for Traceability of Requirements to Finite State Machine Behavior. In *Proceedings of the Conference on Systems Engineering Research, CSER 2012, St. Louis, MO, USA, March 19-22, 2012*. 214–219. DOI:http://dx.doi.org/10.1016/j.procs.2012.01.045

Stephen Few. 2006. *Information Dashboard Design: The Effective Visual Communication of Data*. O'Reilly Media, Inc.

Jesse Fletcher and Jane Cleland-Huang. 2006. Softgoal Traceability Patterns. In *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE '06)*. IEEE Computer Society, Washington, DC, USA, 363–374. DOI:http://dx.doi.org/10.1109/ISSRE.2006.42

Arbi Ghazarian. 2008. Traceability Patterns: An Approach to Requirement-component Traceability in Agile Software Development. In *Proceedings of the 8th Conference on Applied Computer Scince (ACS'08)*. World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 236–241. http://dl.acm.org/citation.cfm?id=1504034.1504078

Orlena Gotel, Jane Cleland-Huang, Jane Huffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grünbacher, Alex Dekhtyar, Giuliano Antoniol, and Jonathan I. Maletic. 2012. *The Grand Challenge of Traceability (v1.0)*. Springer-Verlag London Limited. 343–409 pages.

Varun Gupta and Jitender Kumar Chhabra. 2009. Package coupling measurement in object-oriented software. *J. Comput. Sci. Technol.* 24, 2 (March 2009), 273–283. DOI:http://dx.doi.org/10.1007/s11390-009-9223-6

Thomas Haitzer and Uwe Zdun. 2012. DSL-based support for semi-automated architectural component model abstraction throughout the software lifecycle. In *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures (QoSA '12)*. ACM, New York, NY, USA, 61–70. DOI:http://dx.doi.org/10.1145/2304696.2304709

Maen Hammad, Michael L. Collard, and Jonathan I. Maletic. 2011. Automatically identifying changes that impact code-to-design traceability during evolution. *Software Quality Control* 19, 1 (March 2011), 35–64. DOI:http://dx.doi.org/10.1007/s11219-010-9103-x

Donna Harman. 1992. Ranking algorithms. In *Information Retrieval: Data Structures & Algorithms*, William B. Frakes and Ricardo Baeza-Yates (Eds.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 363–392.

Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. 2006. Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods. *IEEE Trans. Softw. Eng.* 32, 1 (Jan. 2006), 4–19. DOI:http://dx.doi.org/10.1109/TSE.2006.3

Jimin Hwa, Sukhee Lee, and Yong Rae Kwon. 2009. Hierarchical Understandability Assessment Model for Large-Scale OO System. In *Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference (APSEC '09)*. IEEE Computer Society, Washington, DC, USA, 11–18. DOI:http://dx.doi.org/10.1109/APSEC.2009.60

IEEE. 1990. IEEE Standard Std 610.12, Glossary of Software Engineering Terminology. (Dec 1990), 1–84. DOI:http://dx.doi.org/10.1109/IEEESTD.1990.101064

Muhammad Atif Javed, Srdjan Stevanetic, and Uwe Zdun. 2015. Cost-Effective Traceability Links for Architecture-Level Software Understanding: A Controlled Experiment. In *Proceedings of the 24th Australasian Software Engineering Conference (ASWEC)*. ACM, 5. DOI:http://dx.doi.org/10.1145/2811681.2811695

Muhammad Atif Javed and Uwe Zdun. On the Effects of Traceability Links in Differently Sized Software Systems. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering (EASE 2015)*. ACM.

Muhammad Atif Javed and Uwe Zdun. The Supportive Effect of Traceability Links in Architecture-Level Software Understanding: Two Controlled Experiments. In *Proceedings of the 11th Working IEEE/IFIP Conference on Software Architecture (WICSA 2014)*. IEEE, 215–224. DOI:http://dx.doi.org/10.1109/WICSA.2014.43

Muhammad Atif Javed and Uwe Zdun. The Supportive Effect of Traceability Links in Change Impact Analysis for Evolving Architectures – Two Controlled Experiments. In *14th International Conference on Software Reuse (ICSR 2015)*. Springer link.

Muhammad Atif Javed and Uwe Zdun. 2014. A Systematic Literature Review of Traceability Approaches Between Software Architecture and Source Code. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE '14)*. ACM, New York, NY, USA, Article 16, 10 pages. DOI:http://dx.doi.org/10.1145/2601248.2601278

Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. 2007. Recovering Traceability Links in Software Artifact Management Systems Using Information Retrieval Methods. *ACM Trans. Softw. Eng. Methodol.* 16, 4, Article 13 (Sept. 2007). DOI:http://dx.doi.org/10.1145/1276933.1276934

Yutao Ma, Keqing He, Bing Li, Jing Liu, and Xiao-Yan Zhou. 2010. A Hybrid Set of Complexity Metrics for Large-Scale Object-Oriented Software Systems. *J. Comput. Sci. Technol.* 25, 6 (2010), 1184–1201. http://dblp.uni-trier.de/db/journals/jcst/jcst25.html#MaHLLZ10

Patrick Mäder and Orlena Gotel. 2012. Towards Automated Traceability Maintenance. *J. Syst. Softw.* 85, 10 (Oct. 2012), 2205–2227. DOI:http://dx.doi.org/10.1016/j.jss.2011.10.023

Julie McClure. 2011. CMMI for Development V 1.3 by Mary Beth Chrissis, Mike Konrad and Sandy Shrum. *SIGSOFT Softw. Eng. Notes* 36, 4 (Aug. 2011), 34–35. DOI:http://dx.doi.org/10.1145/1988997.1989007

Mehdi Mirakhorli, Yonghee Shin, Jane Cleland-Huang, and Murat Cinar. 2012. A tactic-centric approach for automating traceability of quality concerns. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012)*. IEEE Press, Piscataway, NJ, USA, 639–649. http://dl.acm.org/citation.cfm?id=2337223.2337298

Elena Navarro and Carlos E. Cuesta. 2008. Automating the Trace of Architectural Design Decisions and Rationales Using a MDD Approach. In *Proceedings of the 2nd European conference on Software Architecture (ECSA '08)*. Springer-Verlag, Berlin, Heidelberg, 114–130. DOI:http://dx.doi.org/10.1007/978-3-540-88030-1_10

Tien N. Nguyen, Ethan V. Munson, and Cheng Thao. 2005. Object-oriented Configuration Management Technology can Improve Software Architectural Traceability. In *Proceedings of the Third ACIS Int'l Conference on Software Engineering Research, Management and Applications (SERA '05)*. IEEE Computer Society, Washington, DC, USA, 86–93. http://dl.acm.org/citation.cfm?id=1105925.1106170

Nan Niu, Anas Mahmoud, Zhangji Chen, and Gary Bradshaw. 2013. Departures from Optimality: Understanding Human Analyst's Information Foraging in Assisted Requirements Tracing. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 572–581.

Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery. In *Proceedings of the 18th International Conference on Program Comprehension (ICPC '10)*. IEEE, 68–71. DOI:http://dx.doi.org/10.1109/ICPC.2010.20

European O. Radio Technical Commission for Aeronautics RTCA. 1992. *RTCA: Software Considerations in Airbone Systems and Equipment Certification Standard Document no. DO-178B/ED-12B*. Technical Report.

Santonu Sarkar, Avinash C. Kak, and Girish Maskeri Rama. 2008. Metrics for Measuring the Quality of Modularization of Large-Scale Object-Oriented Software. *IEEE Trans. Softw. Eng.* 34, 5 (Sept. 2008), 700–720. DOI:http://dx.doi.org/10.1109/TSE.2008.43

Srdjan Stevanetic, Muhammad Atif Javed, and Uwe Zdun. 2014. 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)*. IEEE Computer Society, Sydney, Australia.

S. Stevanetic and U. Zdun. 2014a. Exploring the Relationships between the Understandability of Architectural Components and Graph-Based Component Level Metrics. In *2014 14th International Conference on Quality Software*. 353–358. DOI:http://dx.doi.org/10.1109/QSIC.2014.21

Srdjan Stevanetic and Uwe Zdun. 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, UK.

Srdjan Stevanetic and Uwe Zdun. 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*. http://eprints.cs.univie.ac.at/4321/

Srdjan Stevanetic and Uwe Zdun. 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)*. http://eprints.cs.univie.ac.at/4632/

Huy Tran, Uwe Zdun, and Schahram Dustdar. 2011. VbTrace: using view-based and model-driven development to support traceability in process-driven SOAs. *Softw. Syst. Model.* 10, 1 (Feb. 2011), 5–29. DOI:http://dx.doi.org/10.1007/s10270-009-0137-0

Xuchang Zou, Raffaella Settimi, and Jane Cleland-Huang. 2010. Improving Automated Requirements Trace Retrieval: A Study of Term-based Enhancement Methods. *Empirical Softw. Engg.* 15, 2 (April 2010), 119–146. DOI:http://dx.doi.org/10.1007/s10664-009-9114-z