KAMP: Karlsruhe Architectural Maintainability Prediction

Johannes Stammel
stammel@fzi.de
Forschungszentrum Informatik (FZI), Karlsruhe, Germany

Ralf Reussner
reussner@kit.edu
Karlsruhe Institute of Technology (KIT), Germany

Abstract: In their lifetime software systems usually need to be adapted in order to fit in a changing environment or to cover new required functionality. The effort necessary for implementing changes is related to the maintainability of the software system. Therefore, maintainability is an important quality aspect of software systems.

Today Software Architecture plays an important role in achieving software quality goals. Therefore, it is useful to evaluate software architectures regarding their impact on the quality of the program. However, unlike other quality attributes, such as performance or reliability, there is relatively less work on the impact of the software architecture on maintainability in a quantitative manner. In particular, the cost of software evolution not only stems from software-development activities, such as re-implementation, but also from software management activities, such as re-deployment, upgrade installation, etc. Most metrics for software maintainability base on code of object-oriented designs, but not on architectures, and do not consider costs from software management activities. Likewise, existing current architectural maintainability evaluation techniques manually yield just qualitative (and often subjective) results and also do concentrate on software (re-)development costs.

In this paper, we present *KAMP*, the Karlsruhe Architectural Maintainability Prediction Method, a quantitative approach to evaluate the maintainability of software architectures. Our approach estimates the costs of change requests for a given architecture and takes into account re-implementation costs as well as re-deployment and upgrade activities. We combine several strengths of existing approaches. First, our method evaluates maintainability for concrete change requests and makes use of explicit architecture models. Second, it estimates change efforts using semi-automatic derivation of work plans, bottom-up effort estimation, and guidance in investigation of estimation supports (e.g. design and code properties, team organization, development environment, and other influence factors).

1 Introduction

During its life cycle a software system needs to be frequently adapted in order to fit in its changing environment. The costs of maintenance due to system adaptations can be very high. Therefore, system architects need to ensure that frequent changes can be done as easy as possible and as proper. This quality attribute of software systems is commonly known as adaptability placed within the wider topic of maintainability.

With respect to [ISO90] we define maintainability as *"The capability of a software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications"*. This *capability of being modified* in our point of view can be quantified by means of the effort it takes to implement changes in a software system. If changes can be done with less effort for one software system alternative than for another it indicates that the maintainability for the first one is better.

An important means for making software engineering decisions is the software architecture, which according to [IEE07] represents *"the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."* It is commonly agreed, that the quality of a software system highly depends on its software architecture. One specific benefit of architectures is that they provide documentation for different activities, also beyond software implementation, such as software management or cost-estimation [PB01]. Software architects nowadays specify software architectures using formal architecture models, like [BKR07]. Ideally, it should already be possible to predict the resulting system's quality on basis of these models. The Palladio [BKR07] approach for instance allows an early performance prediction for component-based software architectures. In the same way, maintainability is highly influenced by the architecture and should be supported within an early design-time prediction. This observation is already considered by qualitative architecture evaluation techniques, such as ATAM or SAAM [CKK05]. However, these approaches mostly produce very subjective results as they offer little guidance through tool support and highly rely on the instructors experience [BLBvV04], [BB99]. Moreover, they only marginally respect follow-up costs, e. g. due to system deployment properties, team organization properties, development environment and generally lack meaningful quantitative metrics for characterizing the effort for implementing changes.

This paper presents as a contribution *KAMP*, the Karlsruhe Architectural Maintainability Method, a novel method for the quantitative evaluation of software architectures based on formal architecture models. One specific benefit of KAMP is that it takes into account in a unified manner costs of software development (re-design, re-implementation) and software management (re-deployment, upgrade installation). Moreover, KAMP combines a top-down phase where change requests are decomposed into several change tasks with a bottom-up phase where the effort of performing these change tasks is estimated. The top-down phase bases on architectural analyses, while the second phase utilises existing bottom-up cost estimation techniques [PB01]. Our approach evaluates the maintainability of an architecture by estimating the effort required to perform certain change requests for a given architecture. By doing this it also differs from classical cost estimation approaches which do not systematically reflect architectural information for cost estimation. The approach is limited to adaptive and perfective maintainability. This includes modifications or adaptations of a system in order to meet changes in environment, requirements and functional specifications, but not the corrections of defects, as also mentioned by the definition of maintainability we previously gave. *Paper Structure:* In Section 2 we summarize related work. In Section 3 we introduce the Karlsruhe Architectural Maintainability Prediction approach. Section 4 gives conclusions and points out future work.

2 Related Work

2.1 Scenario-Based Architecture Quality Analysis

In literature there are several approaches which analyse quality of software systems based on software architectures. In the following paragraphs we discuss approaches which make explicitly use of scenarios. There are already two survey papers ([BZJ04], [Dob02]) which summarize and compare existing architecture evaluation methods.

Software Architecture Analysis Method (SAAM) [CKK05] SAAM was developed in 1994 by Rick Kazman, Len Bass, Mike Webb and Gregory Abowd at the SEI as one of the first methods to evaluate software architectures regarding their changeability (as well as to related quality properties, such as extendibility, portability and reusability). It uses an informally described architecture (mainly the structural view) and starts with gathering change scenarios. Then via different steps, it is tried to find interrelated scenarios, i.e., change scenarios where the intersection of the respective sets of affected components is not empty. The components affected by several interrelated scenarios are considered to by critical and deserve attention. For each change scenario, its costs are estimated. The outcome of SAAM are classified change scenarios and a possibly revised architecture with less critical components.

The Architecture Trade-Off Analysis Method (ATAM) [CKK05] ATAM was developed by a similar group for people from the SEI taking into account the experiences with SAAM. In particular, one wanted to overcome SAAM's limitation of considering only one quality attribute, namely, changeability. Much more, one realised that most quality attributes are in many architectures related, i.e., changing one quality attribute impacts other quality attributes. Therefore, the ATAM tries to identify trade-offs between different quality attributes. It also expands the SAAM by giving more guidance in finding change scenarios. After these are identified, each quality attribute is firstly analysed in isolation. Than, different to SAAM, architectural decisions are identified and the effect (sensitivity) of the design decisions on each quality attribute is tried to be predicted. By this "sensitivity analysis" one systematically tries to find related quality attributes and trade-offs are made explicit. While the ATAM provides more guidance as SAAM, still tool support is lacking due to informal architectural descriptions and the influence of the personal experience is high. (Therefore, more modern approaches try to lower the personal influence, e.g., POSAAM [dCP08].) Different to our approach, change effort is not measured as costs on ATAM.

The Architecture-Level Prediction of Software Maintenance (ALPSM) [BB99] ALPSM is a method that solely focuses on predicting software maintainability of a software system based on its architecture. The method starts with the definition of a representative set of change scenarios for the different maintenance categories (e.g. correct faults or adapt to changed environment), which afterwards are weighted according to the likelihood of occurrence during the systems's lifetime. Then for each scenario, the impact of implementing it within the architecture is evaluated based on component size estimations (called scenario scripting). Using this information, the method finally allows to predict

the overall maintenance effort by calculating a weighted average of the effort for each change scenario. As a main advantage compared to SAAM and ATAM the authors point out that ALPSM neither requires a final architecture nor involves all stakeholders. Thus, it requires less resources and time and can be used by software architects only to repeatedly evaluate maintainability. However, the method still heavily depends on the expertise of the software architects and provides little guidance through tool support or automation. Moreover, ALPSM only proposes a very coarse approach for quantifying the effort based on simple component size measures like LOC.

The Architecture-Level Modifiability Analysis (ALMA) [BLBvV04]
The ALMA method represents a scenario-based software architecture analysis technique specialized on modifiability and was created as a combination of the ALPSM approach [ALPSM] with [Lassing1999a]. Regarding the required steps, ALMA to a large extend corresponds to the ALPSM approach, but features two major advantages. First, ALMA supports multiple analysis goals for architecture-level modifiability prediction, namely maintenance effort prediction, risk estimation and comparison of architecture alternatives. Second, the effort or risk estimation for single change scenarios is more elaborated as it explicitly considers ripple effects by taking into account the responsible architects' or developers' expert knowledge (bottom up estimation technique). Regarding effort metrics, ALMA principally allows for the definition of arbitrary quantitative or qualitative metrics, but the paper itself mainly focuses on lines of code (LOC) for expressing component size and complexity of modification (LOC/month). Moreover, the approach as presented in the paper so far only focuses on modifications relating to software development activities (like component (re-)implementation), but does not take into account software management activities, such as re-deployment, upgrade installation, etc.

2.2 Change Effort Estimation

Top-Down Effort Estimation Approaches in this section estimate efforts in top-down manor. Although they are intended for forward engineering development projects, one could also assume their potential applicability in evolution projects. Starting from the requirement level, estimates about code size are made. Code size is then related somehow to time effort. There are two prominent representatives of top-down estimation techniques: *Function Point Analysis (FPA)* [IFP99] and *Comprehensive Cost Model (COCOMO) II* [Boe00]. COCOMO-II contains three approaches for cost estimation, one to be used during the requirement stage, one during early architectural design stage and one during late design stage of a project. Only the first one and partially the second one are top-down techniques. Although FPA and COCOMO-II-stage-I differ in detail, their overall approach is sufficiently similar to be treated commonly in this paper. In both approaches, the extent of the functionality of a planned software system is quantified by the abstract unit of function points (called "applications points" in COCOMO). Both approaches provide guidance in counting function points given an informal requirements description. Eventually, the effort is estimated by dividing the total number of function points by the productivity of the development team. (COCOMO-II-stage-I also takes the expected degree of software reuse

into account.) In particular COCOMO-II in the later two stages takes additional information about the software development project into account, such as the degree of generated code, stability of requirements, platform complexity, etc. Interestingly, architectural information is used only in a very coarse grained manner (such as number of components). Both approaches require a sufficient amount of historical data for calibration. Nevertheless, it is considered hard to make accurate predictions with top-down estimations techniques. Even Barry Boehm (the author of COCOMO) notes that hitting the right order of magnitude is possible, but no higher accuracy[1].

Bottom-Up Effort Estimation – Architecture-Centric Project Management [PB01] (ACPM) is a comprehensive approach for software project management which uses the software architecture description as the central document for various planning and management activities. For our context, the architecture based cost estimation is of particular interest. Here, the architecture is used to decompose planned software changes into several tasks to realise this change. This decomposition into tasks is architecture specific. For each task the assigned developer is asked to estimate the effort of doing the change. This estimation is guided by pre-defined forms. Also, there is no scientific empirical validation. But one can argue that this estimation technique is likely to yield more accurate prediction as the aforementioned top-down techniques, as (a) architectural information is used and (b) by asking the developer being concerned with the execution of the task, personal productivity factors are implicitly taken into account. This approach is similar to KAMP by using a bottom-up estimation technique and by using the architecture to decompose change scenarios into smaller tasks. However, KAMP goes beyond ACPM by using a formalized input (architectural models must be an instance of a predefined meta-model). This enables tool-support. In addition, ACPM uses only the structural view of an architecture and thus does not take software management costs, such as re-deployment into account.

3 The Karlsruhe Architectural Maintainability Prediction Approach

The Karlsruhe Architectural Maintainability Prediction (KAMP) approach enables software architects to compare architecture alternatives with respect to their level of difficulty to implement a specific change request. For each alternative the effort it takes to implement a change request is estimated. An important measure for maintainability is the change effort necessary for implementation of a change. Therefore, the capability of change effort estimation is very important in the context of maintainability prediction. Our method shows how change effort estimation can be embedded within an architectural maintainability prediction methodology and predicts the maintainability using change requests. This is because an architecture is not able to treat all change requests with the same level of ease. An architecture should be optimized in a way that frequent change requests can be implemented easier than less frequent ones. Our method makes explicit use of meta-modelled software architecture models. From software architecture models our method derives important inputs for change effort estimation in a semi-automatic way. Additionally, we use a bottom-up approach for estimation of change efforts. Thus, our method incorporates

[1] http://cost.jsc.nasa.gov/COCOMO.html

developer-based estimates of work effort. In order to get a higher precision of estimates our method helps in investigation of estimation supports. We do this by explicitly considering several kinds of influence factors, i.e. architectural properties, design and code properties, team organization properties, and development environment properties. *Use Case: Evaluate Single Change Request* One use case is to compare two architecture alternatives given a single change request. The method helps answering the question which alternative supports better the implementation of the given change request. *Use Case: Evaluate Multiple Change Requests* If there are several change requests at hand which occur with different frequencies the method helps answering questions like: Is there a dependency between given change requests? Is there a trade-off situation, where only one change request can be considered? Which architecture alternative provides best for implementation of a given set of change requests?

3.1 Properties of KAMP

Compared to the approaches presented in the related works section above, our approach has the following specific benefits: (a) a higher degree of automation, (b) lower influence of personal experience of the instructor, (c) stronger implicit consideration of personal developer productivity. Different to other approaches, we use architectural models, which are defined in a meta model (the Palladio Component Model). While this it no means to an end, the use of well defined architectural models is necessary to provide automated tool-support for architectural dependency analysis. Such tools are embedded into a general guidance through the effort estimation process.

3.2 Maintainability Analysis Process

As Figure 1 shows the maintainability prediction process is divided into the following phases: Preparation, Analysis, Result Interpretation. Each phase is described below. *Running Example:* In order to get a better understanding of the approach we provide a running example, which considers a client-server business application where several clients issue requests to a server in order to retrieve customer address information stored in a database. There are 100 Clients deployed in the system.

Preparation Phase In the model preparation phase, the software architect at first sets up a *software architecture description* for each architectural alternative. For this we use a meta-modelled component-based architecture model, which can be handled within an tool chain. *Example:* According to our running example the architects create an architecture model of the client-server architecture. They identify two architecture alternatives. In *Architecture Alternative 1 (AA1)* the clients specify SQL query statements and use JDBC to send them to the server. The server delegates queries to the database and sends results back to the client. In *Architecture Alternative 2 (AA2)* client and server use a specific IAddress Interface which allows clients to directly ask servers for addresses belonging to

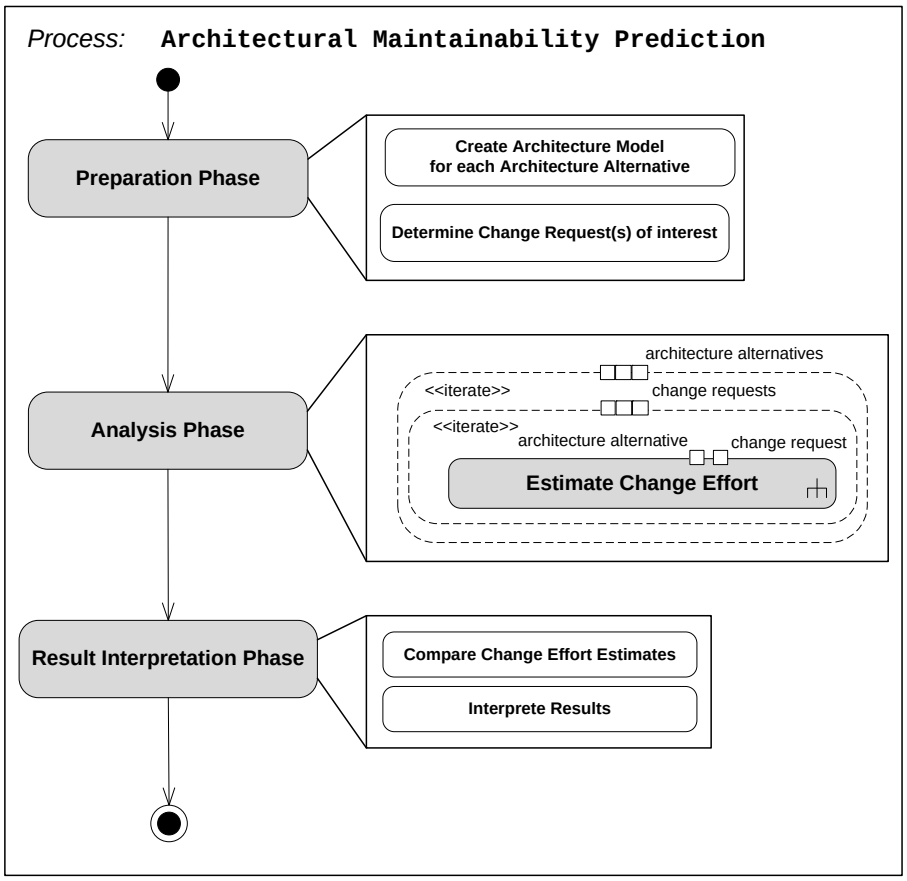

Figure 1: Architectural Maintainability Prediction Process

a certain ID. The server transforms requests into SQL query statements and communicates with the database via JDBC. The architecture model according to Figure 2 consists of a three components (Client, Server, Database). Several Clients are connected to one Server. Server and Database are connected via Interface Ports implementing JDBC Interface. The database schema is considered as a Data Type in the architecture model. Since there are two alternatives the architect creates two models. In alternative *AA1* Client component and Server component are connected via interface ports implementing JDBC Interface. In alternative *AA2* these interface ports use a IAddress Interface instead.

As a second preparation step the architect describes the considered *change requests*. A description of change request contains 1) a name, 2) an informal description of change cause, and 3) a list of already known affected architecture elements. *Example:* Independently from the system structure the architects have to design the database schema. Since they can not agree on a certain database schema they want to keep the system flexible enough

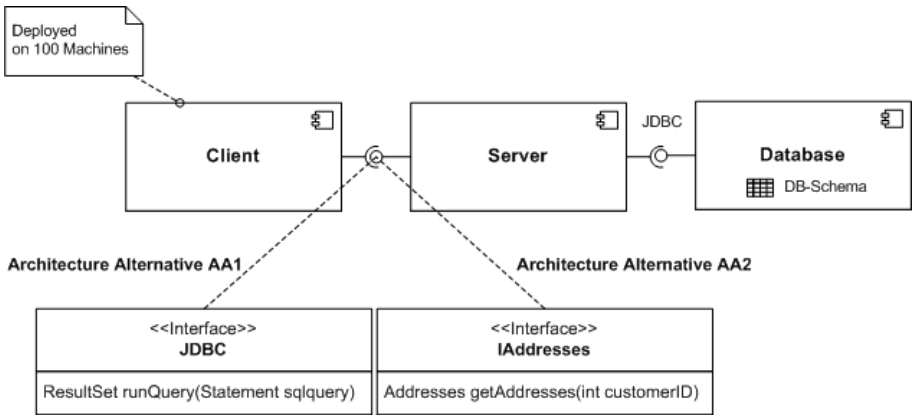

Figure 2: Running Example Architecture Overview

to handle changes to database schema. In the following paragraphs we show how the architects can use our approach to identify the better alternative with respect to expected changes to database schema. Hence, the following change request description is given: *Name:* CR-DBS, *Change Cause:* Database Schema needs to be changed due to internal restructuring, *List of known affected architecture elements:* Data Type "DB-Schema".

Maintainability Analysis Phase In the maintainability analysis phase for each architectural alternative and each change request a *Change Effort Estimation* is done. The process of change effort estimation is shown in Section 3.4. *Example:* The architects in our example want to analyse which architecture alternative (*AA1* or *AA2*) needs less effort to implement Change Request *CR-DBS*.

Result Interpretation Phase Finally the process summarises results, compares calculated change efforts and enriches them with qualitative information and presents them to the software architect.

3.3 Quality Model

Before we go on with the effort estimation process in Section 3.4 it is necessary to introduce the underlying maintainability quality model and used metrics. In general, maintainability characteristics have a rather qualitative and subjective nature. In literature maintainability is usually split up into several sub-characteristics. For example in [ISO90] the quality model divides maintainability into analysability, stability, changeability, testability and maintainability compliance. Unfortunately the given definitions of these terms are rather abstract and therefore not directly applicable. Thus, we first use a systematic way to derive maintainability characteristics and metrics. In order to get a quality model with adequate metrics which provide consequently for specific analysis goals the Goal-Question-Metrics method (GQM) [BCR94] is applied. In this approach, in the first step,

a set of analysis goals is specified by characteristics like analysis purpose, issue, object, and viewpoint as explained here: *1) Purpose:* What should be achieved by the measurement? *2) Issue:* Which characteristics should be measured? *3) Object:* Which artefact will be assessed? *4) Viewpoint:* From which perspective is the goal defined? (e.g., the end user or the development team). The next step is to define questions that will, when answered, provide information that will help to find a solution to the goal. To answer these questions quantitatively every question is associated with a set of metrics. It has to be considered that not only objective metrics can be collected here. Also metrics that are subjective to the viewpoint of the goal can be listed here. Regarding the GQM method we specify the following goal for maintainability analysis: *1) Purpose:* Comparison of Architectural Alternative AA_i and AA_j, *2) Issue:* Maintainability, *3) Object:* Service and Software Architecture with respect to a specific Change Request CR_k, *4) Viewpoint:* Software Architect, Software Developer. The following questions and sub-questions are defined according to the maintainability definitions above.

Question 1:	How much is the **maintenance effort** caused by the architectural alternative AA_i for implementing the Change Request CR_k?
Question 1.1:	How much is the **maintenance workload** for implementing the Change Request CR_k?
Question 1.2:	How much **maintenance time** is spent for implementing the Change Request CR_k?
Question 1.3:	How much are the **maintenance costs** for implementing the Change Request CR_k?

Based on the questions above we divide Maintenance Effort Metrics into Maintenance Workload Metrics, Maintenance Time Metrics, Maintenance Cost Metrics.

Maintenance Workload Metrics Maintenance Workload Metrics represent the amount of work associated with a change. To be more specific we consider several possible *work activities* and then derive counts and complexity metrics according to these work activities. Figure 3 shows how work activity types are found. A work activity is composed of a basic activity which is applied to an artefact of the architecture. Basic activities are *Add*, *Change*, and *Remove*. The considered architecture elements are *Component*, *Interface*, *Operation*, and *Datatype*. This list is not comprehensive, but can be extended with further architecture elements. Usually when we describe changes we refer to the *Implementation* of elements. In the case of *Interface* and *Operation* it is useful to distinguish *Definition* and *Implementation* since there is usually only one definition but several implementations which cause individual work activities. Some architecture meta-models use the concept of *Interface Ports* to bind an interface to a component. From the perspective of work activities an *Implementation of Interface* is equal to an *Interface Port*. In Figure 3 there is also a (incomplete) list of resulting work activity types. The following metrics are defined based on Work Activity Types.

Number of work activities of type WAT_i: This metric counts the number of work activities of type WAT_i. *Complexity annotations for work activity WA_i*: At this point several types of complexity annotations can be used, e. g. number of resulting activities, complexity

Basic Activities	Architecture Elements	Work Activity Set
Add	Component	Add Component
Change	Interface	Change Component
Remove	Operation	Remove Component
	Datatype	Add Interface
	...	...

Figure 3: Work Activity Types

of affected elements in source code, number of affected files, number of affected classes, number of resulting test cases to be run, number of resulting redeployments.
Maintenance Time Metrics These metrics describe effort spent in terms of design and development time. The following metrics are proposed in this category: *Working time for activity* WA_i: This metric covers the total time in person months spent for working on activity WA_i. *Time until completion for activity* WA_i: This metric describes the time between start and end of activity WA_i. *Total time for work plan* WP_i: This metric describes the time between start and end of work plan WP_i.
Maintenance Cost Metrics This subcategory represents maintenance efforts in terms of spent money. *Development costs for activity* WA_i */ work plan* WP_i: The money paid for development in activity WA_i or of work plan WP_i.

3.4 Change Effort Estimation Process

The architect has to describe how the change is going to be implemented. Based on the architecture model he points out affected model artefacts. Our approach proposes several starting points and guided refinements for detection of affected model elements. Starting points represent affected model element which the architect can *directly* identify. Three typical starting points are: 1) an already known Data Type change, 2) an already known Interface Definition change, 3) an already known Component change. Regarding our running example the architect identifies the following direct changes to the architecture: *Change of Datatype Implementation "DB-Schema"*. Directly identified changes are described as work activities in a work plan. A work plan is a hierarchical structured collection of work activities. We now stepwise refine the work plan into small tasks. This means we identify resulting changes and describe high-level changes on a lower level of abstraction. The idea is the effort of such low-level activities can be identified easier and with higher accuracy than high-level coarse grained change requests, in particular, as the latter do not include any architectural informations. There are several types of relations between architectural elements. These relations help to systematically refine change requests and work plans. In the following, these different types of relations are defined, their role in work plan refinement is discussed and examples are given. Thereby, the actual relations depend of the architectural meta model used. However, in one way or another such relation types are present in many architectural meta models. In the following, we present relations

which are present in the Palladio Component (Meta-) Model (PCM) and the Q-ImPrESS Service Architecture Meta-Model (SAMM). **Include- / contains-relations:** Architectural elements which are contained in each other, are in a contains-relationship. This means, that any change of the inner element implies a change of the outer element. Also, any work activity of the outer element can be refined in a set of work activities of the inner elements. *Examples:* A System consists of Components. A Component has Interface Ports (i.e. Implementation of Interface). An Implementation of Interface contains Implementations of Operations. In our running example a change to the database schema also implies a change to database component. Hence, we get another work activity: *Change Component Impl. of "Database" Component*. **References-relations:** Architectural elements which reference (or use) other elements are related by a reference-relation. Such relations are the base for architectural change propagation analyses. Such analyses allow to find potential follow-up changes. This means, that the using entity which references a used entity is potentially affected by a change of the used entity. The direction of the change propagation is reversed to the direction of the reference-relation. This implies, that on a model level one needs to navigate the backward direction of reference-relations. *Examples:* An Interface Port references a Definition of Interface. A Definition of Operation uses a Data Type. A Definition of Interface uses transitively a Data Type. In our running example a change to Data Type "DB-Schema" also affects the JDBC Interface and all Interface Ports which implement the JDBC interface. In both alternatives (*AA1* and *AA2*) the Server component has a JDBC-implementing interface port. Hence, we get the additional work activity: Change Interface Port "JDBC" of Component "Server". In alternative *AA1* client components also implement the JDBC interface. Hence, another work activity is identified: Change Interface Port "JDBC" of Component "Client". By using kinds of relations we systematically identify work activities. For each work activity additional complexity annotations are derived. This can comprise architecture properties (e. g. existing architecture styles or patterns), design / code properties (e. g. how many files or classes are affected), team organization properties (e. g. how many teams and developers are involved), development properties (e. g. how many test cases are affected) system management properties (e. g. how many deployments are affected). If those complexity annotations are present in the architecture model they can be gathered with tool-support. *Examples:* In our architecture meta-model we have a deployment view which specifies deployment complexity of components. In our running example client components are deployed on 100 machines. Thus, a change activity affecting Client components also implies a redeployment on 100 component instances. After work activities are identified and workload metrics are calculated the architect has to assign time effort estimates for all work activities. Following a bottom-up approach the architect presents low-level activities to respective developers and asks them to give estimates. The results of the change estimation process are 1) the work plan with a detailed list of work activities and annotated workload metrics, time effort estimates and costs, and 2) aggregated effort metric values.

Conclusions and Future Work In this paper, we presented a quantitative architecture-based maintainability prediction method. It estimates the effort of performing change requests for a given software architecture. Costs of software re-development *and* costs of software management are considered. By this, KAMP takes a more comprehensive

approach than competing approaches. KAMP combines the strength of a top-down architecture based analysis which decomposes the change requests into smaller tasks with the benefits of a bottom-up estimation technique. We described the central change effort estimation process in this paper. By extrapolating from the properties of our approach compared to other qualitative approaches, we claim the benefits of lower influence of personal experience on the prediction results accuracy and a higher scalability through a higher degree of automation. By using bottom-up estimates we claim the personal productivity is implicitly reflected. However, it is clear that we need to empirically validate such claims. Therefore, we plan as future work a larger industrial case study and several smaller controlled experiments to test the validity of these claims.

Acknowledgements – This work was funded in the context of the Q-ImPrESS research project (http://www.q-impress.eu) by the European Union under the ICT priority of the 7th Research Framework Programme.

References

[BB99] P. Bengtsson and J. Bosch. Architecture level prediction of software maintenance. *Software Maintenance and Reengineering, 1999. Proc. of the Third European Conference on*, pages 139–147, 1999.

[BCR94] V. Basili, G. Caldeira, and H. D. Rombach. *Encyclopedia of Software Engineering, chapter The Goal Question Metric - Approach*. Wiley, 1994.

[BKR07] S. Becker, H. Koziolek, and Ralf H. Reussner. Model-based Performance Prediction with the Palladio Component Model. In *WOSP '07: Proc. the 6th Int. Works. on Software and performance*, pages 54–65, New York, NY, USA, Febr. 2007. ACM.

[BLBvV04] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Architecture-level modifiability analysis (ALMA). *Journ. of Systems and Software*, 69(1-2):129 – 147, 2004.

[Boe00] Barry W. Boehm, editor. *Software cost estimation with Cocomo II*. Prentice Hall, Upper Saddle River, NJ, 2000.

[BZJ04] M.A. Babar, L. Zhu, and R. Jeffery. A framework for classifying and comparing software architecture evaluation methods. *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, pages 309–318, 2004.

[CKK05] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating software architectures*. Addison-Wesley, 4. print. edition, 2005.

[dCP08] David Bettencourt da Cruz and Birgit Penzenstadler. Designing, Documenting, and Evaluating Software Architecture. Technical Report TUM-INFO-06-I0818-0/1.-FI, Technische Universität München, Institut für Informatik, jun 2008.

[Dob02] E. Dobrica, L.; Niemela. A survey on software architecture analysis methods. *Transactions on Software Engineering*, 28(7):638–653, Jul 2002.

[IEE07] ISO/IEC IEEE. Systems and software engineering - Recommended practice for architectural description of software-intensive systems. *ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15*, pages c1–24, 15 2007.

[IFP99] IFPUG. *Function Point Counting Practices Manual*. International Function Points Users Group: Mequon WI, 1999.

[ISO90] ISO/IEC. Software Engineering - Product Quality - Part 1: Quality. *ISO/IEC 9126-1:2001(E)*, Dec 1990.

[PB01] Daniel J. Paulish and Len Bass. *Architecture-Centric Software Project Management: A Practical Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.