Tool-Supported Estimation of Software Evolution Effort in Service-Oriented Systems

Johannes Stammel and Mircea Trifu
FZI Forschungszentrum Informatik
10-14 Haid-und-Neu Str., Karlsruhe, Germany
{stammel, mtrifu}@fzi.de

Abstract

Existing software systems need to evolve in order to keep up with changes in requirements, platforms and technologies. And because software evolution is a costly business, an early and accurate estimation of evolution efforts is highly desirable in any software development project. In this paper we present KAMP, a tool-supported approach, based on change impact analysis, enabling software architects to express a potential design for a given change request at the architecture level and to accurately estimate the evolution effort associated with its implementation in source code. The approach is also used to compare alternative designs before carrying out the implementation work. We apply the KAMP approach on an enterprise SOA showcase and show how it supports the predictable and cost-effective evolution of this software system.

Keywords: *software evolution, effort estimation, architecture, change impact analysis.*

1 Introduction

In order to keep service-oriented software systems up-to-date with changes in requirements, platforms and technologies they need to evolve. Evolution is very common and involves changes to the software system and its architecture. However, architecture changes may have significant impact on software quality attributes such as performance, reliability and maintainability, which is why it is highly desirable to be able to predict these quality impacts.

As a result, the topic of quality impact prediction for evolving service-oriented software systems has been addressed in various research initiatives, such as the Q-ImPrESS [3] research project. The project provides tool-supported methods that predict and analyze quality attributes on architecture level using formalized architecture models.

The project considers multiple quality attributes, i.e., performance, reliability and maintainability, that are conflicting to each other and provides support for balancing them and exploring trade-offs. This paper describes the maintainability prediction approach within the Q-ImPrESS project.

Within this paper, we focus on three types of evolution scenarios. The first type is the requirement of new or changed functionality, i.e., a new service needs to be implemented. The second type is a changed runtime environment, e.g., a new middleware platform, that has to be supported. And the third type is a changed usage profile, e.g., system has to cope with more users without impaired performance.

The implementation of an evolution scenario generates costs and for a good project management it is important to have control over costs and efforts related to evolution. For this reason within the context of the Q-ImPrESS project, we developed the KAMP (Karlsruhe Architectural Maintainability Prediction) approach, a tool-supported approach, based on change impact analysis, enabling software architects to express a potential design for a given change request at the architecture level. Its goal is to estimate the evolution effort associated with its implementation in source code.

The approach is based on architecture modeling using views for static structure, behavior and deployment of services. A service is represented as a deployed component providing service operations to its clients.

Note that evolution effort does not represent only implementation effort. KAMP is able to cover different efforts for different activities, utilizing the information annotated in the architecture models. In particular, KAMP considers management efforts for (re-)deployment and (re-)configuration.

Section 2 discusses foundations about estimation of software evolution efforts, maintainability and change requests. In Section 3 we present the KAMP approach. Section 4 describes the results of an initial case-study applying KAMP. Section 5 summarizes the related work, while section 6 concludes this paper.

2 Foundations

2.1 Effort Estimation

Nowadays, effort estimation is an essential part during planning and execution of software development projects. Most projects have a limited budget, therefore a careful planning is necessary to avoid running out of budget. Effort estimation helps to detect resource problems early and allows for timely corrections, where necessary.

Effort estimation approaches derive their estimation values from planned project scope and from anticipated complexity. In addition most approaches have plenty of input parameters.

Existing effort estimation approaches support different project phases. There are approaches, which focus on the requirements engineering phase, others aim at estimation during the design phase or the implementation phase.

The input parameters depend on the supported project phase and are determined by the project artifacts available at that point, i.e., during the requirement engineering phase the inputs come from the requirements document, during the design phase the inputs are based on design specification and during the implementation phase data and progress values of the running project can be used. The earlier the estimation is done the more imprecise the available data is and the less confident the estimations are.

So, in order to get valid results from estimation approaches the input parameters need to be calibrated to fit the given project context. This is done with data derived from similar projects for example by using a project database. However, such a database is not always available at the beginning. Nevertheless, the input parameters have to be readjusted while the project is progressing in order to fit the actual circumstances.

Therefore estimation should be established as a continuous task during the project life cycle. Since the architecture is one of the central artifacts for managing software development projects, effort estimation should be closely aligned with it, and support for seamless continuous effort estimation during the architecture design phase is highly desirable.

2.2 Maintainability Definition

Maintenance efforts represent a significant part of the total effort of a software development project. During the lifetime of a system, the system has to evolve in order to be still usable.

With respect to [12], 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"*. We focus our approach on the last part of the maintainability definition which is covering the evolution aspect.

Maintainability is strongly associated with the effort required to implement occurring change requests, which is why, for now, the KAMP approach is concerned with estimating maintenance efforts.

2.3 Change Requests

A change request is a particular situation when the software system needs to be modified. Since an architecture can not be arbitrarily flexible and implementing flexibility costs time and money, it is difficult to make a general statement about maintainability. Even patterns and anti-patterns are not clearly distinguishable on architecture level without relation to change requests. In order to reduce the effort for change requests that need to be implemented, one needs to anticipate which changes occur in the future. Overall our approach helps with estimating the efforts necessary for implementing anticipated change requests.

Within this paper we distinguish several *kinds of change requests*, based on their causes or stimuli. There are requirement changes regarding functionality, that request a new or altered functionality. Another stimulus is the evolution of the technical environment, which the software system depends on, e.g., changes in the platform (operating system, middleware). Another stimulus is the evolution of a COTS product, used by the software system, i.e., API changes of underlying libraries. Other stimuli arise from changed user profiles, (e.g., increased number of users, different usage behaviors), which require changes in order to fulfill non-functional aspects like performance, reliability, and security.

Besides the stimulus, the *effect of a change request* plays an important role in KAMP, since a change request needs to be translated into concrete work tasks. The effect is represented by all tasks and subtasks that lead to the fulfilment of the change request, including follow-up tasks due to change propagation. These tasks can affect various kinds of effort types. This comprises in the first place efforts for implementation (code changes), but also efforts for (re-)configuration, (re-)compilation, (re-)testing, (re-)deployment, data handling (modeling, conversion, migration), components-off-the-shelf (COTS) handling (survey, selection, tailoring, configuration, replacement), as well as efforts for retaining and increasing the internal code quality (refactoring, anti-pattern detection and removal).

3 The KAMP Approach

3.1 Overall

The Karlsruhe Architectural Maintainability Prediction (KAMP) approach aims to enable effort estimation for change requests based on architecture models.

Given a change request the approach derives from the architecture model a change description, i.e., work plan. This work plan contains the tasks, that are necessary for implementing the change request, coupled with tasks related to other activities like (re-)configuration, (re-)deployment of components, etc.

In order to get effort estimates, KAMP provides support to determine the complexity for each task in the work plan. A bottom-up estimation approach is used to map the complexity of each task to corresponding time effort. Overall KAMP combines a top-down derivation phase for creating the work plan with a bottom-up estimation phase. Note, in the current state of the approach the bottom-up estimates have to be provided manually by the user, whereas the workplan derivation is automated as is explained in the following sections.

The level of detail and granularity of the work plan tasks starts high, covering abstract tasks, and is then stepwise refined, by gathering additional information from the user and from architecture models, following a guided procedure. On the one hand the level of detail can be refined by going from the component level to the level of single service operations, while on the other hand the work plan description can be extended by following up on change activities that are detected using a semi-automated change impact analysis.

3.2 Inputs and outputs

KAMP takes as *inputs* 1) the description of the software architecture and 2) the description of the change request.

For the *description of the software architecture* the user creates an instance of Q-ImPrESS Service Architecture Meta-Model (SAMM) [2] that provides all elements of a component-based and service-oriented software architecture.

The architecture model can be created manually or, given certain conditions, retrieved automatically from source code by applying the Q-ImPrESS Reverse Engineering tool chain. A set of heuristics for detection of structural architecture parts, such as component boundaries and interfaces, as well as the statical analysable behaviour, is provided. In the project context of Q-ImPrESS these heuristics are applicable to Java and C/C++ code.

As an intermediate result of the reverse engineering, we obtain a Generalized Abstract Syntax Tree (GAST) and a mapping between the architecture model and the GAST. The GAST model can be used to calculate code and design metrics, which allows for an automatic determination of complexity metrics for corresponding architecture elements.

The description of a change request contains a name and an informal description of the stimulus, referring to the requirements that are affected by the change request. Moreover it covers the kind of stimulus (functional requirement change, technical environment change, COTS evolution, usage profile change).

The Q-ImPrESS SAMM allows for specifying alternative and sub-alternative models for various sequences of change requests, leading to a tree-like hierarchy of architectural models, each path within this hierarchy starting from the root model representing an evolution alternative of the software architecture. Each element in the tree represents a an architecture alternative, that consists of models for each supported architectural view, i.e., repository, system structure, behaviour, hardware environment, deployment, and quality annotations. Each subnode in the tree is basically a copy of its parent alternative with some modifications.

In order to specify how the change request is mapped to the architecture model, the user creates a sub-alternative of the actual system model and adapts the architecture model according to the change request.

The output of KAMP is a work plan, containing the change tasks, annotated with complexity values and effort estimates. Work plans can be compared, by comparing the structure or by comparing the aggregated complexity and effort values.

3.3 Work plan model

A work plan contains a list of activities or tasks. The types of activities are defined in a meta model. An activity refers to an architectural element and a basic activity.

The Q-ImPrESS Service Architecture Meta-Model (SAMM) [2] specifies the architecture elements like Component, Interface, Interface Port, Operation, Parameter, Datatype, etc.

Basic activities are **add**, **modify** and **remove**. **Add** means that the architecture element has to be newly implemented, **modify** means that the element has to be modified, and **remove** means that the element needs to be deleted from the code. For example the work plan can contain activities like ”Modify Component A”, ”Add Operation B”, or ”Remove Parameter C”.

Besides these implementation related work plan activities the work plan metamodel provides activities that cover other effort types related to configuration, testing and deployment. Provided activities are ”Modify configuration”, ”Run Tests”, ”Deploy components” and ”Update deployed

components".

3.4 Work plan derivation

The last section presented the ingredients of the work plan. Let us now have a look at how KAMP derives work plan instances. The work plan derivation in our approach is achieved in two ways. First, the work plan can be derived from changes in the architecture model, and second, the work plan can be derived by following a wizard dialog. The following paragraphs explain both ways in detail and compare them.

Derivation from architecture model changes The first way of work plan derivation is by calculating tasks from changes in the architecture model. In this way the user creates a copy of the architecture model and changes it according to the selected change request. KAMP calculates the differences from the changed architecture model to the base architecture model and translates the differences into work tasks. The work plan is filled with translated work tasks.

For example lets assume a client-server-database application. Now the software architect decides to introduce a cache between the server and the database. The architect creates a sub-alternative model and inserts the cache into the architecture model and fixes the interfaces and connectors using a model editor. Then, KAMP calculates the differences and creates a work plan containing an activity "Add Component Cache". Additionally the changes to interfaces and connectors are retrieved and represented by corresponding work plan activities.

Technically, the SAMM as well as the workplan metamodel are implemented using the EMF Ecore technology. Therefore we use EMFDiff to calculate a diff model between instances of SAMM. We defined a mapping between the elements of the diff model to corresponding workplan activities and wrote a transformation, that creates the workplan out of it.

Derivation by wizard dialog The second way of derivation is by following a wizard dialog. As the first step of this way the user determines the primary changes, i.e., architecture elements that need to be changed representing a starting point for the change. On the first wizard page the user marks the components that have to be added, modified or removed. On the second and third wizard pages the user refines this information to interface port and operation level, thus telling KAMP what interface ports or operations of the selected components need to be added, modified and removed.

For example, if a functionality in a user interface has to be modified, the user points out the components that build up the user interface and marks them with "modify". On the second and third wizard page the user marks the interface ports and operations of the user interface component that need to be modified.

Outgoing from this starting point the approach helps with identifying follow-up changes. For example, the components that are connected to the user interface components that implement the business logic of the functionality have to be changed due to the changes of the user interface components. Computing follow-up tasks is necessary because it ensures that all locations depending on a change, such as an interface signature or a behavior change, are changed consistently. If a change can not be kept locally it will propagate to other system parts.

The KAMP tool suite uses a wizard dialog to query the user to declare the primary changes and mark whether interface changes will propagate. The dialog guides the user stepwise through connected system parts to gather follow-up changes.

Let's briefly compare both derivation approaches. The benefit of using derivation from architecture model changes is that the user, i.e., software architect, can use a simple and familiar architecture model editor. However, there are changes that do not affect the architecture and that can not be derived by changing the architecture model. On the other hand the wizard dialog is something new to the user but can handle activities that are not visible by changing the architecture model. Therefore we recommend a hybrid usage of wizard guidance and architecture modeling.

3.5 Bottom-Up Effort Estimation

The work plan contains the split-up work activities. KAMP uses a bottom-up effort estimation approach for gathering the time effort estimates. In other words, the developers are asked to give time effort estimates for each work activity. KAMP aggregates all effort estimates to a single number at the work plan level.

The benefit of using bottom-up estimation is that a calibration of model parameters from historical data is not necessary since people consider their own productivity implicitly when giving estimates. Nevertheless, the approach is open to be connected to parametric estimation approaches such as Function Point or COCOMO.

4 Initial Case-Study

We implemented the approach as tool in the Eclipse environment and integrated it with the rest of Q-ImPrESS tool chain. In order to show the applicability of the approach we used KAMP on a case study. For this purpose we used the Enterprise SOA showcase, [10], that is one of the demonstration systems of the Q-ImPrESS project.

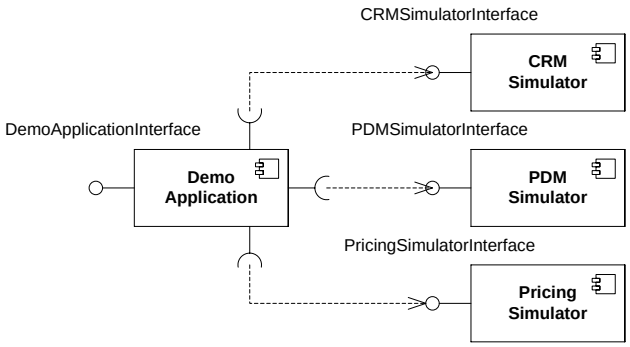

Figure 1. UML component diagram of Enterprise SOA showcase

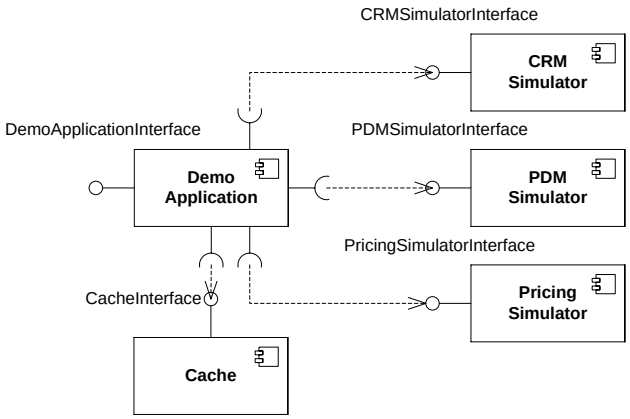

Figure 2. UML component diagram of Enterprise SOA showcase with cache

4.1 System description

The Enterprise SOA showcase consists of several small software systems implementing basic processes in the area of Supply Chain Management and Order Management. Its focus lies on the interaction between those software systems without providing full implementation of the various processes. Also components for simulating the usage of particular software systems are provided. Most of the systems consist of a database, a web front-end and web services for remote access.

The core systems of the Enterprise SOA showcase are CRM (Customer Relationship Management System), PDM (Product Data Management System), Pricing Engine and Inventory System. The simulation systems are Order Simulator, Shipment Simulator and Simulation Manager. Finally there is a Demo application for retrieving information from CRM, PDM and Pricing through Web Services.

A part of the system as UML component diagram is shown in Figure 1.

4.2 Architecture models

Our project partner, Itemis, that is responsible for the showcase, created an architecture model of the system. We refer to this model as the main alternative of the system. For the Enterprise SOA showcase we collected a set of change requests, which are anticipated during system evolution. For illustration purposes in this paper we selected one change request that is described in the following. For this change request a subalternative model has been created in the architecture model evolution hierarchy of the Q-ImPrESS tool chain.

4.3 Change request: Introduce Cache to Demo Application

Change request specification Due to performance issues with respect to the Demo application the following change request arises. A cache should be inserted. The Demo application manager should ask a cache for query results. Only in case of cache miss it should submit requests to the web services of the other subsystems CRM, PDM and Pricing.

Scenario modelling KAMP is utilized to determine a work plan for this change request. Therefore a subalternative architecture model is created and adapted according to the change request.

Here we list the steps done in the model editor: The cache component (Cache) is inserted into the repository. A cache interface (CacheInterface) with three operations (getQueryResult, putQueryResult, clear) for putting and getting of values and clearing the cache is specified in the repository. The cache component gets a provided interface port of type CacheInterface. A subcomponent instance of type Cache is created. A required interface port of type CacheInterface is added to the Demo application manager component. A connector is drawn that links provided and required interface ports of Cache component and Demo application manager component.

Besides the structural changes the architects adapt the dynamics. As a result the provided operation queryPrice of the Demo application manager is modified. The control flow is adapted by inserting a branch action to differentiate the cases of cache hit and cache miss.

An UML component diagram of the changed static structure is presented in Figure 2

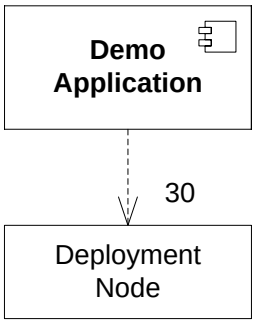

Figure 3. Deployment diagram for DemoApplication component. Component is deployed on 30 nodes.

Work plan derivation After creating the subalternative model that represents the target model after the change request is implemented the architect starts the KAMP derivation process. KAMP calculates a differences model between the mainalternative model files and the subalternative model files. The resulting work plan from the derivation process is shown in Listing 1.

Listing 1. Workplan for Change Request

```
Add InterfaceDefinition CacheInterface
  Add OperationDefinition getQueryResults
  Add OperationDefinition putQueryResults
  Add OperationDefinition clear
Add Component Cache
  Add Provided InterfaceImplementation
       CacheInterface
    Add OperationImplementation
         getQueryResults
    Add OperationImplementation
         putQueryResults
    Add OperationImplementation clear
Modify Component DemoApplication
  Modify Provided InterfaceImplementation
       DemoApplicationInterface
    Modify OperationImplementation queryPrice
```

Deriving deployment activities As can be seen in the work plan the component DemoApplication has to be modified. From the information present in the deployment view of the architecture model (see Figure 3) KAMP retrieves that this component is allocated to 30 nodes. Hence, KAMP adds a new activity to the work plan: Redeploy component DemoApplication (on 30 Nodes). As a result, the modification of a component leads to the follow-up effort for redeployment of the components.

Effort Estimation Our project partners annotated the work plan activities with time effort estimates in Person Days. The aggregated time efforts are then exported to the

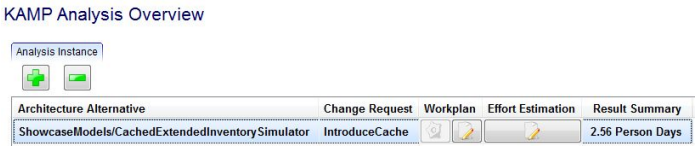

Figure 4. Result overview

Q-ImPrESS result model which can be used as input for trade-off analysis. A screenshot of the result overview is shown in Figure 4. The results are given in Person Days.

5 Related work

5.1 Scenario-Based Architecture Quality Analysis

In literature there are several approaches which analyze 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 ([1], [9]) which summarize and compare existing architecture evaluation methods.

Software Architecture Analysis Method (SAAM) [7] 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 extensibility, 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 be 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) [7] 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. Then, 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 [8].) Different to our approach, change effort is not measured as costs on ATAM.

The Architecture-Level Prediction of Software Maintenance (ALPSM) [4] 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) [5]
The ALMA method represents a scenario-based software architecture analysis technique specialized on modifiability and was created as a combination of the ALPSM approach [4] with [13]. 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.

5.2 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)* [11] and *Comprehensive Cost Model (COCOMO) II* [6]. 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 [14]
(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

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

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.

6 Conclusions

In this paper we presented the KAMP approach for estimating the evolution effort of a given change request based on the architectural model of a service-oriented software system. The main contributions of our method are:

- a way to map change requests to architecture models and derive a work plan by calculating differences between models, enhanced with user inputs from a wizard dialog and
- an integrated bottom-up estimation approach providing evolution effort estimations, which are not limited to implementation efforts only.

We showed the applicability of our approach by using it on the Enterprise SOA Showcase, an open-source industrial demonstration systems developed within the Q-ImPrESS project.

7 Acknowledgements

The work presented in this paper was funded within the Q-ImPrESS research project (FP7-215013) by the European Union under the Information and Communication Technologies priority of FP7.

References

[1] M. 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.

[2] S. Becker, L. Bulej, T. Bures, P. Hnetynka, L. Kapova, J. Kofron, H. Koziolek, J. Kraft, R. Mirandola, J. Stammel, G. Tamburelli, and M. Trifu. Q-ImPrESS Project Deliverable D2.1: Service Architecture Meta Model (SAMM). Technical Report 1.0, Q-ImPrESS consortium, September 2008. http://www.q-impress.eu/wordpress/wp-content/uploads/2009/05/d21-service_architecture_meta-model.pdf.

[3] S. Becker, M. Trifu, and R. Reussner. Towards Supporting Evolution of Service Oriented Architectures through Quality Impact Prediction. In *1st International Workshop on Automated engineeRing of Autonomous and run-time evolving Systems (ARAMIS 2008)*, September 2008.

[4] 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.

[5] 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.

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

[7] P. Clements, R. Kazman, and M. Klein. *Evaluating software architectures*. Addison-Wesley, 4. print. edition, 2005.

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

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

[10] C. Häcker, A. Baier, W. Safonov, J. Tysiak, and W. Frank. Q-ImPrESS Project Deliverable D8.6 Enterprise SOA Showcase initial version. Technical Report 1.0, Q-ImPrESS consortium, January 2009.

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

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

[13] N. Lassing, D. Rijsenbrij, and H. van Vliet. Towards a broader view on software architecture analysis of flexibility. *Software Engineering Conference, 1999. (APSEC '99) Proceedings. Sixth Asia Pacific*, pages 238–245, 1999.

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