Architecture-based Assessment and Planning of Change Requests

Kiana Rostami
Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5
76131 Karlsruhe, Germany
rostami@kit.edu

Johannes Stammel
andrena objects ag
Albert-Nestler-Str. 9
76131 Karlsruhe, Germany
johannes.stammel@andrena.de

Robert Heinrich
Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5
76131 Karlsruhe, Germany
heinrich@kit.edu

Ralf Reussner
Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5
76131 Karlsruhe, Germany
reussner@kit.edu

ABSTRACT

Software architecture reflects important decisions on structure, used technology and resources. Architecture decisions influence to a large extent requirements on software quality. During software evolution change requests have to be implemented in a way that the software maintains its quality, as various potential implementations of a specific change request influence the quality properties differently. Software development processes involve various organisational and technical roles. Thus, for sound decision making it is important to understand the consequences of the decisions on the various software engineering artefacts (e.g. architecture, code, test cases, build, or deployments) when analysing the impact of a change request. However, existing approaches do not use sufficient architecture descriptions or are limited to software development without taking management tasks into account. In this paper, we present the tool-supported approach Karlsruhe Architectural Maintainability Prediction (KAMP) to analyse the change propagation caused by a change request in a software system based on the architecture model. Using context information annotated on the architecture KAMP enables project members to assess the effects of a change request on various technical and organisational artefacts and tasks during software life cycle. We evaluate KAMP in an empirical study, which showed that it improves scalability of analysis for information systems due to automatically generated task lists containing more complete and precise context annotations than manually created ones.

QoSA'15, May 04-08, 2015, Montreal, QC, Canada

http://dx.doi.org/10.1145/2737182.2737198.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.9 [**Software Engineering**]: Management; D.2.11 [**Software Engineering**]: Software Architecture

General Terms

Software Architecture, Maintainability, Change Propagation

Keywords

Software Evolution; Change Request; Impact Analysis

1. INTRODUCTION

The architecture model is one of the main artefacts of a software system and is central to many organisational tasks, such as reuse decisions, staffing, and effort estimation. In addition, the software architecture deals with entities such as components and interfaces which also play a key role in other software artefacts (e.g. object oriented designs, deployment diagrams, or test cases). A software system has to change during its life cycle to reflect the changes in its environment, such as requirements, technology, and usage profile. This process is known as software evolution [18]. The maintainability of software refers to efforts to implement changes during software evolution. We investigate maintainability with regard to a set of anticipated change requests.

Coordinating and implementing change requests are difficult tasks, as the software development process involves various organisational and technical roles. Usually a change request can be implemented in various ways. In addition, different ways of realising a change request lead to different efforts in adapting code and other artefacts. Furthermore, there are strong interdependencies between organisational and technical issues. In a real-world software system different people own knowledge and responsibility for various technical and organisational work areas, such as design, implementation, configuration, building, testing, rollout, deployment, and all tasks involving management and organisational operations. Considering artefacts from the perspective of various

roles, such as designer, implementer, tester, deployer, and operator helps software architects to understand the impact of changes. The main idea of the presented approaches is based on two observations: 1) All these different roles use artefacts, which are tightly related to the software architecture. 2) Although the maintenance effort of the software architecture alone is often rather small, reflecting changes in software architecture models helps to identify tasks of maintaining other artefacts.

Automation helps software architects to calculate change requests by answering the following questions: 1) How can a change request represented in the architecture model be implemented with respect to various technical and organisational work areas? 2) What are the technical and organisational impacts of a change implementation? 3) Which artefacts are affected by a change request during software life cycle? 4) How can concurrent change requests be coordinated and implemented?

Existing approaches often do not have sufficient architecture descriptions or are limited to software development without taking management tasks into account. Work on task-based project planning (e.g. [16, 7, 4]) uses the architecture only in a very coarse-grained manner, if at all. Also approaches to architecture-based project planning (e.g. [20, 5]) and architecture-based software evolution (e.g. [12, 19]) neither offer automated change impact analysis nor support derivation of change activities or change effort estimation based on a given architecture. Scenario-based architecture analysis approaches (e.g. [9, 3, 20]) that focus on software development activities and artefacts, lack a formalized architecture model and seldom come along with tool-support.

This paper presents a tool-supported approach, *Karlsruhe Architectural Maintainability Prediction (KAMP)*, to analyse the change propagation caused by a change request in the software architecture model. A preliminary version of KAMP's idea was published in [21, 22]. In this paper, we extend and refine KAMP, give a formal definition of the approach and evaluate it in an empirical study. KAMP is novel with respect to the fact, that it is not only concerned with the design phase of software, but basically with all software life-cycle phases, including testing, deployment and operation. It enables project members to assess the effect of change requests on technical and organisational work areas during software life cycle. KAMP enables software architects to model the initial architecture with annotated context information involving technical and organisational tasks, called base architecture, and the architecture after the change request is implemented, called target architecture. Then, the tool calculates the differences between the base and the target architecture model and generates task lists by applying derivation and interpretation rules on the architecture model. In addition to the structural propagation of changes in the architecture, KAMP computes impacts on all major work areas involving tasks such as test development and execution, build configuration, deployment and their corresponding artefacts which need to be processed during software life-cycle.

We conduct an empirical study to validate our approach. KAMP is implemented as an extension of the Palladio Component Model (PCM) [1] in order to model the software architecture. PCM enables meta-modelling a component-based system with regard to performance relevant aspects. We demonstrate how KAMP can help semi-automatically derive task lists for a specified change request and thus improves the scalability of analysis. KAMP produces higher quality and more homogeneous task lists compared to manual analysis. The study shows that the semi-automatically generated task lists for the software architecture presented in Fig. 1 are more complete and more precise than manual ones. While the quality of manual task lists depends on the experience of the user, the semi-automatically generated task lists of various users are similar.

The reminder of this paper is organised as follows: In Sec. 2, we introduce a running example to demonstrate our approach throughout this paper. An overview of KAMP is given in Sec. 3, the formalisation of KAMP is presented in Sec. 4. Sec. 5 summarizes related work. We present the evaluation of our approach using an empirical study in Sec. 6. Sec. 7 concludes the paper and proposes future work.

2. RUNNING EXAMPLE

In this section, we introduce a user management system which acts as a running example for demonstration and validation purposes in the following sections. Whenever we are referring to the running example, the paragraph is marked with a black bar at the left hand side.

2.1 System Overview

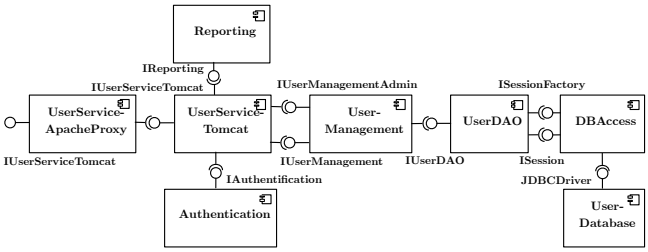

Figure 1: Component Diagram of User Management System

Fig. 1 shows a component-based architecture model of the system and the corresponding interfaces. `User-Database` stores user data. `DBAccess` performs object-relational mappings. `UserDAO` provides access to persisted user data. `UserManagement` encapsulates the business logic of user management. `UserServiceTomcat` represents a service which is deployed in a web server container. It provides a *REpresentational State Transfer (REST)* interface to users, which uses *JavaScript Object Notation (JSON)* data format. `UserServiceApache-Proxy` deals with port switches and forwards calls to `UserServiceTomcat`. `Reporting` collects reports about transactions. `Authentication` is a third-party service for authentication. Additionally, there are seven data types in the system: `User`, `UserList`, `Report`, `SessionFactory`, `Session`, `Token`, and `WebServiceResponse`.

2.2 Change Scenario

In order to demonstrate KAMP we investigate a change request involving the `User` data type. In our system a `User` is represented by several attributes (e.g. first name, surname). We extend `User` data type to include an additional `postalCode` attribute.

3. KAMP ANALYSIS PROCESS

In this section, we present the phases of KAMP analysis process and apply it to the running example and the change scenario. As illustrated in Fig. 2, KAMP consists of the *Preparation Phase* and the *Change Request Analysis Phase*.

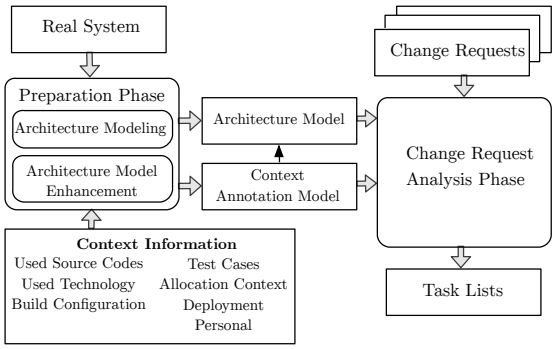

Figure 2: Overview of KAMP

In the *Preparation Phase*, the user models the architecture of an existing software using the extended PCM and then annotates context information, involving source code files, configuration files, technology stack, test cases, build configurations, release configurations, and deployed instances.

In the running example, a user applies the extended PCM to create an architecture model, involving all data types, interfaces and components in a repository, similar to Fig. 1. Moreover, the user annotates the architecture model with the following context information : 1) *Java Source Files* for each data type, interface, and component, 2) two *Hibernate Configuration Files* for object-relational mapping component, 3) one *Build Configuration* annotation, called "Eclipse project properties" attached to the following components: `UserServiceApacheProxy`, `UserServiceTomcat`, `UserManagement`, `Reporting`, `UserDAO`, `DatabaseAccess`, and `UserDatabase`, 4) 7, 5, and 1 unit tests for provided interface of `UserServiceTomcat`, `UserManagement`, and `Authentification` components, respectively, which are annotated on the architecture model using *Test Case* annotations, 5) one instance of each component is deployed on a single server, 6) `DBAccess` as a *Third Party* component, 7) `Authentication` as an *External* component. The user models a pessimistic intra-component dependency, which specifies, that every provided interface of each component depends on all required interfaces of that component.

The *Preparation Phase* results in an architecture model and a context annotation model, which serve as inputs for the *Change Request Analysis Phase*. For each change request this phase is done. In this phase the user modifies the architecture model to reflect the target model and semi-automatically calculates change propagation and derives the corresponding work plan. The work plan contains required tasks to implement the change request and the corresponding software artefacts, which have to be altered. This phase comprises the following six steps:

1) Creating target model: The KAMP creates a cloned version of base architecture model and context annotation model serving as the initial version of the target model.

In the running example, the user triggers the KAMP tool to create a copy of the user management system architecture and annotation models from previous phase.

2) Modelling the change request: As a change request can be implemented in different ways, the user modifies the cloned architecture model to reflect the change request. This process results in a target architecture model. If the modifications are not visible to the elements of the architecture model, such as architectural structure or signature of interface methods, the user can add special annotations to mark modified elements.

In the running example, the user marks the `user` data type of the `UserDatabase` as modified.

3) Calculating structural change propagation: KAMP automatically calculates the expected structural change propagation. We consider several kinds of structural propagations, which can be summarized in propagation from data types to interfaces, propagation from interfaces to components, horizontal propagation, involving the inter-component and intra-component propagation, and vertical propagation, involving change propagation through composite-components.

The user triggers the tool to calculate the change propagation which leads to insertion of modification marks into the extended architecture model. KAMP iteratively marks all components affected by a change request. In the first sub-iteration of each iteration, it analyses the inter-component propagation via the connectors to other components (e.g. the change propagation from `UserDatabase` to `DBAccess`). In the next sub-iteration, it considers, how a change to a required interface propagates through the component (e.g. the change propagation through predefined dependencies in `DBAccess`).

4) Updating change-specific context annotations: At this point, the user has to extend and refine the context annotations, as automatically calculating change requests can result in an inconsistent state. For example, if a change request results in removing an element of the architecture model, there could be context annotations without any corresponding architecture element. Furthermore, if a change request causes adding a new architecture element, KAMP automatically suggests the following context annotations: 1) the number and the types of test cases required based on size of the interface and the new added element, 2) mapping the new element to a build configuration, 3) technology stack based on the level of hierarchy, 4) the roles and responsibilities for the new added element. Otherwise, if context annotations are incomplete or inconsistent, KAMP prompts the users to manually correct information. If a change request causes removing an architectural element, KAMP automatically suggests the following steps: 1) removing the invalid context annotations, 2) assigning the invalid context annotations to other elements.

In the example, the user does not need to add or update any elements, as only one data type is modified.

5) Deriving architecture-based task list: At this point, KAMP compares base and target architecture models and finds differences between them. Then, it derives a set of architecture-based tasks with respect to added, removed or modified elements and then automatically derives an architecture-based task list.

In the example, the derived task list is similar to the tasks marked with **(5)** in Tab. 1. The table contains tasks resulting from fifth and sixth step. In the current step, only tasks indicated by **(5)** are produced which indicate architecture-level tasks. Furthermore, Tab. 1 exemplarily shows 2 of 5 iterations of the change propagation analysis phase. Sub-tasks are marked with ≡. This step is based on the earlier phases. Thus, each iteration comprises two sub-iteration involving the inter-component and the intra-component change propagation.

6) Deriving task list with respect to work areas: Based on the architectural tasks, KAMP derives work-area-oriented tasks (i.e. editing source code files, editing meta-data such as database schema, and follow-up tasks involving all tasks needed in order to have a running system).

Tab. 1 shows derived tasks required to implement the change request, which are indicated by **(6)**. These tasks are work-area specific.

Modify Data Type `User` (5)
Modify Interface `JDBCDriver` (5)
Modify Component `UserDatabase` (5) ≡ Modify Provided Interface `JDBCDriver` (5) → Rebuild `UserDatabase` (6) → Redeploy `UserDatabase` (6)
1. Iteration: Inter Component Propagation (5) Modify Connector `DBAccess` → `UserDatabase` (5) **Intra Component Propagation** (5) Modify Component `DBAccess` (5) → Modify configuration files (6) ≡ Modify Required Interface `JDBCDriver` (5) ≡ Modify Provided Interface `ISession` (5) ≡ Modify Provided Interface `ISessionFactory` (5) → Rebuild `DBAccess` (6) → Redeploy `DBAccess` (6)
...
5. Iteration: Inter Component Propagation (5) Modify Connector `UserServiceApache` → `UserServiceTomcat` (5) ≡ Modify Required Interface `IUserServiceTomcat` (5) **Intra Component Propagation** (5) Modify Component `UserServiceApache` (5) → Modify Source Files (6) → Modify Test Cases (6) → Execute Test Cases (6) ≡ Modify Provided Interface `IUserServiceTomcat` (5) → Rebuild `UserServiceApache` (6) → Redeploy `UserServiceApache` (6)

Table 1: Task list for the change scenario

4. FORMALISATION

In this section, we present a formalisation of our approach. Using meta-models and graph representations we define inputs and results of KAMP. Furthermore, we describe our approach as a combination of graph algorithms and model transformations. KAMP uses two input data models involving both *architecture model* and *context annotated model* and one result data model specifying task lists in an *task model*. An overview of meta-models and graph definitions required are given in Sec. 4.1. In Sec. 4.2, we describe meta-model transformations and graph operations needed in the preparation phase. Finally, we present in Sec. 4.3 the change request analysis phase and characterize some graph operations.

4.1 Meta-models and Graph Definitions

Following we describe the meta-model and graph definition of the architecture model in Sec. 4.1.1, the context annotated model in Sec. 4.1.2, and the task model in Sec. 4.1.3.

4.1.1 Architecture Model

In the first sub-phase of the preparation phase, the user provides a component-based model of the software architecture as described hereafter.

Meta-model: To model the architecture we use the PCM meta-model, which is described in detail in [1]. In the following we discuss a couple of extensions to PCM meta-model required for KAMP. In order to analyse the change propagation through a component, knowledge about the component intra-dependencies is required. Thus, we extend PCM to include a dependency model element, known as `ComponentIntraDependency`, which defines the dependency between provided and required interfaces of a component. To enable users to indicate changes, which are invisible to an architecture model, we extend PCM to include the meta-class `ModificationMark`. Thus, the model supports modification annotation for data types, interfaces, components, required interfaces, provided interfaces and assembly connectors.

Graph: From the above meta-model extension, we define the architecture graph $G_{architecture}$ to represent the static software architecture. A Graph $G_{architecture} = (V_{architecture}, E_{architecture})$ comprises a set of vertices, which represent the static elements of an architecture, and a set of edges, which represent links connecting the architecture elements. In the following we give an overview of properties of $V_{architecture}$ and $E_{architecture}$:

- *Properties of Vertices*: The vertices have a type *v.type*, a unique identifier *v.id*, and a name *v.name*. They satisfy $\forall v \in V_{architecture} : v.type \in VT_{architecture}$, where $VT_{architecture}$ is the set of vertex types in the architecture graph, containing `primitiveDataType`, `compositeDataType`, `operation`, `interface`, `providedInterface`, `system`, `basicComponent`, `compositeComponent`, and other types of vertices.

- *Properties of Edges*: In contrast to vertices, the edges have only a type *e.type* and satisfy $\forall e \in E_{architecture} : e.type \in ET_{architecture}$, where $ET_{architecture}$ is the set of edge types in the architecture graph, containing `operationDefinition`, `parameterDataTypeDefinition`, `providedInterface`, `interfaceDefinitionForProvidedInterface`, and other types of edges. For example, `providedInterface` as a type of edge, that connects a component vertex with a required interface vertex, is formally defined as:
  $\forall e : (v_L, v_R) \in E_{architecture}$:
  $e.type = providedInterface \Rightarrow v_L.type \in \{basicComponent, compositeComponent, system\}$
  $\bigwedge v_R.type = providedInterface$

4.1.2 Context Annotated Model

In the second sub-phase of the preparation phase the user annotates the architecture model with additional information, as presented hereafter.

Meta-model: In order to meta-model the annotation, we extend PCM to include the meta-class `ArchitectureModelEnhancement`, which has the following subclasses: `SourceCodeFile`, `SourceCodeFileAggregation`, `MetaDataFile`, `Meta-`

DataFileAggregation, BuildConfiguration, TestCase, TestCaseAggregation, DeliveryConfiguration, DeploymentConfiguration, RuntimeInstance, RuntimeInstanceAggregation, and TechnologySpecification. Each subclass defines a specific technical or organisational work area to be considered when analysing change propagation. This extension enables annotating either a component or a provided interface.

Graph: Based on the meta-model, we define a context annotated graph $G_{enhancement} = (V_{enhancement}, E_{enhancement})$ comprising a set Vertices $V_{enhancement}$ and a set of edges $E_{enhancement}$. In the following we give an overview of properties of $V_{enhancement}$ and $E_{enhancement}$:

- *Properties of Vertices*: The vertices of a $G_{enhancement}$ have two properties: a type *v.type* and an optional name *v.name*. They satisfy $\forall v \in V_{enhancement} : v.type \in VT_{enhancement}$, where $VT_{enhancement}$ is a set of all possible types of vertices in a context annotated graph. Each type of vertices has a corresponding subclass of `architectureModelEnhancement`.
- *Properties of Edges*: Since the context annotated graph would be used to annotate the architecture graph, the set of edges has to be defined generically:
  $\forall e : (v_L, v_R) \in E_{enhancement} : e.v_L \in V_{enhancement}$
  $\bigwedge e.v_R \in V_{architecture}$
  $\bigwedge e.v_R.type \in \{$ `basicComponent`, `compositeComponent`, `system`, `providedInterface` $\}$

4.1.3 Task Model

The Change Request Analysis Phase returns work plans in terms of tasks to realise a certain change request.

Meta-model: The root meta-class of model is the `WorkPlan`, containing abstract class `Task`. The tasks can be ordered and can be nested within each other. Its subclasses are `ArchitectureOrientedTask` and `WorkAreaOrientedTask`. `ArchitectureOrientedTask` states the modification of the static elements of the architecture, such as data types, components, interfaces, provided and required interfaces and their operations, and connectors. We define further operations for each class of tasks. For example, `ProvidedInterface` has the following subclasses: `AddingProvidedInterface`, `RemovingProvidedInterface`, and `ModifyingProvidedInterface`. `WorkAreaOrientedTask` states sequences in each work area, such as coding, building, testing, and deploying. Each metaclass has further subclasses. For example, `Testing` consists of `DevelopingTests` and `ExecutingTests`.

Graph: Based on the meta-model of `task`, we define the task graph $G_{task} = (V_{task}, E_{task})$ comprising a set of vertices V_{task} and a set of edges E_{task}. The nodes of this graph have a type, *v.type*, and a name, *v.name*. They satisfy: $\forall v \in V_{task} : v.type \in VT_{architectureOrientedTask} \bigvee v.type \in VT_{workAreaOrientedTask}$, where $VT_{architectureOrientedTask}$ and $VT_{workAreaOrientedTask}$ are the set of vertex types of the task graph. $VT_{architectureOrientedTask}$ involves `addingDatatype`, `addingInterface`, `removingInterface`, `addingProvidedInterface`, and other architecture-oriented tasks, whereas $VT_{workAreaOrientedTask}$ involves `editingSourceCode`, `developingTestCases`, `deploymentConfiguration` and other task-oriented tasks.

4.2 Preparation Phase

From graph definition in the previous section we describe the graph operations for each sub-phase of preparation phase:

4.2.1 Architecture Modelling

In the first sub-phase we identify graph operations to create an architecture graph $G_{architecture}$. Examples of operations on graphs are adding a new basic component, removing a data type parameter from a composite data type, adding a provided interface delegation to a composite component, and adding a new provided interface to an existing component. The latter is described in Algorithm 1.

Algorithm 1 Adding a Provided Interface to a Component

1: create node v_{new}
2: set $v_{new}.type =$ `providedInterface`
3: add v_{new} to $V_{architecture}$
4: create a new `providedInterface` edge between the `providedInterface` node and the `component` node.
5: create a new `interfaceDefinitionForProvidedInterface` edge between the `providedInterface` node and the `interface` node.
6: **return** updated $V_{architecture}$ and $E_{architecture}$

4.2.2 Architecture Model Enhancement

In the second sub-phase we identify graph operations to enhance the architecture graph. The operations required for the resulting graph $G_{enhancement}$ include specifying metadata, build configuration, test cases, and source code. For example, Algorithm 2 describes the latter operation, which enables users to create source code nodes to represent the source code files of a component.

Algorithm 2 Specifying Source Code Files

1: create node $v_{sourceCodeFile}$
2: set $v_{sourceCodeFile}.type =$ `sourceCodeFile`
3: set $v_{sourceCodeFile}.name = \alpha$
4: add $v_{sourceCodeFile}$ to $V_{enhancement}$
5: create edge $e_{annotation}$
6: set $e_{annotation}.v_L = v_{sourceCodeFile}$
7: set $e_{annotation}.v_R = v_{component}$
8: add $e_{annotation}$ to $E_{enhancement}$
9: **return** updated $V_{enhancement}$ and $E_{enhancement}$

4.3 Change Request Analysis Phase

We describe the operations needed for the analysis phase of KAMP, comprising the following sub-phases:

Versioning Initial Architecture: A copy of both the architecture model $G_{architecture}$ and context annotated model $G_{enhancement}$ is set up, referred to $G'_{architecture}$ and $G'_{enhancement}$ respectively.

Modelling the Changes: The user changes $G'_{architecture}$ to model the architecture, after all changes are implemented.

Analysing Change Propagation: KAMP considers the structural propagation of changes within each component and between components, as shown in Algorithm 3. In KAMP, components, interfaces, and data types are characterized as first class entities, which can thus be marked as modified.

Extending and updating additional information: After $G'_{architecture}$ is modified, the additional information may not be valid. Thus, the additional information must be extended and updated to reflect the changes.

Comparing the architectures to consider architecture-oriented tasks: In this step, KAMP compares $G_{architecture}$ and $G'_{architecture}$ and find differences between the architectures, since the nodes and edges in $G_{architecture}$ and corresponding elements in $G'_{architecture}$ have the same identifier. For example, tasks to add a new provided interface `AddingProvidedInterface` are given by the following set:

Algorithm 3 Change Propagation Analysis Algorithm

1: **Require**: modified $G'_{architecture}$

1: Calculate change propagation from data types to interfaces

2: Identify modified data types
3: Determine interfaces invoking modified data types in calling parameters or in return types. Mark them as modified.
4: Get temporary results. Allow users to correct them (optional)

2: Calculate change propagation from interfaces to required and provided interfaces

5: Identify modified interfaces
6: Determine the corresponding required and provided interfaces to the modified interface. Mark them as modified.
7: Get temporary results. Allow users to correct them (optional)

3: Calculate the inter- and intra-component propagation

8: **while** There is at least one new modification **do**
9: Calculate the inter-component propagation
10: Identify modified provided interfaces, involving any provided interfaces, considered as modified and any provided interfaces with modified operations.
11: Determine their assembly connector. Mark them as modified.
12: Get temporary results. Allow users to correct them (optional).
13: Calculate the intra-component propagation
14: Identify modified required interfaces, involving any required interfaces, which is linked to an assembly connector, considered as modified.
15: Basic components: Determine component intra-dependency. Mark connected provided interfaces as modified.
16: Composite component: Follow the delegation connector to determine the affected sub-components. Mark them as modified.
17: Get temporary results. Allow users to correct them (optional)
18: **end while**

$V_{AddingProvidedInterface} =$
$\{v' | v' \in V_{added} \wedge v'.type = providedInterface\}$

Determining work-area-oriented tasks: In order to derive task lists in each technical and organisational work area from the architecture-oriented tasks we use model transformation. To this end, the deriving rules can be considered in terms of triple graph grammars [15]. For example, a transformation rule considers, how component tasks for a component, which is annotated with source code files result in a task list involving editing source code files.

5. RELATED WORK

Work related to KAMP comprises task-based project planning (Sec. 5.1), architecture-based project planning (Sec. 5.2), architecture-based software evolution (Sec. 5.3), and scenario-based architecture analysis (Sec. 5.4), as discussed hereafter.

5.1 Task-based Project Planning

Hierarchical Task Analysis (HTA) [16] is a method to decompose a high-level task into a hierarchy of subtasks in a systematic and structured fashion. A similar goal is addressed by the Goals, Operators, Methods, and Selection rules model (GOMS) [7]. The Keystroke level method [6] calculates execution time for an entire task by summing up the estimated times of the individual actions. Function Point Analysis (FPA) [11] estimates the size of a system by adding up the number and weights of all transaction and data elements. The Comprehensive Cost Model (COCOMO) II [4] includes different approaches for cost estimation during requirements phase and architectural design phase by applying the abstract measure of function points (applications points in COCOMO) based on an informal requirements description. However, if used at all, these techniques use the software architecture only in a very coarse-grained manner. Thus, it is hard to make accurate predictions using task-based approaches.

5.2 Architecture-based Project Planning

Conway's Law [10] links software design to project organisation by stating that the software architecture must reflect an organisation's communication structures. Architecture-Centered Software Project Planning (ACSPP) [20] considers a software architecture as an artefact in project planning. The goal of ACSPP is to estimate realistic schedules by combining top-down and bottom-up effort estimation techniques. Based on experience in mechanical engineering, Carbon [5] proposes a procedure to align product design and production planning in a software development context. The procedure allows for 1) early identification of potential problems in production and 2) assessing the software architecture for completeness. Work on architecture-based project planning, however, does not support estimating change efforts based on a given architecture and does not offer automated change impact analysis and derivation of change activities.

5.3 Architecture-based Software Evolution

Based on the assumption that software evolution follows certain common patterns, Garlan et al. [12] proposed an approach to assist for expressing architectural evolution and for reasoning about the correctness and quality of evolution paths. Naab [19] presents an approach to flexibility analysis and thus the maintainability of a software architecture at design-time, however, neglects operation and management tasks related to the architecture. Again, work on architecture-based software evolution does not support change efforts estimation and automated change impact analysis.

5.4 Scenario-based Architecture Analysis

Changes to requirements are triggers for changes in the system but drawing inference from the extent of changes in requirements about efforts for implementing the changes is not possible without considering the system's architecture. Some existing approaches target at scenario-based software architecture analysis but lack a formalised architecture description or are limited to software development without taking management tasks into account. Software Architecture Analysis Method (SAAM) [9] evaluates software architectures regarding modifiability by using an informal architecture description (mainly the structural view). SAAM gathers change scenarios and tries to find interrelated scenarios. Components affected by interrelated scenarios are identified and costs are estimated. In contrast to SAAM, the Architecture Trade-Off Analysis Method (ATAM) [9] aims at identifying trade-offs between various quality aspects by considering the effect of architectural decisions. Architecture-Level Prediction of Software Maintenance (ALPSM) [2] defines and weights scenarios to evaluate their impact of the overall maintenance effort based on component size estimations (LOC). ALPSM heavily depends on the expertise of the architects and provides little

guidance through tool support. Architecture-Level Modifiability Analysis (ALMA) [3] is a combination of ALPSM and the approach by Lassing et al. [17] to consider ripple effects by taking into account expert knowledge. A disadvantage is that ALMA does not take into account software management activities. Architecture-Centric Project Management (ACPM) [20] applies software architecture as the central artefact for planning and management activities. For architecture-based cost estimation, the architecture is used to decompose planned software changes into various tasks to realise the changes. For each task the assigned developer is asked to estimate the effort of realising the change. KAMP goes beyond ACPM by using formalised architectural models where ACPM uses only the structural view of architecture and therefore does not consider management costs. KAMP combines several strengths of existing approaches. It makes explicit use of formal software architecture models, provides guidance and automation via tool support, and considers management as well as development effort. Applications of KAMP for deriving work plans to solve performance issues [13] and for automated software project planning [14] has already been proposed.

6. EVALUATION

KAMP enables users to describe how a change request can be implemented in a software system. To this end, the user alters the base architecture model into a target architecture model to reflect the change implementation. Then, KAMP automatically calculates modifications, determines structural change propagation and mines annotated context information in order to derive task lists with respect to multiple technical and organisational work areas, which are represented by context information.

The main contribution of our approach is the automatic evaluation of context information and identification of tasks required to implement a change request. The expected benefits of automation are high scalability and high quality of analysis results. In other words, we expect that a tool-supported approach provides precise and complete task lists. Furthermore, we expect, that KAMP reduces analysis overhead, as the number of potential implementation of a specific change request is increased or as software systems and models grow. Moreover, we expect that using KAMP can help users to derive task list **regardless of their skill and experience**.

Scalability is a generally accepted property of automation approaches. However, our main focus of validation lies on the quality of analysis results. By quality we regard precision and completeness of task types and the corresponding artefacts. We investigate scalability implicitly by comparing the amount of time required for tool-supported analysis with manual analysis.

We validated the quality of analysis results in an empirical study. We compared analysis results obtained from our tool-supported approach with manually created task lists. We are especially interested in results determined by less-experienced users. Therefore, we integrate our empirical study into a study course at university.

6.1 Experiment Design

In the empirical study, we use the user management system introduced in Sec. 2. The participants comprise both less-experienced users and experienced users. While the less-experienced users are divided into a Treatment Group (TG) and a Control Group (CG), the EXpert Group (EXG) comprises experienced users. The treatment group applies the tool-supported KAMP approach, whereas the control group and expert group analyse the change propagation manually. Following, we describe the experiment's goals, questions, metrics and hypothesis using the GQM plan in Tab. 2.

Goal	**Empirically evaluate the differences between automatically derived task lists and manually created ones**	
Question 1	How precise and complete are the treatment group's (TG's) task lists compared with control group's (CG's) task lists?	
Question 1.1	... with regard to task types (TT)?	
Metric 1.1.1	F_1 score for TG	F_1(TT, TG)
Metric 1.1.2	F_1 score for CG	F_1(TT, CG)
Hypothesis	Automatic results are as good or better	F_1(TT, TG)$\geq$ F_1(TT, CG)
Question 1.2	... with regard to task annotations (TA)?	
Metric 1.2.1	F_1 score for TG	F_1(TA, TG)
Metric 1.2.2	F_1 score for CG	F_1(TA, CG)
Hypothesis	Automatic results are as good or better	F_1(TA, TG)$\geq$ F_1(TA, CG)
Question 2	How precise and complete are the TG's task lists compared with expert group's (EXG's) task lists?	
Question 2.1	... with regard to task types (TT)?	
Metric 2.1.1	F_1 score for TG	F_1(TT, TG)
Metric 2.1.2	F_1 score for EXG	F_1(TT, EXG)
Hypothesis	Automatic results are as good or better	F_1(TT, TG)$\geq$ F_1(TT, EXG)
Question 2.2	... with regard to task annotations (TA)?	
Metric 2.2.1	F_1 score for TG	F_1(TA, TG)
Metric 2.2.2	F_1 score for EXG	F_1(TA, EXG)
Hypothesis	Automatic results are as good or better	F_1(TA, TG)$\geq$ F_1(TA, EXG)

Table 2: GQM plan overview

The overall goal is to compare tool-derived results with manually determined results. We investigate this by asking two research questions. The higher level questions (i.e. Question 1 and Question 2) are concerned with user experience. While Question 1 compares two groups of less-experienced users (i.e. TG and CG), Question 2 compares the less-experienced user group with the experienced group (i.e. TG and EXG).

Each of these questions comprises two sub-questions (i.e. Question 1.1, 1.2, 2.1, 2.2) concerning information details regarding task types vs. detailed task annotations. Questions 1.1 and 2.1 look at the task types which the participants identify (e.g. editing source file or test development). Questions 1.2 and 2.2 determine, whether the participants correctly consider all task annotations for each task (e.g. the number of test cases).

The questions are quantified by metrics. In our experiment we use F_1 score (i.e. harmonic mean) to aggregate recall and precision of task types and task annotations. We compare results of all participants with a reference solution (i.e. benchmark). This comparison leads to two types of errors: 1) **False negative** is when the participant misses an element of the reference solution. 2) **False positive** is when an element is considered, which is not element of the reference solution. Given the number of correctly considered elements,

t_p, the number of false negatives, f_n, and the number of false positives, f_p, precision and recall can be calculated as follows: $precision = \frac{t_p}{t_p+f_p}$, $recall = \frac{t_p}{t_p+f_n}$. F_1 score is the harmonic mean of the both metrics: $F_1 = 2\,\frac{precision \times recall}{precision+recall}$.

In short, the GQM plan covers the following metrics. Questions 1.1 and 2.1: F_1-score for recall and precision of task types. Questions 1.2 and 2.2: F_1-score for recall and precision of task annotations.

Group Setup

We compare the results of a treatment group, comprising 6 computer science students, a control group, comprising 8 computer science students, and an expert group, comprising 5 researchers with master or PhD degree in computer science. While the treatment group used KAMP to derive the task lists and identify affected artefacts, the control and the expert group realised the same analysis manually. The 14 computer science students were participants of a practical course, in which they had to develop a distributed software system for mobile devices and servers. They learned all stages of a software development process. During the practical course we observed different student progresses. In order to divide the participants into two homogeneous groups, we used a stratified sampling method. The better students were assigned to the first stratum, whereas the other students were assigned to the second group. In the next step, we distributed each stratum randomly and equally to treatment group and control group.

Preparations

In order to prepare student participants for the experiment and establish equal conditions they were trained in software architecture knowledge and modelling. We assumed expert participants to be familiar with software architecture knowledge and modelling. Moreover, we trained the treatment group in KAMP tool application.

Experiment Material

Tab. 3 summarizes the experiment materials. As seen, all groups received a textual system specification with additional information and a PCM architecture model, while the treatment group was given the KAMP analysis tool and the architecture model together with the context annotation model. The context annotation model covered annotations as described in third paragraph of Section 3. The tool was accompanied by a usage guide. All participants got four task descriptions with detailed work instructions. This included an overview of task types and task list examples. The experiment consisted of three stages: the warm-up stage, the analysis stage with the analysis tasks, and the after-glow stage. Each stage was accompanied by a questionnaire: 1) The **warm-up questionnaire** helped participants get familiar with the software system and its environment. 2) **Analysis questionnaires**, either for tool results or for manual results. 3) **Post-experimental questionnaires** provides us with participant feedback on the experiment and a self-assessment.

Experiment Tasks – Change Requests

The experiment comprises four change requests in the user management system, referred to tasks:

Change Request 1 (CR1): `User` data type should be extended by an additional `postal code` field. This is the change request already presented in Sec. 2.2.

Change Request 2 (CR2): The interface of the external `authentication` service evolves. A provided method of this REST-interface is extended by an additional parameter. In particular, `IAuthentification` provides `checkUser` method, which includes `nickname` and `password` parameter. This method is extended by parameter `dateOfBirth`.

Materials	Treatment Group	Control Group	Expert Group
Textual System Specification with Context Information	x	x	x
Architecture Model Using PCM	x	x	x
Context Annotation Model Using KAMP	x		
KAMP Tool	x		
KAMP Usage Guide	x		
Task Description	x	x	x
Task Types Overview		x	x
Task Lists Examples		x	x
Warm-up quest.	x	x	x
KAMP analysis quest.	x		
Manual Analysis quest.		x	x
Post-experimental quest.	x	x	x

Table 3: Overview of materials for various groups

Change Request 3 (CR3): The data format of the REST-interface `IUserServiceTomcat` should be changed from JSON to XML.

Change Request 4 (CR4): In order to map object/relational, the user management system uses `Hibernate`. Due to licensing restrictions the `Hibernate` component should be replaced by an in-house developed component.

6.2 Results Summary

Tab. 4 shows average results of F1 metrics. Highest results are represented in bold. For task types (TT), over all change requests, the treatment group's average F1 score is higher than control group's but lower than expert group's. Treatment group is better than control group but slightly worse than expert group. This is caused by lower results for CR1 and CR4. In these cases the tool results were missing some task types compared to expert group. The root cause for this is a missing annotation in the annotation model, which was given to participants upfront. For task annotations (TA), over all change requests, the treatment group's average F1 score is higher than control group's and expert group's. For task annotations the average F1 score of treatment group is in all cases higher than control group's and expert group's values, either for average over all change requests and for individual change requests.

F1 score	Treatment Group	Control Group	Expert Group
F1(TT, all CRs)	0.914	0.850	**0.941**
F1(TA, all CRs)	**0.839**	0.416	0.598
F1(TT, CR1)	0.857	0.907	**0.958**
F1(TA, CR1)	**0.852**	0.620	0.736
F1(TT, CR2)	**1.000**	0.894	0.951
F1(TA, CR2)	**0.790**	0.389	0.488
F1(TT, CR3)	**1.000**	0.824	0.969
F1(TA, CR3)	**1.000**	0.352	0.758
F1(TT, CR4)	0.800	0.775	**0.887**
F1(TA, CR4)	**0.712**	0.306	0.409

Table 4: Average F1 score over all participants in each group and for each task

Using boxplots we compare the experiment result of the groups regarding task annotations, shown in Fig. 3 and Fig. 4. We observe that the treatment group achieved better

F1 scores compared to the other two groups. This can be seen for F1 scores for individual change requests in Fig. 3 and for all change requests in Fig. 4. Overall the study indicates that less-experienced users, who uses our approach identifies more affected artefacts compared with less-experienced users, who does not use KAMP. Thus, KAMP improves traceability of modifications in the software architecture model and facilitates impact analysis.

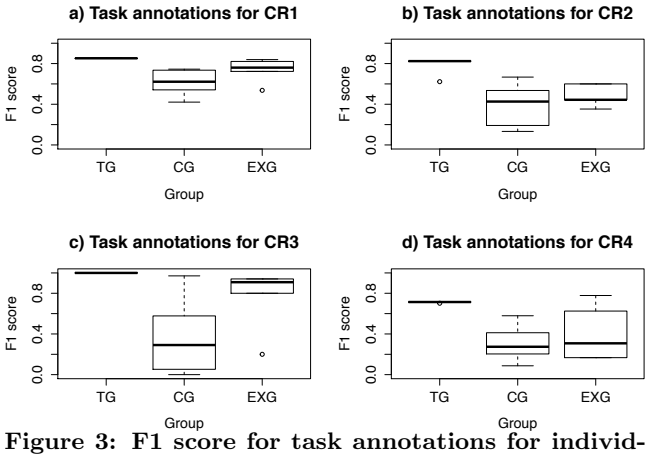

Figure 3: F1 score for task annotations for individual change requests

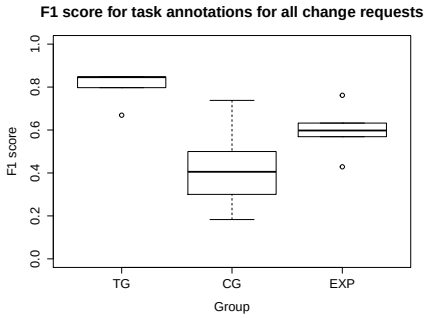

Figure 4: F1 score for task annotations for all change requests

As illustrated in Table 5, the treatment group spent less time for analysis compared with control group and expert group. Tool support reduced manual writing overhead and lookup time for task annotations. Within the same time users can analyse more change requests by using our tool. Hence, our approach improves scalability.

Groups \ Time	CR1	CR2	CR3	CR4	Total
EXP	27.8	13.4	10.4	10.8	62.4
KG	19	10.75	10.38	8.43	48
BG	10.33	7.17	6.83	5	24.25

Table 5: Average time in minutes to realise the change requests by each participant group

6.3 Validity discussions

Following we discuss the internal and external validity of our empirical study and the potential threats to validity.

Internal Validity

The *Internal validity* indicates whether the changes in the dependent variables caused the observed changes in the independent variable [23]. In other words, we have to ensure, that our treatment leads to resulting observations, and no side effects have influences on results. To improve the quality of our experiment we followed guidelines on empirical studies with students described in [8]. Experiment was properly integrated in student course. Experiment timeline was aligned with course schedule. Data and results were collected anonymously. Study goals are formulated to fit student involvement. External validity issues were considered early and also reflected in experiment goals. We divide the students based on their experience and skills into homogenous groups. All participants were trained to gain the same level of knowledge and received the same information and materials. Filling the orientation questionnaire helped the participants to get familiar with the system, tools, and information. The only explicit difference between groups was the treatment of the treatment group. This consists of the usage of analysis tool instead of manual analysis and additional trainings in tool application for the treatment group. The results of the treatment group comprise the tool-derived task lists. We took several means to avoid side effects. All analysis tasks were done individually by participants without any communication. Thus, the results were independently evaluated. The size of groups is limited. Differences in group sizes are due to participant drop-out after the treatment group was already trained. Thus, we could not re-balance groups. Missing annotations in the context model lead to reduced results for treatment group.

External Validity

The *External validity* indicates whether the results observed in the experiment can be applied to other groups, systems, or configurations [23]. In our experiment the students had to apply our tool to semi-automatically analyse the changes and to derive the task lists, as the students are a representative group of less-experienced users. On the other hand the researchers represent the experienced users.

We provided the participants with architecture model and context information in a textual manner, since building the architecture model and gathering the context information are not analysed in our experiment. In a real project, however, context information is widely dispersed. In our experiment, a set of all potential task types was provided, which does not exist in a real software development project. This approach assumes that the user models the structure of the system and the context information as complete as possible. A general limitation of model driven approaches is to find trade-offs between modelling effort und the applicability of the approach.

As described in group setup we used the number of participants was rather small. However, the empirical study used in the paper is appropriate to show the capability of KAMP, as the subject sample consists of both experienced and less-experienced user groups. Furthermore, we used stratified sampling methods to divide the participants in homogeneous groups. Therefore, we expect that a larger sample size does not significantly change the results of the study.

In conclusion the validation showed that the tool-based KAMP analysis by less-experienced users for the software architecture presented in Fig. 1 resulted in a better quality of task lists compared with manual analysis of less-experienced users and in task lists of similar quality compared to experts. According to task annotations KAMP's results were even better than less-experienced users and experts.

7. CONCLUSIONS

In this paper, we presented the scenario-based approach KAMP, which enables users to semi-automatically derive task lists from architecture models, which were manually annotated with context information involving necessary information to realise change requests. This task lists consider the changes to be made in all kinds of software artefacts (e.g. source code, test cases, or deployed instances). In addition, we described a running example illustrating each phase of KAMP: 1) In the preparation phase the software architect models and annotates the software architecture. 2) In the change request analysis phase the user models the change request by modifying the architecture model and triggers the tool to automatically calculate the work plan comprising the task lists. Furthermore, we presented a formalisation of KAMP by extending PCM and meta-model transformation. In order to evaluate KAMP, we conducted an empirical study with 14 computer science students and 5 researchers divided into three groups: an experiment group, a control group and an expert group. We analyse a real user management system. The study indicates, that KAMP improves the scalability of change propagation analysis due to automation and provides more homogenous and precise results.

Using an architecture model with context information, KAMP enables project members to semi-automatically generate a task list regarding various technical and organisational work areas during each phase of software life cycle. It considers not only stages in the development process, but also further aspects of a software product, such as test cases, deployment, build configurations. Furthermore, we consider both architecture modelling tasks and project management tasks. We assess the evolution of software using change requests as a special case of change scenarios and analyse the structural propagation of changes using an initial change request. Additionally, an automatically generated task list improves the scalability of change propagation analysis.

We aim to extend KAMP to include the meta-models of other domains, such as manufacturing automated system and their associated processes. To this end, KAMP's idea can be combined with existing cost estimation approaches, such as function point analysis and CoCoMO II. Furthermore, using KAMP an architecture maintainability simulator can be developed to calculate the maintainability of an architecture based on a set of change requests with certain probabilities. Moreover, KAMP could be extended to automatically derive change scenarios from recurring requirements to improve traceability.

Acknowledgments

This work was supported by the DFG (German Research Foundation) under the Priority Program SPP 1593: Design For Future – Managed Software Evolution (grant RE 1674/7-1). We thank Thomas Knapp for inspiring discussions and support during conception and tool development.

8. REFERENCES

[1] S. Becker et al. The palladio component model for model-driven performance prediction. *J. Syst. Softw.*, 82(1):3–22, 2009.

[2] P. Bengtsson and J. Bosch. Architecture level prediction of software maintenance. In *Proc. of 3rd CSMR*, pages 139–147, 1999.

[3] P. Bengtsson et al. Architecture-level modifiability analysis (ALMA). *J. Syst. Softw.*, 69(1-2):129–147, 2004.

[4] B. W. Boehm et al. *Software Cost Estimation with Cocomo II with Cdrom*. Prentice Hall, 2000.

[5] R. Carbon et al. *Architecture-Centric Software Producibility Analysis*. Fraunhofer IRB Verlag, 2012.

[6] S. K. Card et al. The keystroke-level model for user performance time with interactive systems. *CACM*, 23(7):396–410, 1980.

[7] S. K. Card et al. *The Psychology of Human-Computer Interaction*. L. Erlbaum Associates Inc., 1983.

[8] J. C. Carver et al. A checklist for integrating student empirical studies with research and teaching goals. *ESEJ*, 15(1):35–59, 2010.

[9] P. Clements et al. *Evaluating Software Architectures: Methods and Case Studies*. AW, 2002.

[10] M. Conway. How do committees invent? *Datamation*, 14:28–31, 1968.

[11] J. B. Dreger. *Function Point Analysis*. Prentice-Hall, Inc., 1989.

[12] D. Garlan et al. Evolution styles: Foundations and tool support for software architecture evolution. In *Software Architecture, WICSA/ECSA*, pages 131–140. IEEE, 2009.

[13] C. Heger and R. Heinrich. Deriving work plans for solving performance and scalability problems. In *EPEW*, pages 104–118, 2014.

[14] O. Hummel and R. Heinrich. Towards automated software project planning - extending palladio for the simulation of software processes. In *KPDAYS*, pages 20–29, 2013.

[15] E. Kindler and R. Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical report, University of Paderborn, Department of Computer Science, 2007.

[16] B. Kirwan and L. Ainsworth. *A Guide To Task Analysis: The Task Analysis Working Group*. Taylor & Francis, 2003.

[17] N. Lassing et al. Towards a broader view on software architecture analysis of flexibility. In *Proc. of APSEC*, pages 238–245, 1999.

[18] M. M. Lehman et al. Metrics and laws of software evolution - the nineties view. In *Proc. of 4th Intern. METRICS*, pages 20–, 1997.

[19] M. Naab. *Enhancing architecture design methods for improved flexibility in long-living information systems*. PhD thesis, Fraunhofer IESE, 2012.

[20] D. J. Paulish. *Architecture-centric Software Project Management: A Practical Guide*. AW, 2002.

[21] J. Stammel and R. Reussner. Kamp: Karlsruhe architectural maintainability prediction. In *Proc. of 1st. L2S2 Workshop*, pages 87–98, 2009.

[22] J. Stammel and M. Trifu. Tool-supported estimation of software evolution effort in service-oriented systems. In *Proc. 5th Intern. (MDSM) and 5th Intern. (SQM) Workshop*, volume 708, pages 56–63. CEUR-WS, 2011.

[23] C. Wohlin et al. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.