Deriving Work Plans for Solving Performance and Scalability Problems

Christoph Heger and Robert Heinrich

Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
{christoph.heger,robert.heinrich}@kit.edu

Abstract. The performance of an enterprise application (e.g. response time, throughput, or resource utilization) is an important quality attribute that can have a significant impact on a company's success. When a performance problem such as a performance bottleneck has been detected, the root cause identified and a solution proposed, developers have to identify the elements of the application often manually that will undergo changes and determine how these elements must be changed in order to implement the solution. Many existing approaches are able to identify the elements that have to be modified but only few are able to determine the necessary types of changes on these elements. Neither of the approaches supports developers with a work plan sketching the implementation steps. In this paper, we propose an approach to point developers the way towards an implementation of a performance or scalability solution with an ordered set of work activities. Rules are used to derive a work plan sketching the implementation of a solution for the particular application based on an initial set of work activities. The rule-based approach identifies impacted elements and determines how they should be changed. We demonstrate the proposed approach with a solution of a performance bottleneck as an example.

Keywords: Software Performance Engineering, Solution Implementation Support, Rules, Impact Propagation.

1 Introduction

The performance (i.e. resource usage, timing behaviour and throughput) of an information system directly influences the total cost of ownership (TCO) as well as the user satisfaction which are highly business critical metrics. Performance problems can be caused by various modifications such as error corrections, improvements, extensions or variations in user quantity and behaviour as well as changing requirements. After a performance problem has been detected and the root cause has been isolated, a performance expert often proposes a solution in form of abstract work activities like spliting an interface that have to be implemented in order to solve the problem. A solution can affect various parts of the application such as the architecture, implementation and/or configuration. Therefore, all elements (including possible side-effects) have to be identified that will undergo changes as well as the necessary types of changes for these elements. This time-consuming task is often done manually by developers or performance experts in advance to get an understanding of the implementation effort of a solution. Such cost factors can then be considered to select the most appropriate solution when a variety of solutions exists.

Currently, developers are not supported in this regard at the implementation level. Existing approaches for solving performance problems, for example [10,5,28,31], are model-based. Only Jing Xu considers in [31] to use the effort estimation of the designer for the necessary design model changes in selecting a solution among alternatives and suggests what should be changed, and in what way, but not how to do it. The approaches neither consider an existing code base nor support of the developer for implementing a solution at the implementation level. Existing impact propagation approaches on the other hand can identify and classify following changes and the impacted elements (side-effects) based on an initial set of changes [19] but neither of the approaches determines a work plan for developers describing the essential work activities of a change's implementation.

In light of these observations, we propose an approach to support developers in implementing a performance or scalability solution at the implementation level by deriving a work plan for the particular solution and application based on an initial set of work activities as a part of Vergil. Vergil is an approach to guide developers from a performance or scalability issue to solutions, by providing hypotheses about what to change, evaluating the changes in the context of the particular application and ranking the solutions to support developers in making a decision. The two major challenges in building a work plan are to identify the impacted elements and to determine how they are actually impacted; respectively how they should be changed. Vergil considers changes on the architecture level, implementation level and configuration level of the application. Architecture level changes are evaluated with architecture level peformance models, like the Palladio Component Model [7] or Marte [1] for UML, in contrast to implementation level changes that are evaluated with a system under test and measurement-based experiments. Due to the different levels of abstraction, work activities have to be traced down from the architecture level to the implementation level while developers are often most familiar with the implementation artifacts of an application.

The proposed approach for building work plans uses rules to propagate the impact within and between the architecture performance model and the source code model of an application. The impact propagation uses a correspondence model describing the equality of elements in the architecture performance model and the source code model. Work plans describe which elements of the application are impacted and how they should be changed. This offers three benefits for developers: (1) they are aware of how to change the application, (2) they are able to estimate the implementation effort based on work activities, and (3) they can discuss alternative solutions based on the type and number of impacted elements, and the types of changes required to implement a certain solution.

We demonstrate the approach using the solution of the “GOD” class antipattern [25] as an example in the context of the MediaStore [7] application. We use the Palladio Component Model as the architecture performance model and Java as programming language. Both are established and relevant technologies in industrial practice. Overall, in this paper, we make the following contributions:

1) We propose an approach to derive work activities sketching the implementation of a solution based on impact propagation between model elements.
2) We demonstrate the applicability of the approach with an example.

The remainder of the paper is structured as follows: In Section 2, we present the foundations of our approach. In Section 3, we give an overview of Vergil to position the content of this paper in the overall approach. In Section 4, we introduce the MediaStore example. We present the approach for building Vergil's work plans in Section 5. In Section 6, we discuss related work and conclude the paper in Section 7.

2 Foundations

The concept of work plans is inspired by the idea of the Karlsruhe Architectural Maintainability Prediction (KAMP) [26,27] approach to estimate the change effort based on a work plan. The goal of KAMP is to compare architectural design alternatives, which are represented by instances of the Palladio Component Model (PCM) [7], by estimating the effort of a change request in the context of each alternative. PCM is a software architecture simulation approach to analyze software at the model level for performance bottlenecks and scalability issues. It enables software architects to test and compare various design alternatives without the need to fully implement the application or buying expensive execution environments. PCM has already been used to detect and solve performance problems [28].

KAMP combines a top-down phase to determine the work activities and to create the work plan, with a bottom-up phase in which developers assign an effort estimation to each work activity in the plan. A work plan is a hierarchical structured collection of work activities and is stepwise refined into small tasks by identifying resulting changes and describing high-level changes on a lower level of abstraction. KAMP relies on the following assumptions: (a) change efforts must take into account all artifacts of system development and operation. Focusing only on code is not sufficient, (b) there are specified change scenarios, and (c) it is easier to estimate costs of small specific tasks than of coarse-grained tasks. We also rely on this assumptions.

The change estimation process starts with identifying model elements directly affected by a change request, such as an interface or component. To identify resulting changes, the direction of the change propagation is reversed to the direction of the reference-relation of architectural elements (architectural elements which refer to (or use) other elements are related by a reference-relation). In the case of include-/contains-relations (architectural elements which are contained in each other), a change of the inner element is propagated to the outer element and any change of the outer element is propagated to inner elements resulting in a refined set of work activities. Additionally, KAMP considers also other work activities like running unit or integration tests or deploying components [26].

The major difference between KAMP and our proposed approach is that KAMP operates on models only while our approach also considers the code basis. This allows for traceability and change impact propagation covering both, code and models. KAMP results in an unsorted set of tasks that have to be executed to realize a certain change request. As KAMP targets at effort estimation, there is no need to arrange the tasks in an order. In contrast, the proposed approach aims at creating a work plan that can be followed by developers through an ordered list of work activities.

3 Vergil Overview

We are currently developing the Vergil approach. The main goal of Vergil is the provisioning of solutions (e.g. to split an interface or to move functionality to a certain component) to developers for solving performance and scalability problems. Vergil combines the advantages of model-based performance improvement approaches like [10,31,5] and extends them with the introduction of measurement-based performance problem solution at the implementation level by means of monitoring-driven testing techniques and work plans sketching the implementation of the solution at the implementation level. The knowledge of how to change a system is formalized in rules (henceforth called change hypotheses) that are explored, evaluated and rated. The process consists of four major activities as shown in Figure 1 as BPMN diagram [14]. In the context of this paper, we are focussing only on the concepts of the work plans and the activities *Propagate Work Activities* and *Estimate Effort of Work Plans*.

The process starts with the *Extract Models* activity [22] that takes the source code of the application as input. The source code is parsed into the Source Code Model (SCM). An architecture performance model (APM) is extracted from the source code or when such a model already exists an existing one is imported. The APM provides an architectural view of the application and is used to evaluate architectural change hypotheses in the remainder of the process. During extraction, a correspondence model (COM) is build that links model elements in the APM and SCM that correspond to each other like interfaces. The extracted models are forwarded to the *Explore Change Hypotheses* subprocess.

The *Explore Change Hypotheses* subprocess takes the SCM, COM, APM and the Performance Problem Model (PPM) as input. The performance problem is formalized as symptom trace through the application and results in the PPM. A symptom like high response time or high CPU utilization references an SCM element where it can be observed such as a method and a description of the workload and usage profile as formalized in [29]. The subprocess starts with the *Test Change Hypotheses* activity that takes the change hypotheses, the performance requirements, the test environment and the models as input. In this activity, the applicability of change hypotheses is tested and the effect of change

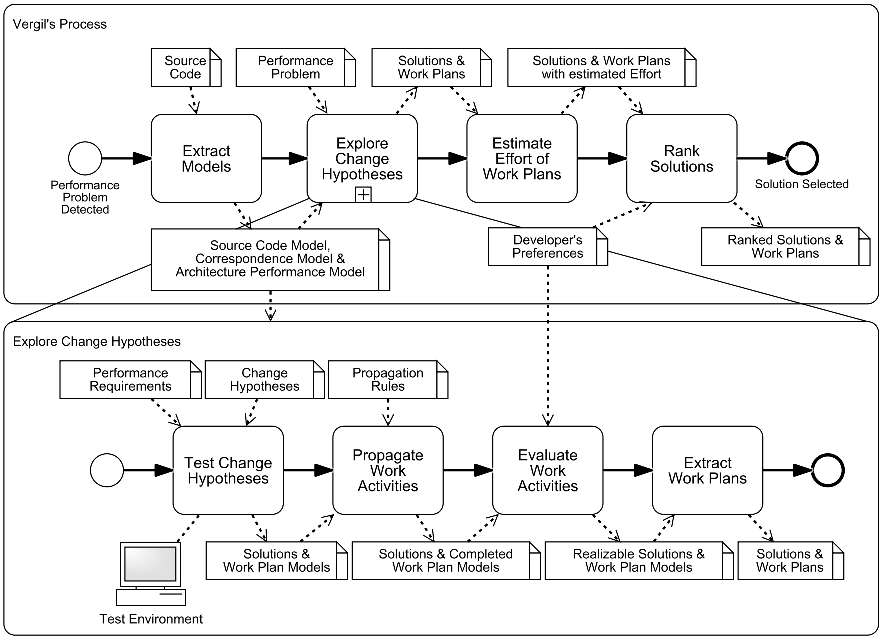
Vergil's Process
Source Code
Performance Problem
Solutions & Work Plans
Solutions & Work Plans with estimated Effort
Performance Problem Detected
Extract Models
Explore Change Hypotheses
Estimate Effort of Work Plans
Rank Solutions
Solution Selected
Source Code Model, Correspondence Model & Architecture Performance Model
Developer's Preferences
Ranked Solutions & Work Plans
Explore Change Hypotheses
Performance Requirements
Change Hypotheses
Propagation Rules
Test Change Hypotheses
Propagate Work Activities
Evaluate Work Activities
Extract Work Plans
Solutions & Work Plan Models
Solutions & Completed Work Plan Models
Realizable Solutions & Work Plan Models
Solutions & Work Plans
Test Environment

Fig. 1. Vergil Overview

hypotheses is evaluated to build solutions [31]. Change hypotheses provide the knowledge about what can be changed to solve a performance problem. It consists of a precondition that must be fulfilled in order to be applicable and a set of transformation rules that apply the changes on the defined level of abstraction (e.g. APM, SCM, etc.). Each change hypothesis also has a postcondition testing if the desired effect has taken place as well as the work plan model template. A condition can consist of any number of structural (on the SCM, PPM and APM model) and behavioral (on measurement or prediction results) conditions testing static and dynamic requirements of the hypothesis. The conditions are rules expressed in first-order logic. First-order logic has already been used before in literature to formalize performance antipatterns [11]. The exploration algorithm selects sets of change hypotheses with fulfilled precondition and evaluates their effect through instantiating the changes in the context of the particular application and on the hypothesis' level of abstraction (e.g. APM, SCM, etc.) and evaluates the performance. To give an example, two approaches for automated model refactoring for solving performance problems are presented in [4] and [31]. The performance evaluation is done by calibrating and simulating the APM instance or executing measurements with the system under test. The postcondition is tested with the returned results and if fulfilled, the impacted elements of the application are identified as well as how they are actually impacted building the Work Plan Models (WPMs) and its initial work activities based on the templates. The performance evaluation and the postcondition also ensure, that the changes do not lead to a performance degradation [3]. If the evaluation results of a single change hypothesis do not satisfy the performance requirements (for example response time, throughput, or CPU utilization), the current algorithm uses backtracking (as suggested by Arcelli and Cortellessa [3]) to find composite solutions (combinations of two or more change hypotheses) that fulfill the performance requirements. The solutions and the corresponding WPMs fulfilling the requirements are forwarded to the *Propagate Work Activities* activity. Propagation rules and the WPM of each solution are used to identify all impacted elements of a solution and to determine the necessary type of work activity for each element to complete the WPM.

The solutions and completed WPMs are forwarded to the *Evaluate Work Activities* activity. In this activity, the work activities and their referenced elements are validated that they can be changed [3] based on the information in the developer's preferences. In the developer's preferences, developers express what they prefer to change (e.g. configuration of a component, implementation etc.) and what they are unwilling to change (or cannot change), for example the architecture or particular parts of the application. If elements are impacted that the developer is unwilling to change or cannot change such as legacy systems or the architecture, the solution is discarded. Arcelli and Cortellessa raised the concern to take constraints such as costs and legacy constraints (e.g. the database cannot be changed) into account when proposing solutions [3]. The remaining solutions and WPMs are forwarded to the *Extract Work Plans* activity in which the WPMs are translated from their graph-based structure into an ordered list of work activities for the developers. The solutions and work plans are then forwarded to the *Estimate Effort of Work Plans* activity.

In the *Estimate Effort of Work Plans* activity, the developers are asked for an estimation of the effort they will need to complete the work plan. The effort estimation is a manual task done by the developers themself. The effort can vary between individual developers depending on their knowledge, experience and practice. Hence, it is possibly unreliable when done automatically. The effort is provided as unitless quantities, leaving the decision of the concrete unit of measurement by the developers, and is provided for all atomic work activities. The unit of measurement has to be the same for all work activities. The effort can be estimated, for example, in person (-hours, -days, or -months) [31], or story points (in the context of agile development such as SCRUM [24]) as well as function points which is also an established means for effort estimation [2]. In the case of experienced development teams, that have historic data from previous development projects for cost model calibration, the usage of a cost model such as COCOMO [8] is also possible. The process uses the given unitless quantities of each atomic work activity such as adding a method to an interface as input to compute the total effort estimation for a work plan. The estimated effort addresses the concern of taking the costs of solution alternatives into account [3,31]. The solutions and the work plans with estimated effort are forwarded to the *Rank Solutions* activity.

In the *Rank Solutions* activity, a multi-criteria decision analysis to rank the solutions is done. This activity addresses the issue of taking costs and constraints into account in deciding on an appropriate solution when a variety of choices exist [3,31]. The solution rating is done with a combination of the Simple Multi-Attribute Rating Technique (SMART) [12] and the Analytic Hierarchy Process (AHP) [23] taking the performance impact, cost factors, constraints and the developer's preferences into account. Developers are then able to discuss the proposed solutions based on the work plans, the impacted elements—and how they are actually impacted, the costs, and the estimated performance improvement. The selected solution and its work plan are the final result of the process.

4 MediaStore Example

One of the 14 notion- and domain-independent software performance antipattern defined by Smith and Williams is the "GOD" class [25]. The antipattern describes the problem of poorly distributed application intelligence when one class is performing all the work or contains all the application's data. A proposed solution [25] is to employ the locality principle and to move the functionality close to where it is needed. We have already investigated the "GOD" class antipattern and shown how it is automatic detectable with systematic experiments based on measurements in our previous work [30].

In this section, we use the "GOD" class as motivating example with the MediaStore [7] application. The MediaStore allows its users to upload and store audio files as well as to download audio files encoded in a less or equal audio bit rate compared to the uploaded one. The MediaStore is implemented with Java Enterprise Edition. The "GOD" class in the example is the `MediaStoreBean` that is accessed from the `WebGUIBean` and provides all the functionality as shown in Figure 2 (the Java source code elements are shown in UML notation for the sake of illustration). We omitted other components of the MediaStore application which are irrelevant for the example for the sake of simplicity. A detailed overview on the application is given in [7]. Figure 2 is devided into the *Current State* and *Current Deployment* showing the state of the application with the problem and the *Target State* and *Target Deployment* showing the application with the solution. The `WebGUIBean` and the `MediaStoreBean` are deployed on different servers. The `WebGUIBean` has to communicate with the `MediaStoreBean` to register or login users causing high response times for both operations. The change hypothesis of Vergil is to split the interface `IMediaStoreBean` and to move the functionality closer to the `WebGUIBean`. The hypothesis evaluates the changes on a higher level of abstraction for simplicity and uses a PCM of the MediaStore application. The automated refactoring of architectural models for solving performance problems has been shown, for example, in [4]. The change hypothesis provides the work activities to split the `IMediaStoreBean` interface, to add a new interface (in this example the `IUserManagementBean` interface), to move the methods `register` and `logIn` to the new interface, and to update the deployment descriptor of the application. The propagation of the provided work activities in the source code adds the work activities to split the `MediaStoreBean` class, to add a new class (in this example the `UserManagementBean`), to move

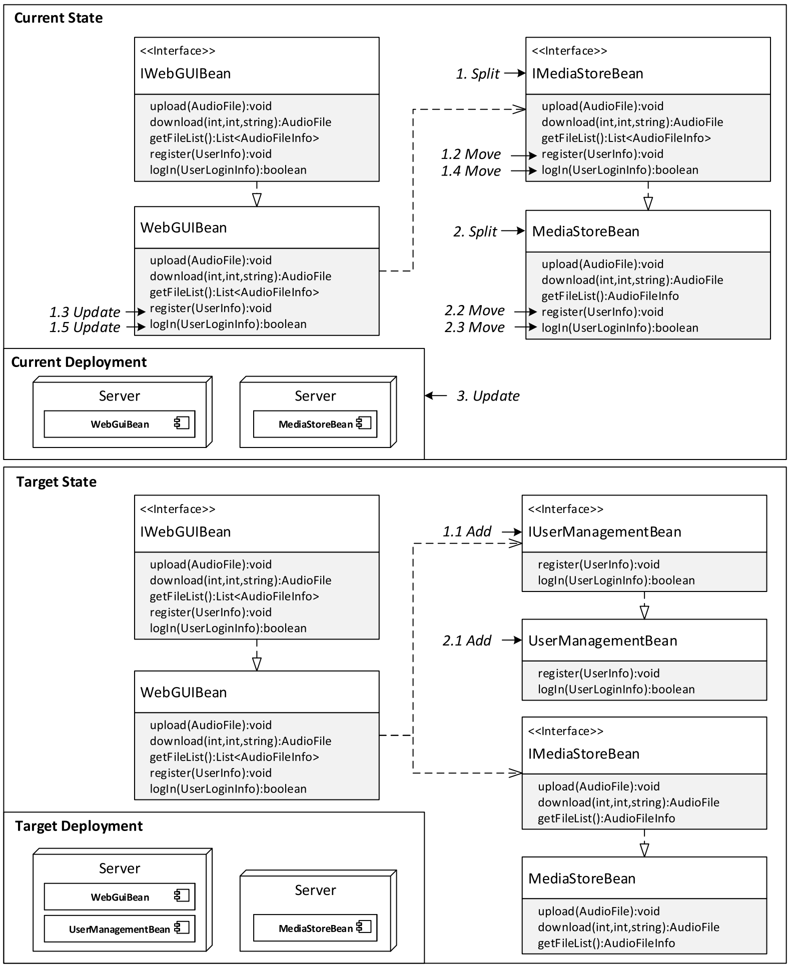

Fig. 2. MediaStore Example Overview

the methods implementing interface methods to the new class, and to update the methods of the `WebGUIBean` to call the corresponding methods of the new interface. The necessary work activities and their proposed order are shown in Figure 2. The determined work plan for the refactoring is shown in Table 1.

In this work plan, there are composite activities (*split*, *move*), that can consist of other composite activities or atomic activities (*add, update, delete*). All composite activities are broken down until they are expressed by atomic activities. The follow-up activities (*test* and *deploy*) result from the atomic and composite work activities.

In the scenario above, we initially have an architectural change proposed and embodied in the change hypothesis. Vergil uses the PCM as starting point for building the work plan of architectural changes. We propagate the impact to the implementation using the correspondence between model elements and source

Table 1. Work Plan MediaStore Example

Work Activity	**Effort [Minutes]**
Split interface IMediaStoreBean	
Add `NewInterface`	10
Move method `register` to `NewInterface`	
Add method `register` to `NewInterface`	5
Delete method `register` from `IMediaStoreBean`	5
Update method `register` of `WebGUIBean`	15
Move method `logIn` to `NewInterface`	
Add method `logIn` to `NewInterface`	5
Delete method `logIn` from `IMediaStoreBean`	5
Update method `logIn` of `WebGUIBean`	15
Split class MediaStoreBean	
Add class `NewClass`	10
Move method `register` to `NewClass`	
Add method `register` to `NewClass`	10
Delete method `register` from `MediaStoreBean`	5
Move method `logIn` to `NewClass`	
Add method `logIn` to `NewClass`	10
Delete method `logIn` from `MediaStoreBean`	5
Update Deployment Descriptor	5
Test application	30
Deploy application	60
Estimated Effort	**195**

code elements. Therefore, we extract a model from the source code (for example with a tool such as the Java Model Parser and Printer [15]). A correspondence, for example, describes the equality relation of the element of type interface `IMediaStoreBean` in the PCM instance of the application and the `IMediaStoreBean` interface in the source code.

5 Vergil's Work Plans

A work plan sketches what essentially needs to be done to implement the solution by modelling abstract work activities without prescribing (and limiting the developer) on how the solution is concretely implemented in the application. A work plan is an ordered set of work activities. A work acitivity can be atomic such as add, delete, or update an element like a class, interface, or method, or composite such as split, move, merge, swap, or replace elements. The concept of work activities is inspired by the taxonomy of change types which has been introduced by Lehnert et al. in [18]. This taxonomy is similar to the work activity concept used in KAMP. Both approaches consider a graph-based representation of the software artifacts and use atomic operations and composite operations to categories modifications. The formalization of the work plan is shown in Figure 3. The work plan also lists follow-up activities such as redeployment or testing activities.

A composite activity can be composed of other composite and atomic activities. Refinement rules are used to break composite activities in the work plan down into atomic activities. For example, the composite work activity "Move" to move a method from one interface to a new one consists of the atomic work

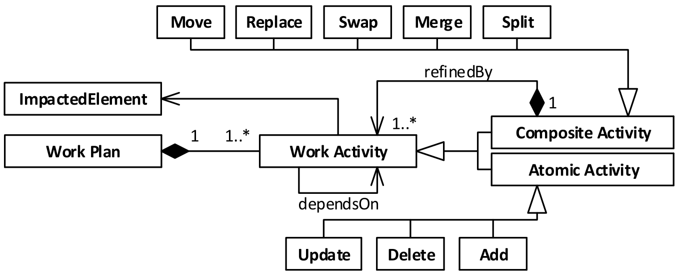

Fig. 3. Work Plan meta-model

activities "Add" to add the method to the new interface and "Delete" to delete the method from the old interface. The evolution of the work plan model through the application of such a refinement rule for the latter work activitiy is shown as an example in Figure 4. The underlying graph transformation rule matches elements of type `Method` in the SCM that are referenced from a *Move* work activitiy but are not already referenced by a *Delete* work activitiy. For all matches, the rule adds the *Delete* work activity as refinement expressed through the added *refinedBy* relation to the work plan.

The *refinedBy* and *dependsOn* relations between work activities in the work plan model instance are used to extract the ordered list of work activities for the developers. An activity like "Split" that has a *refinedBy* relation to an "Move" activity is added as child of that activitiy in the work plan. An activitiy like "Split" that has a *dependsOn* relationship to another "Split" activity is added after that activity in the work plan.

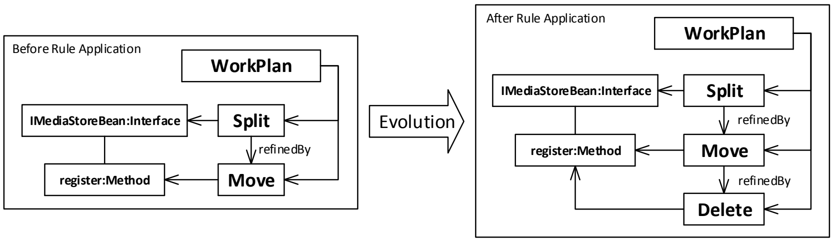

Fig. 4. Refinement Rule "Delete Method" Example

The initial work activities are provided by the change hypothesis as work plan template and the impacted elements are determined by the instantiation of the changes.

The impact on an element can cause an impact on other elements that are in a relation. To identify all elements and how they are impacted, Vergil uses impact propagation rules to identify additional impacted elements that are in a relation to already impacted elements and the work aktivities in the work plan referencing the impacted elements. The propagation uses the work plan model as starting point and correspondences of elements in different model instances for propagation between APM and SCM (vertical propagation) and the relation within APM and SCM for propagation wihtin the models (horizontal propagation). The correspondence model describes the equality relation (One-To-One relation) of elements from different model instances and meta models but the same underlying application. The correspondences are described in the correspondence model. The correspondence model is independent from other meta models. It only references elements from other model instances. For each impacted element, the rule knows the resulting work activitiy and adds it to the work plan together with a reference on the impacted element. Rules are also used to conclude follow-up activities such as adding tests when a new interface is created or the redeployment of components when elements of the implementation of that component are impacted.

For example, when an interface in SCM is referenced from a *Split* work activity, then the rule knows that a class implementing that interface has to be splitted too. Figure 5 shows the evolution of the work plan model through applying the rule for the split interface work activity as an example. The rule matches all classes that are not referenced from a *Split* work activity and that implement the interface which is referenced from a *Split* work activity. For all matches, the rule adds the *Split* work activity to the work plan model and a reference to the impacted class.

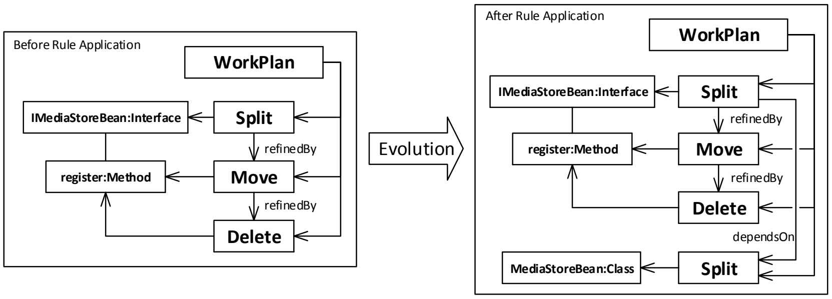

Fig. 5. Horizontal Propagation Rule "Split Interface" Example

6 Related Work

In this section, we review impact propagation approaches for their support of determining work plans to implement solutions and we discuss common effort estimation approaches.

Impact Propagation. Most approaches that have been proposed to assess the propagation of change impacts are limited to source code as shown by a recent study [17] and are often able to identify impacted elements only. Some of the proposed rule-based approaches, such as the approaches of Keller et al. [16] or Briand et al. [9], are able to classify how the impacted element is actually impacted and has to be changed while others are not able to detect impacts in heterogeneous software artefacts like source code and models. The research conducted by Lindvall and Sandahl [20] outlines the applicability of traceability relations for impact analysis. The use of correspondence relations between different views and viewpoints has been shown by Eramo et al. in [13]. A more detailed review of existing approaches and their limitations is provided in [19]. An introduction to the topic of change impact analysis discussing techniques and problems is given in [6].

The impact analysis approach of Lehnert et al. [19] combines impact analysis, multi-perspective modelling, and horizontal traceability to determine further change propagation. The impact propagation technique is based on the type of dependency which exists between EMF-based models and the type of change which is applied on one of the model elements. The underlying hypothesis is that the change type, dependency type and the types of involved artifacts determine if and how a change ripples to related artifacts. Therefore, a set of impact propagation rules is used to identify all impacted elements in a recursive manner. The rules are derived, as example, from relations defined in the meta-models of artifacts such as inheritance-relations between classes or implementation-relations between classes and interfaces as well as from Correspondences according to design methodologies such as the equivalence between UML and Java classes or between UML and Java packages [19]. The type of changes is expressed through a taxonomy of change types comprising of atomic operations (*add*, *delete*, *update*) and composite operations (*move*, *replace*, *split*, *merge*, *swap*), where the latter can consist of sequences of atomic and composite operations. Each rule receives the changed element, the type of change, and a list of all related elements as input. From this input, a list of all impacted element(s) along with the resulting change type(s) is computed. This output is then again fed into the impact analysis process. The propagation takes the dependency relation into account to limit the size of impact sets. The rules determine how impacted elements are actually impacted [19].

The main objective of impact propagation approaches is to identify the impacted elements. Few of them also deal with the determination of how an element must be changed. Neither of the approaches has the objective to build a model that abstractly models the implementation of the changes with the necessary work activities. Our proposed approach extends the concepts of existing impact propagation approaches, i.e. [19], by building such a model describing the refactoring of the application that can be transfered into a work plan for developers.

Effort Estimation. Three categories of related work on change effort estimations can be distinguished – complexity metrics (esp. of source code), estimations based on the extent of changes in requirements, and architecture-based procedures. One of the most common complexity metric for software systems is the cyclomatic complexity [21]. Complexity metrics only take into account the structure of the system. They can be elicited automatically but their conversion into effort or costs is unclear and empirically not consistent proven. There are various approaches to initial effort estimation of software development projects based on requirements, cf. algorithmic effort estimation and effort-based project planning [8]. 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 architecture. Some existing approaches target at scenario-based software architecture analysis but lack a formalized architecture description or are limited to software development but do not take into account management tasks. An overview of related work on architecture-based effort estimation is given in [26]. 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 development as well as management effort. KAMP evaluates maintainability for concrete change requests. It estimates change efforts using semi-automatic derivation of work plans, bottom-up effort estimation, and guidance in investigation of estimation supports (e.g., design and code properties, team organization, development environment, and other influence factors).

In our proposed approach, we use the work plan-based concept of KAMP as the foundation for effort estimation by developers to consider cost factors in making a decision among solution alternatives as well as to build a model of the refactoring. We apply the concept of work activities not only on the architecture performance model as it is currently provided by KAMP but also on implementation level artifacts. We also introduce relations in the work plan model that are used to extract an ordered list of work activities.

7 Conclusion

Sketching the implementation of a solution is an essential part of guiding developers to the solution of performance and scalability problems. Vergil's work plans aim to provide this support for the proposed solutions. The work plans as an ordered list of work activities guide developers without prescribing how the implementation is concretely realized. Developers are able to discuss and compare solution alternatives based on the impacted elements and the necessary type of change, they are aware of how they have to change the application and which parts of the application are affected and they can estimate the implementation effort for each solution before making a decision on which solution will be implemented. Vergil uses rules and a graph-based representation of the application to determine the necessary work activities and to build the work plans. We demonstrated the approach with an example while the validation of the approach is part of our current research. We plan to conduct an emperical case study to validate the approach with a group of developers at SAP.

Due to the graph-based approach, we are not limited to a particular architecture performance model like the Palladio Component Model or to a specific programming language like Java as long as a graph-based representation is available. Rules may have to be adjusted and new ones have to be created in order to take other models such as UML or Entity Relationship diagrams into account that describe the application from a different perspective.

We plan to implement the proposed approach in the context of Vergil's framework that is currently under development. Vergil's automated work plan builder will then be used in the validation of the overall approach.

Acknowledgements. We would like to thank Alexander Wert, Roozbeh Farahbod, Michael Langhammer and Max Kramer for the discussions and their feedback. We would also like to thank Jonas Kunz and Sven Kohlhaas for their support. This work is supported by the German Research Foundation (DFG), grant RE 1674/6-1, and the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

References

1. UML Marte, `http://www.omgmarte.org`
2. Albrecht, A.J.: Measuring application development productivity. In: Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium, vol. 10 (1979)
3. Arcelli, D., Cortellessa, V.: Software model refactoring based on performance analysis: Better working on software or performance side? In: FESCA (2013)
4. Arcelli, D., Cortellessa, V., Di Ruscio, D.: Applying model differences to automate performance-driven refactoring of software models. In: Balsamo, M.S., Knottenbelt, W.J., Marin, A. (eds.) EPEW 2013. LNCS, vol. 8168, pp. 312–324. Springer, Heidelberg (2013)
5. Arcelli, D., Cortellessa, V., Trubiani, C.: Antipattern-based model refactoring for software performance improvement. In: Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures. ACM (2012)
6. Arnold, R.S., Bohner, S.A.: Software Change Impact Analysis. IEEE Computer Society Press (1996)
7. Becker, S., Koziolek, H., Reussner, R.: The palladio component model for model-driven performance prediction. Journal of Systems and Software 82(1) (2009)
8. Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R., Selby, R.: Cost models for future software life cycle processes: Cocomo 2.0. Annals of Software Engineering 1(1) (1995)
9. Briand, L.C., Labiche, Y., O'sullivan, L.: Impact analysis and change management of uml models. In: Proceedings of the International Conference on Software Maintenance, ICSM 2003. IEEE (2003)
10. Cortellessa, V., Di Marco, A., Eramo, R., Pierantonio, A., Trubiani, C.: Approaching the model-driven generation of feedback to remove software performance flaws. In: 35th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2009. IEEE (2009)
11. Cortellessa, V., Di Marco, A., Trubiani, C.: An approach for modeling and detecting software performance antipatterns based on first-order logics. Software and Systems Modeling (2012)
12. Edwards, W.: How to use multiattribute utility measurement for social decision making. IEEE Transactions on Systems, Man and Cybernetics 7(5) (1977)
13. Eramo, R., Pierantonio, A., Romero, J.R., Vallecillo, A.: Change management in multi-viewpoint system using asp. In: 2008 12th Enterprise Distributed Object Computing Conference Workshops. IEEE (2008)
14. O.M. Group. Business process model and notation, bpmn (2011)
15. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Jamopp: The java model parser and printer. Techn. Univ., Fakultät Informatik (2009)
16. Keller, A., Schippers, H., Demeyer, S.: Supporting inconsistency resolution through predictive change impact analysis. In: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation, p. 9. ACM (2009)

17. Lehnert, S.: A review of software change impact analysis. Ilmenau University of Technology, Tech. Rep. (2011)
18. Lehnert, S., Farooq, Q., Riebisch, M.: A taxonomy of change types and its application in software evolution. In: 2012 IEEE 19th International Conference and Workshops on Engineering of Computer Based Systems (ECBS) (April 2012)
19. Lehnert, S., Riebisch, M., et al.: Rule-based impact analysis for heterogeneous software artifacts. In: 2013 17th European Conference on Software Maintenance and Reengineering (CSMR). IEEE (2013)
20. Lindvall, M., Sandahl, K.: Traceability aspects of impact analysis in object-oriented systems. Journal of Software Maintenance: Research and Practice 10(1) (1998)
21. McCabe, T.J.: A complexity measure. In: Proceedings of the 2nd International Conference on Software Engineering, ICSE 1976, Los Alamitos, CA, USA. IEEE Computer Society Press (1976)
22. Parsons, T.: Automatic Detection of Performance Design and Deployment Antipatterns in Component Based Enterprise Systems. PhD thesis, University College Dublin (2007)
23. Saaty, T.: The analytic hierarchy and analytic network processes for the measurement of intangible criteria and for decision-making. In: Multiple Criteria Decision Analysis: State of the Art Surveys. Springer, New York (2005)
24. Schwaber, K., Beedle, M.: Agile Software Development with Scrum. Pearson (2002)
25. Smith, C.U., Williams, L.G.: Software performance antipatterns. In: Workshop on Software and Performance (2000)
26. Stammel, J., Reussner, R.: Kamp: Karlsruhe architectural maintainability prediction. In: Proceedings of the 1st Workshop des GI-Arbeitskreises Langlebige Softwaresysteme (L2S2): Design for Future-Langlebige Softwaresysteme (2009)
27. Stammel, J., Trifu, M.: Tool-supported estimation of software evolution effort in service-oriented systems. In: Joint Proceedings of the First International Workshop on Model-Driven Software Migration (MDSM 2011) and the Fifth International Workshop on Software Quality and Maintainability (SQM 2011), vol. 708 (2011)
28. Trubiani, C., Koziolek, A.: Detection and solution of software performance antipatterns in palladio architectural models. In: ICPE (2011)
29. van Hoorn, A., Rohr, M., Hasselbring, W.: Generating probabilistic and intensity-varying workload for web-based software systems. In: Kounev, S., Gorton, I., Sachs, K. (eds.) SIPEW 2008. LNCS, vol. 5119, pp. 124–143. Springer, Heidelberg (2008)
30. Wert, A., Oehler, M., Heger, C., Farahbod, R.: Automatic Detection of Performance Anti-patterns in Inter-component Communications. In: Proceedings of the 10th International Conference on Quality of Software Architecture, QoSA 2014 (2014)
31. Xu, J.: Rule-based automatic software performance diagnosis and improvement. Performance Evaluation 69(11) (2012)