in Proceedings of Third European Conference on Software Maintenance and Reengineering, Amsterdam, Netherlands, March 1999, pp. 139-147.

Architecture Level Prediction of Software Maintenance

PerOlof Bengtsson & Jan Bosch
Department of Computer Science and Business Administration
University of Karlskrona/Ronneby
S-372 25 Ronneby, Sweden
+46 457 787 41
[PerOlof.Bengtsson | Jan.Bosch] @ide.hk-r.se

Abstract

A method for the prediction of software maintainability during software architecture design is presented. The method takes (1) the requirement specification, (2) the design of the architecture (3) expertise from software engineers and, possibly, (4) historical data as input and generates a prediction of the average effort for a maintenance task. Scenarios are used by the method to concretize the maintainability requirements and to analyze the architecture for the prediction of the maintainability. The method is formulated based on extensive experience in software architecture design and detailed design and exemplified using the design of software architecture for a haemo dialysis machine. Experiments for evaluation and validation of the method are ongoing and future work.

1 Introduction

One of the major issues in software development today is the software quality. Rather than designing and implementing the correct functionality in products, the main challenge is to satisfy the software quality requirements, e.g. performance, reliability, maintainability and flexibility. The notion of software architecture has emerged during the recent years as the appropriate level to deal with software qualities. This because, it has been recognized [1,2] that the software architecture sets the boundaries for the software qualities of the resulting system.

Traditional object-oriented software design methods, e.g. [5,14,21] focus primarily on the software functionality and give no support for software quality attribute-oriented design, with the exception of reusability and flexibility. Other research communities focus on a single quality attribute, e.g. performance, fault-tolerance or real-time. However, real-world software systems are never just a real-time system or a fault-tolerant system, but generally require a balance of different software qualities. For instance, a real-time system that is impossible to maintain or a high-performance computing system with no reliability is of little use.

To address these issues, our ongoing research efforts aim on developing a method for designing software architectures, i.e. the ARCS method [6]. In short, the method starts with an initial architecture where little or no attention has been given to the required software qualities. This architecture is evaluated using available techniques and the result is compared to the requirements. Unless the requirements are met, the architect transforms the architecture in order to improve the software quality that was not met. Then the architecture is again evaluated and this process is repeated until all the software quality requirements have been met or until it is clear that no economically or technically feasible solution exists.

The evaluation of software architectures plays a central role in architectural design. However, software architecture evaluation is not well understood and few methods and techniques exist. Notable exceptions are the SAAM method discussed in [15] and the approach described in [10]. In this paper, we propose a method for predicting maintainability of a software system based on its architecture. The method defines a *maintenance profile*, i.e. a set of change scenarios representing perfective and adaptive maintenance tasks. Using the maintenance profile, the architecture is evaluated using so-called scenario scripting and the expected maintenance effort for each change scenario is evaluated. Based on this data, the required maintenance effort for a software system can be estimated. The method is based on our experience in architectural design and its empirical validation is part of ongoing and future work.

The remainder of this paper is organized as follows. In the next section, the maintenance prediction method is presented in more detail. The architecture used as an example is discussed in section 3 and the application of the method in section 4. Related work is discussed in section 5 and the paper is concluded in section 6.

2 Maintenance Prediction Method

The maintenance prediction method presented in this paper estimate the required maintenance effort for a software system during architectural design. The estimated effort can be used to compare two architecture alternatives or to balance maintainability against other quality attributes.

The method has a number of inputs: (1) the requirement specification, (2) the design of the architecture (3) expertise from software engineers and, possibly, (4) historical maintenance data. The main output of the method is, obviously, an estimation of the required maintenance effort of the system built based on the software architecture. The *maintenance profile* is a second output from the method. The profile contains a set of scenario categories and a set of scenarios for each category with associated weighting and analysis (scripting) results.

The maintenance prediction method consists of six steps:

1. Identify categories of maintenance tasks
2. Synthesize scenarios
3. Assign each scenario a weight
4. Estimate the size of all elements.
5. Script the scenarios
6. Calculate the predicted maintenance effort.

The steps are discussed in more detail in the following sections.

2.1 Identify categories of maintenance tasks

Software maintainability is defined by IEEE [13] as:

> *The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.*

This definition renders three categories of maintenance, i.e. corrective, perfective and adaptive. The prediction method focuses only on perfective and adaptive maintenance and does not predict efforts required for corrective maintenance. Nevertheless, the remaining categories are too abstract to be relevant in this process step. Instead, the categories are defined based on the application or domain description. For example, a haemo dialysis machine might have maintenance scenarios concerned with treatment maintenance, hardware changes, safety regulation changes, etc. These categories reflect the meaning of the maintainability requirement in the context of this application or domain and give the designer a better understanding of the requirements posed on the architecture.

2.2 Synthesize scenarios

For each of the maintenance categories, a representative set of concrete scenarios is defined. The software architect, or the domain expert, is responsible for selecting the scenarios such that the set is representative for the maintenance category. The number of scenarios in the set is dependent on the application and the domain, but in our experience we define about ten scenarios for each category. Scenarios should define very concrete situations. Scenarios that specify types of maintenance cases or sub categories is to be avoided. For example, a scenario could be, “Due to changed safety regulations, a temperature alarm must be added to the hydraulic module in the dialysis machine”. Another example is, “Due to a new type of pump, the pump interface must be changed from duty cycle into a digital interface, with a set value in kP (kilo Pascal)”.

It is important to note that our use of the term ‘scenarios’ is different from Object-Oriented design methods where the term generally refers to use-case scenarios, i.e. scenarios describing system behavior. Instead, our scenarios describes an action, or sequence of actions that might occur related to the system. Hence, a change scenario, describes a certain maintenance task. In addition, due to reasons of space, in this paper we present scenarios as *vignettes* [15] rather than in their full size representation.

2.3 Assign each scenario a weight

Change scenarios have different likelihood of actually occurring during the lifetime of the system. In order to generate an accurate measure for maintainability, the prediction method requires probability estimates, i.e. weights, for each scenario. These probabilities are used for balancing the impact on the prediction of more occurring and less occurring maintenance tasks. We define the weight measure as the relative probability of this scenario resulting in a maintenance task during a particular time interval, e.g. a year, or between two releases. Consequently, scenarios that describe often-recurring maintenance tasks will get higher probabilities and therefore impact the predicted value more and the architecture will generally be optimized for incorporating those maintenance tasks with minimal effort.

The weight of scenarios is produced in two ways. If no historical maintenance data is available from similar applications or earlier releases, the software architect, or the domain expert, estimates the scenario weights. If empirical data about maintenance of the product exists in the organization, the probability data of earlier maintenance efforts should be used as basis for weighting. Based on the probability data for the individual scenarios, it is possible to calculate a probability figure for each category as well. The exact calculation of probabilities is illustrated in section 4.

2.4 Estimate the size of all elements

To estimate the maintenance effort, the size of the architecture needs to be known and the sizes of the affected components need to be known. The component size influences the effort required to implement a change in the component. At least three techniques can be used for estimating the size of components. First, the size of every component can be estimated using the estimation technique of choice. Secondly, an adaptation of an Object-Oriented metric (SIZE2 [8]) metric may be used (SIZE2') [2]. Finally, when historical data from similar applications or earlier releases is available, existing size data can be used and extrapolated to new components.

2.5 Script the scenarios

Based on the selected scenarios from each maintenance category, the weights defined for each scenario and the categories and the size data for the components, we estimate the maintainability of the architecture by *scripting* [16] the scenarios. For each scenario, the impact of the realization of that scenario in the architecture and its components is evaluated. Thus, find what components are affected and to what extent will they be changed.

For example, implementing the earlier described scenario of adding a temperature alarm in the dialysis machine would require changes to the hydraulic module component and addition of three new components of type device and controlling algorithm. In addition, the components for system definition and the protective system need to be changed.

2.6 Calculate the predicted maintenance effort

The prediction value is a weighted average for the effort (expressed as size of modification) for each maintenance scenario. Based on that, one can calculate an average effort per maintenance task. To predict the required maintenance effort for a period of time, an estimation or calculation of the number of maintenance tasks has to be done. That figure is then multiplied with the average effort per maintenance task. Note that the above is only necessary when predicting maintenance effort for a period of time. When comparing two alternative architectures, it is sufficient to compare the weighted average effort per maintenance task.

$$M_{tot} = \sum_{n=1}^{k_s} \left(P(S_n) \cdot \sum_{m=1}^{k_c} V(S_n, C_m) \right)$$

$P(S_n)$ the probability weight of scenario n
$V(S_n,C_m)$ the affected volume of component m in scenario n
k_s = number of scenarios
k_c = number of components in architecture

Figure 1: Assessment Calculation Equation

3 Example Application Architecture

Haemo dialysis systems present an area in the domain of medical equipment where competition has been increasing drastically during recent years. The aim of a dialysis system is to remove water and certain natural waste products from the patient's blood. Patients that have, generally serious, kidney problems and consequently produce little or no urine use this type of system. The dialysis system replaces this natural process with an artificial one.

An overview of a dialysis system is presented in figure 2. The system is physically separated into two parts by the dialysis membrane. On the left side the dialysis fluid circuit takes the water from a supply of a certain purity (not necessarily sterile), dialysis concentrate is added using a pump. A sensor monitors the concentration of the dialysis fluid and the measured value is used to control the pump. A second pump maintains the flow of dialysis fluid, whereas a third pump increases the flow and thus reduces the pressure at the dialysis fluid side. This is needed to pull the waste products from the patient's blood through the membrane into the dialysis fluid. A constant flow of dialysis fluid is maintained by the hydro mechanic devices denoted in the figure with rectangles with curls.

On the right side of figure 2, the extra corporal circuit, i.e. the blood-part, has a pump for maintaining a specified blood flow on its side of the membrane. The patient is connected to this part through two needles usually located in the arm that take blood to and from the patient. The extra corporal circuit uses a number of sensors, e.g. for identifying air bubbles, and actuators, e.g. a heparin pump to avoid cluttering of the patients blood while it is outside the body.

However, these details are omitted since they are not needed for the discussion in this paper.

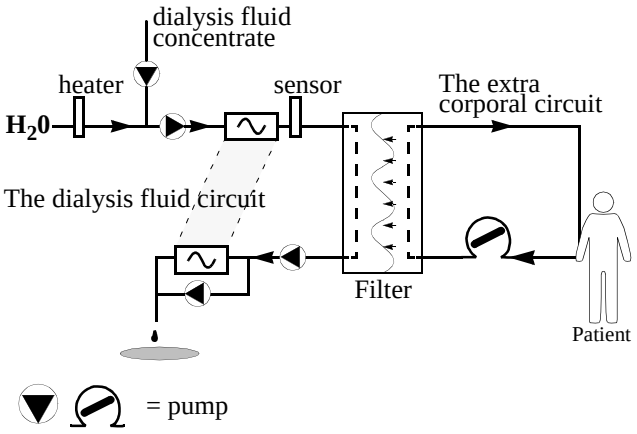

Figure 2: Schematic of Haemo Dialysis Machine

The dialysis process, or *treatment*, is by no means a standard process. A fair collection of treatments exists including, for example, Haemo Dialysis Filtration (HDF), Ultra Filtration (UF) and other variations, such as single needle/single pump, double needle/single pump. Treatments are changed due to new research results but also since the effectiveness of a particular treatment decreases when it is used too long for a patient. Although the abstract function of a dialysis system is constant, a considerable set of variations exists already. Based on experience, the involved company anticipates several additional changes to the software, hardware and mechanical parts of the system that will be necessary in response to developments in medical research.

3.1 Requirements

The aim during architectural design is to optimize the potential of the architecture (and the system built based on it) to fulfil the software quality requirements. For dialysis systems, the driving software quality requirements are *maintainability*, *reusability*, *safety*, *real-timeliness* and *demonstrability*. Below, we elaborate on the maintainability requirement.

Maintainability. Past haemo dialysis machines produced by our partner company have proven to be hard to maintain. Each release of software with bug corrections and function extensions have made the software harder and harder to comprehend and maintain. One of the major requirements for the software architecture for the new dialysis system family is that maintainability should be considerably better than the existing systems, with respect to *corrective* but especially *adaptive* maintenance:

- *Corrective maintenance* has been hard in the existing systems since dependencies between different parts of the software have been hard to identify and visualize.
- *Adaptive maintenance* is initiated by a constant stream of new and changing requirements. Examples include new mechanical components as pumps, heaters and AD/DA converters, but also new treatments, control algorithms and safety regulations. All these new requirements need to be introduced in the system as easily as possible. Changes to the mechanics or hardware of the system almost always require changes to the software as well. In the existing system, all these extensions have deteriorated the structure, and consequently the maintainability, of the software and subsequent changes are harder to implement. Adaptive maintainability was perhaps the most important requirement on the system.

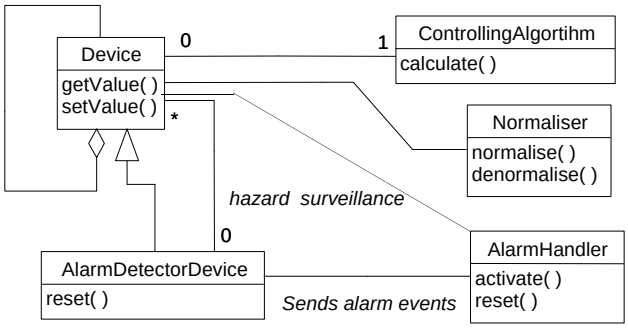

Figure 3: The relations of the logical archetypes

3.2 Logic Archetypes

One of our main concerns when we designed the software architecture for the haemo dialysis machine was maintainability. The logical archetypes are based on device hierarchy (figure 3). The archetypes are central to the design and important for understanding the haemo dialysis application architecture when doing the scripting, i.e. change impact analysis.

Device. The system is modeled as a device hierarchy, starting with the entities close to the hardware as leaves, ending with the complete system as the root. For every device, there are zero or more sub-devices and a controlling algorithm. The device is either a leaf device or a logical device.

ControllingAlgorithm. In the device archetype, information about relations and configuration is stored. Computation is done in a separate archetype, the ControllingAlgorithm, which is used to parameterize Device components.

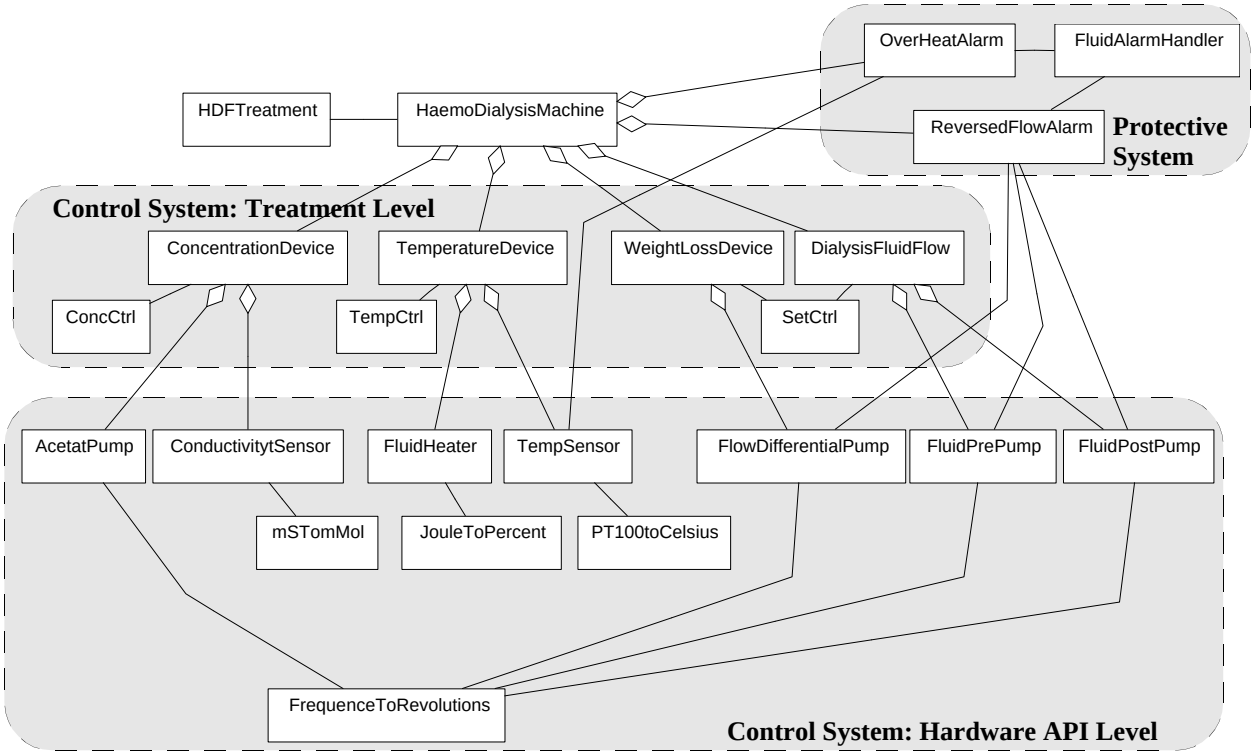

Figure 4: Example Haemo Dialysis Application Architecture

Normaliser. To convert from and to different units of measurement the normalization archetype is used.

AlarmDetectorDevice. Is a specialization of the Device archetype. Components of the AlarmDetectorDevice archetype are responsible for monitoring the sub devices. When threshold limits are crossed an AlarmHandler component is invoked.

AlarmHandler. The AlarmHandler is the archetype responsible for responding to alarms by returning the haemo dialysis machine to a safe-state or by addressing the cause of the alarm.

3.3 Scheduling Archetypes

Haemo dialysis machines are required to operate in real time. However, haemo dialysis is a slow process that makes the deadline requirements on the system less tough to adhere to. A treatment typically takes a few hours and during that time the system is normally stable. Since the timing requirements are not that tight we designed the concurrency using the *Periodic Object pattern* [19]. It has been used successfully in earlier embedded software projects.

Scheduler. The scheduler archetype is responsible for scheduling and invoking the periodic objects. Only one scheduler element in the application may exist and it handles all periodic objects of the architecture. The scheduler accepts registrations from periodic objects and then distributes the execution between all the registered periodic objects.

Periodic object. A periodic object is responsible for implementing its task using non-blocking I/O and using only the established time quanta. The tick() method will run to its completion and invoke the necessary methods to complete its task.

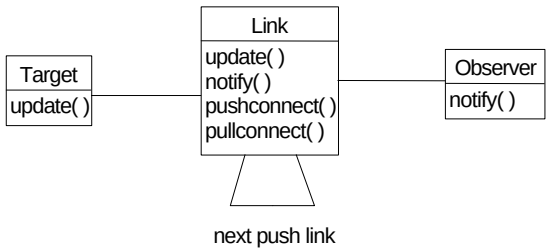

Figure 5: Push/Pull Update Connection

3.4 Connector Archetypes

Causal connections [18] implements the communication between the architecture elements. The principle is similar to the *Observer pattern* [11] and the *Publisher-Subscriber pattern* [7]. The usage of the connection allows for

dynamic reconfiguration of the connection, i.e. push or pull. (Figure 5)

Target. Maintains information that other entities may be dependent on. The target is responsible for notifying the link when its state changes.

Observer. Depends on the data or change of data in the target. Is either updated by a change or by own request.

Link. Maintains the dependencies between the target and its observers. Also holds the information about the type of connection, i.e. push or pull. It would be possible to extend the connection model with periodic updates.

3.5 Application Architecture

The archetypes represent the building blocks that we may use to model the application architecture of a haemo dialysis machine. In figure 4, the application architecture is presented. The archetypes allow for the application architecture to be specified in a hierarchical way, with the alarm devices being orthogonal to the control systems device hierarchy. The description serves as input for scenario scripting, which is architecture level impact analysis in the maintainability case.

This also allows for a layered view of the system, not meaning that the architecture is layered. For example, to specify a treatment we only have to interface the closest layer of devices to the HaemoDialysisMachine device (figure 4). There would be no need to understand or interfacing the lowest layer.

4 PREDICTION Example

In this section, we will present an example prediction for the architecture presented in section 3. It is presented to illustrate the practical usage of the method, rather than to give a perfect prediction of this particular case.

4.1 Scenario Categories

In the example domain we identify the following categories of maintenance tasks;

1. Hardware changes, i.e. additions and replacements of hardware require changes to software.
2. Algorithm changes, i.e. algorithms become obsolete and is replaced by new improved ones.
3. Safety changes, i.e. safety standards changes and sets new requirements on the system.
4. Medical advances requires changes, i.e. new treatments and parameters are introduced.
5. Communication and I/O change.
6. Market driven changes. Different markets or countries require certain functionality.

We use these broad categories of maintenance task in the next step of the method to ensure that we include all the important aspects in the broad sense.

4.2 Change Scenarios

When we have the categories, we list a number of scenarios for each category that describe concrete maintenance tasks that may occur during the next maintenance phase.

Scenarios describe a possible situation and change scenarios, in particular, describe possible change situations that will cause the maintenance organization to perform changes in the software and/or hardware. For reasons of space, the scenarios are presented very brief. In our real world application of the method, the scenarios are generally more verbose.

This list presented in table 1 represents a maintenance profile, i.e. it profiles the relevant interpretation of software maintenance for the resulting system.

Table 1: Maintenance Profile

Category	Scenario Description	Weight
Market Driven	**C1** Change measurement units from Celsius to Fahrenheit for temperature in a treatment.	0.043
Hardware	**C2** Add second concentrate pump and conductivity sensor.	0.043
Safety	**C3** Add alarm for reversed flow through membrane.	0.087
Hardware	**C4** Replace duty-cycle controlled heater with digitally interfaced heater using percent of full effect.	0.174
Medical Advances	**C5** Modify treatment from linear weight loss curve over time to inverse logarithmic.	0.217
Medical Advances	**C6** Change alarm from fixed flow limits to follow treatment.	0.087
Medical Advances	**C7** Add sensor and alarm for patient blood pressure	0.087
Sum		1.0

Table 1: Maintenance Profile

Category	Scenario Description	Weight
Hardware	**C8** Replace blood pumps using revolutions per minute with pumps using actual flow rate (ml/s).	0.087
Com. and I/O	**C9** Add function for uploading treatment data to patient's digital journal.	0.043
Algorithm Change	**C10** Change controlling algorithm for concentration of dialysis fluid from PI to PID.	0.132
Sum		1.0

In section 2.2, a total number of ten scenarios per category were suggested. Both for reasons of space and illustrativeness, we will however only use a total of ten scenarios in this example.

4.3 Assign Weights

Each scenario has a certain likelihood of appearing during the next phase of maintenance. Each scenario is therefore assigned a value for the probability of which any arbitrary maintenance task from the maintenance phase will be like this (see table 1, column 3). The sum of all the weights must be exactly 1.

For assigning each weight we can use two approaches. First, we can make qualified guesses that some changes are more likely than others. Domain experts or software engineers can support the estimation with experiences from the earlier maintenance phases. Second, we can collect and categorize historical data from other similar development projects.

Table 2: Estimated Component Size

Component	Size (LOC)
HDFTreatment	200
HaemoDialysisMachine	500
ConcentrationDevice	100
TemperatureDevice	100
Sum	2805

Table 2: Estimated Component Size

Component	Size (LOC)
WeightlossDevice	150
DialysisFluidFlowDevice	150
ConcCtrl	175
TempCtrl	30
SetCtrl	30
AcetatPump	100
ConductivitySensor	100
FluidHeater	100
TempSensor	100
FlowdifferentialPump	100
FluidPrePump	100
FluidPostPump	100
mSTomMol	20
JouleToPercent	20
PT100toCelsius	40
FrequenceToRevolutions	40
OverHeatAlarm	50
ReversedFlowAlarm	300
FluidAlarmHandler	200
Sum	2805

4.4 Component Size Estimates

There are two ways of estimating the size of components. First, the component sizes are estimated using the estimation technique of choice. In most organizations, some estimation technique is used and could also be used for the method presented in this paper. In many cases the project planning already use and require size estimates of the system for work division. These estimates are either equivalent to those or, the estimates for the architecture are one level more fine grained.

Second, a prototype implementation or a previous release may be available which can be used as basis for the

estimation. The size estimates presented in table 2 are synthesized using the size data from an early prototype implementation.

Table 3: Impact Analysis per Scenario

Scenario	Dirty Components	Volume
C1	HDFTreatment (20% change) + new Normaliser type component	,2*200+ 20 = 60
C2	ConcentrationDevice (20% change) + ConcCtrl (50% change) + reuse with 10% modification of AcetatPump and ConductivitySensor	,2*100+ ,5*175+ ,1*100+ ,1*100 = 127,5
C3	HaemoDialysisMachine (10% change) + new AlarmHandler + new AlarmDevice	,1*500+ 200+100 =350
C4	Fluidheater (10% change), remove DutyCycleControl and replace with reused SetCtrl	,1*100 = 10
C5	HDFTreatment (50% change)	,5*200 = 100
C6	AlarmDetectorDevice (50% change) + HDFTreatment (20% change) + HaemoDialysisMachine (20% change)	,5*100+ ,2*200+ ,2*500 = 190
C7	see C3	= 350
C8	new ControllingAlgorithm + new Normaliser	100+20 = 120
C9	HDFTreatment (20% changes) + HaemoDialysisMachines (50% changes)	,2*200+ ,5*500 = 290
C10	Replacement with new ControllingAlgorithm	= 100

4.5 Script the Scenarios

The scenario scripting, or change impact analysis, is done by investigating the required changes to the components of the application architecture and the severity of the change in percent. To this stage we have not investigated if any particular method for scripting is to prefer over others. An introduction to change impact analysis can be found in [4]. The result of scripting the scenarios in our example is shown in table 3.

4.6 Calculation

The prediction is calculated using the formula presented in figure 1:

0.043*60 + 0.043*127.5 + 0.087*350 + 0.174*10 + 0.217*100 + 0.087*190 + 0.087*350 + 0.087*120 + 0.043*290 + 0.132*100 = 145 LOC / Change

Given that we estimate around 20 maintenance task for the predicted period of time, either from first to second release or for the coming year. Assuming that we also have an estimated or historical data of maintenance productivity we are able to extrapolate the estimate from this method to a total maintenance effort estimate. We assume that we have a perfective maintenance productivity that are similar to the median reported in [12], i.e. 1.7 LOC/day, which amounts to about 0.2 LOC/hour. Then we get the following estimate:

20 changes per 145 LOC = 2900 LOC

2900 / 0.2 = 14 500 hours of effort

This would represent a medium project of about 6-7 persons working around 2300 hours per year.

5 Related work

Architecture assessment is important for achieving the required software quality attributes. A well-known method is the scenario-based architecture assessment method (SAAM) [15]. The SAAM method of assessing software architecture is primarily intended for assessing the final version of the software architecture and involves all stakeholders in the project. The method we propose differs in that it does not involve all stake holders, and thus requires less resources and time, but instead provides an instrument to the software architects that allows them to repeatedly evaluate architecture during design. We recognize the need for stakeholder commitment and believe that these two methods should be used in combination.

In addition, a method based on an ISO standard has been proposed in [10], which suggests a rigorous metrics approach to the problem of software quality evaluation of software architectures. The method make a clear distinction on internal and external views, where the external view is the view important to or seen by the clients of the resulting products. The rigorous ambition makes it hard to believe that the method will be suitable for usage in every cycle in an iterative and incremental software architecture design process.

Within the software maintenance community efforts have been made to predict maintainability. A set of object oriented metrics was validated in [17] to be good predictors

of the software maintenance effort for each module in a software system. However, the metrics suite used requires data that can only be collected from the source code and thus cannot be used for software architecture when no or only prototype source exist.

Software change impact analysis is an established research area within the software maintenance community [4]. A variety of models and techniques exist. However, the techniques are often based on having the software available and its source code and this prohibits their application to software architectures. To the best of our knowledge, no impact analysis method exists that is specific to software architecture.

6 Conclusions

We have presented a method for prediction of maintainability from software architecture. The method provides a number of benefits: First, it is practical and has been used during architectural design. Second, its use provides benefits for more than just the prediction, e.g. improved requirements understanding. Third, it combines the usage of design expertise and historical data for validation of scenario profiles. This way the method more efficiently incorporates the uniqueness of the changes for the predicted period of time. Fourth, the method is very slim in terms of effort and produced artifacts. Finally, it is suitable for design processes that iterate frequently with evaluation in every iteration, e.g. as in the ARCS method [3].

Weaknesses of the method include its dependency on a representative maintenance profile and the problem of validating that a profile is representative. In our future work we aim to address this in a number of ways. First, we are planning a study investigating how individual knowledge and expertise affects the representativeness of a maintenance profile and thus how the activities concerned with generating maintenance profiles should be staffed. Second, we will continue to study industrial maintenance practice and intend to incorporate that knowledge can be incorporated into the method. Finally, we intend to study the sensitivity of the method for variation of the input variables, e.g. if the method is more or less sensitive to the representativeness of the maintenance scenario profile than we currently think, or if the size estimates are more significant for the results.

References

1. L. Bass, P. Clements, R. Kazman, *'Software Architecture In Practise'*, Addison Wesley, 1998.
2. P. Bengtsson, *'Towards Maintainability Metrics on Software Architecture: An Adaptation of Object-Oriented Metrics'*, First Nordic Workshop on Software Architecture (NOSA'98), Ronneby, August 20-21, 1998.
3. P. Bengtsson, J. Bosch, 'Scenario Based Software Architecture Reengineering', *Proceedings of International Conference of Software Reuse 5 (ICSR5)*, 1998.
4. Bohner, S. A, Arnold, R.S., *Software Change Impact Analysis*, IEEE Computer Society Press, 1996.
5. G. Booch, *Object-Oriented Analysis and Design with Applications*, (2nd edition), Benjamin/Cummings Publishing Company, 1994.
6. J. Bosch, P. Molin, 'Software Architecture Design: Evaluation and Transformation', *submitted*, 1997.
7. F. Buschmann, R. Meunier, H. Rohnert, M.Stahl, *Pattern-Oriented Software Architecture - A System of Patterns*, John Wiley & Sons, 1996.
8. S.R. Chidamber and C.F. Kemerer, 'Towards a metrics suite for object-oriented design,' *in proceedings: OOPSLA'91*, pp.197-211, 1991.
9. CEI/IEC 601-2 Safety requirements standard for dialysis machines.
10. J.C. Dueñas, W.L. de Oliveira, J.A. de la Puente, 'A Software Architecture Evaluation Method,' *Proceedings of the Second International ESPRIT ARES Workshop*, Las Palmas, LNCS 1429, Springer Verlag, pp. 148-157, February 1998.
11. E. Gamma, R. Helm, R. Johnson, J.O. Vlissides, *Design Patterns Elements of Reusable Design*, Addison.Wesley, 1995.
12. Henry, J. E., Cain, J. P., "A Quantitative Comparison of Perfective and Corrective Software Maintenance", Journal of Software Maintenance: Research and Practice, John Wiley & Sons, Vol 9, pp. 281-297, 1997
13. IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 610.12-1990.
14. I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, *'Object-oriented software engineering. A use case approach'*, Addison-Wesley, 1992.
15. R. Kazman, L. Bass, G. Abowd, M. Webb, 'SAAM: A Method for Analyzing the Properties of Software Architectures,' *Proceedings of the 16th International Conference on Software Engineering*, pp. 81-90, 1994.
16. P.B. Krutchen, 'The 4+1 View Model of Architecture', *IEEE Software*, pp. 42-50, November 1995.
17. W. Li, S. Henry, 'Object-Oriented Metrics that Predict Maintainability', *Journal of Systems and Software*, vol. 23, no. 2, pp. 111-122, November 1993.
18. C. Lundberg, J. Bosch, "Modelling Causal Connections Between Objects", *Journal of Programming Languages*, 1997.
19. P. Molin, L. Ohlsson, 'Points & Deviations - A pattern language for fire alarm systems,' to be published in *Pattern Languages of Program Design 3*, Addison-Wesley.
20. L. H. Putnam, 'Example of and Early Sizing, Cost and Scehdule Estimate for an Application Software System', *Proceedings of COMPSAC'78*, IEEE , pp. 827-832, Nov 1978.
21. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-oriented modeling and design*, Prentice Hall, 1991.