Scenario-based Software Architecture Reengineering

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

ABSTRACT
A method for reengineering software architectures is presented in this paper. The method explicitly addresses the quality attributes of the software architecture. Assessment of quality attributes is performed primarily through the use of scenarios. Design transformations are used to improve quality attributes that do not satisfy the requirements. Assessment and design transformation can be performed for several iterations until all requirements are met. To illustrate the method, the reengineering of a prototypical measurement system into a domain-specific software architecture is used as an example.

Keywords
software architecture, reengineering, design methods, reuse

1 INTRODUCTION
Reengineering of a software system is generally initiated by a major change in the requirements the system should fulfill. These changes are often concerned with the non-functional rather than the functional requirements. For example, due to architecture erosion [17], the maintainability of the software system may have deteriorated and to improve this, the system is reengineered.

To the best of our knowledge, few architecture reengineering methods have been defined. Traditional system design methods tend to focus on the functionality that is to be provided by the system. Much less effort is spent on the non-functional requirements that are to be fulfilled by the system. Object-oriented methods, e.g. [2,11,21], address reusability but no assessment of the achieved result is done. Other research communities, e.g. real-time [15] and fault-tolerant [25], have proposed design methods that incorporate design steps for supporting their respective non-functional requirement (NFR). However, these approaches generally focus on a single NFR.

In our experience in systems development [4,5,16], we have identified that a system often is a real-time system, as well as a fault-tolerant system and a reusable system: the system should provide all these properties. However, as identified, among others, by [3], non-functional requirements are often conflicting, e.g. real-time versus reusability, flexibility versus performance, reliability versus changeability, etc. The difficulty of system design originates from the need for carefully balancing of the various NFRs. For the discussion in this paper, we distinguish between development NFRs, e.g. reusability and maintainability, and operational NFRs, e.g. reliability and performance.

The ability of a software system to fulfill its NFRs is, up to a considerable extent, determined by its architecture. Once an application architecture is finalised, the upper boundaries for most quality attributes have been set. On the other hand, architectural design and reengineering are the steps in software development that is least understood and supported by traditional means.

The contribution of this paper, we believe, is that a practical method for reengineering software architectures is presented and illustrated using a real-world example.

The remainder of this paper organised as follows. In the next section, the example system that will be reengineered is introduced. The architecture reengineering method is introduced in section 3. Section 4 introduces the measurement system application domain, the non-functional requirements that should be fulfilled and the scenarios used to assess the requirements. In section 5 the method is exemplified by applying it to the example application and described in terms of quality attribute assessments and architecture transformations. Section 6 discusses related work and the paper is concluded in section 7.
2 EXAMPLE: BEER CAN INSPECTION

To illustrate the architecture reengineering method, a beer can inspection system is used as a basis for the discussion in the remainder of the paper. The inspection system is placed at the beginning of a beer can filling process and its goal is to remove dirty beer cans from the input stream. Clean cans should just pass the system without any further action.

The system consists of a triggering sensor, a camera and an actuator that can remove cans from the conveyer belt. When a can is detected, the system receives a trigger event from the hardware trigger. After a predefined amount of time, the camera samples an image of the can. This sampling is repeated a few times and subsequently the measured values, i.e. images, are compared to the ideal images and a decision about removing or not removing the can is made. If the can should be removed, the actuator is invoked at a point in time relative to the point in time when the trigger event took place. In figure 1, the process is presented graphically.

![Figure 1. Example beer can measurement system](image)

The application architecture of the beer can system is presented in figure 2. The architecture is based solely on the functional requirements of the system without any explicit design effort with respect to non-functional requirements such as, for example, reusability and performance.

![Figure 2. Object model of the beer can application](image)

The architecture of the presented system is used as the basis for creating a domain-specific software architecture that allows the software engineer to instantiate applications in the domain of measurement systems.

3 ARCHITECTURE REENGINEERING METHOD

In software industry, our experience is that non-functional requirements are generally dealt with by a rather informal process during architecture design and reengineering. Once the system is implemented, tests are performed to determine whether the NFRs are fulfilled. If not, parts of the system are redesigned. This approach is not satisfactory because, first, the iteration over system development is generally very costly and, second, the redesign cannot be planned and budgeted.

Conventional design methods, e.g. [2,9,20], tend to focus on achieving the required system functionality, rather than the non-functional requirements. The various NFR-based research communities identified this as unsatisfactory, and have proposed their own design methods for developing real-time [15], high-performance [25] and reusable systems [12], respectively. However, all these methods focus on a single quality attribute and treat all others as having secondary importance, if at all. We consider these approaches unsatisfactory since the software engineer needs to balance the various quality attributes for any realistic system. However, lacking a supporting method, software engineers in industry design and reengineer system architectures in an ad-hoc and intuition-based manner, with the associated disadvantages.

To address this, we have defined an architecture reengineering method that provides a more explicit and objective approach. The method is described in the remainder of this section. For a more extensive overview of the method, we refer to [6].

3.1 Overview

The input for the architecture reengineering method consists of the updated requirements specification and the existing software architecture. As output an improved architectural design is generated. In figure 3, the steps in the method are presented graphically. The process starts with incorporating the new functional requirements in the architecture. Although software engineers generally will not design a system less reliable or reusable, the non-functional requirements are not explicitly addressed at this stage. The result is a first version of the application architecture design. This design is assessed with respect to the non-functional requirements. Each quality attribute (QA) is estimated using a qualitative or quantitative assessment technique. The estimated quality attribute values are compared to the NFRs. If all estimations are as good or better than required, the architectural design process is finished. Otherwise, the second stage is entered: architecture transformation. During this stage, the architecture is improved by selecting appropriate QA-optimizing transformation. Each set of transformations (one or more) results in a
new version of the architectural design that is fed back to the earlier stage in the process. This design is again evaluated and the same process is repeated, if necessary, until all NFRs are fulfilled or until the software engineer decides that no feasible solution exists.

3.3 Assessing Non-Functional Requirements

One of the core features of the architecture reengineering method is that the non-functional properties of a system or application architecture are explicitly evaluated. For assessing the architecture of the existing system, the system itself can be used. After the first transformation, however, no concrete system is available for evaluation. So, how does one measure system properties based on an abstract specification such as an architectural design? One has to identify that it is not possible to measure the QAs for the final system based on the architecture design. Instead, the goal is to evaluate the potential of the designed architecture to reach the required levels for the QAs. For example, some architectural styles, e.g. layered architectures, are less suitable for systems where performance is a major issue, even though the flexibility of this style is high.

Four different approaches for assessing quality attributes have been identified, i.e. scenarios, simulation, mathematical modelling and objective reasoning. For each quality attribute, the engineer can select the most suitable approach for evaluation. In the subsequent sections, each approach is described in more detail.

3.3.1 Scenario-based evaluation

To assess a particular quality attribute of an architecture, a set of scenarios is developed that concretises the actual meaning of the attribute. For instance, the maintainability QA may be specified by scenarios that capture typical changes in requirements, underlying hardware, etc. The scenarios can then be used to evaluate the number of changes required to adapt the architecture, and the corresponding system, to the new situation. Another example is robustness where the architecture can be evaluated with respect to the effects of invalid input.

In our experience, scenario-based assessment is particularly useful for development NFRs. NFRs such as maintainability can be expressed very naturally through change scenarios.

3.3.2 Simulation

Simulation of the architecture using an implementation of the application architecture provides a second approach for estimating quality attributes. The main components of the architecture are implemented and other components are simulated resulting in an executable system. The context, in which the system is supposed to execute, could also be simulated at a suitable abstraction level. This implementation can then be used for simulating application behaviour under various circumstances.

Simulation complements the scenario-based approach in that simulation is particularly useful for evaluating operational NFRs, such as performance of fault-tolerance by actually executing the architecture implementation, whereas scenarios are more suited for evaluating development NFRs. However, the implementation of the architecture in the simulation can

---

**Figure 3. Outline of the architecture reengineering method**

The architecture reengineering method has been applied, in various forms, in three projects, i.e. for fire-alarm systems [16], measurement systems [4] and dialysis systems. In the following sections, the individual stages of the method are described in more detail.

3.2 Functionality-based Architecture Redesign

Based on the updated requirement specification, a redesign of the top-level decomposition of the system into its main components is performed. The main issue during this phase is to evaluate the core abstractions based on which the system is structured and change or replace them wherever necessary. Although these abstractions are modelled as objects, our experience, see e.g. [16], is that these objects are not found immediately in the application domain. Instead, they are the result of a creative process that, after analysing the various domain entities, abstracts the most relevant properties and models them as architecture entities. Once the abstractions are identified, the interactions between them are defined in more detail.

The process of identifying the entities that make up the architecture is different from, for instance, traditional object-oriented design methods [2]. Those methods start by modelling the entities present in the domain and organise these in inheritance hierarchies, i.e. a bottom-up approach. Our experience is that during architectural design and reengineering it is not feasible to start bottom-up since that would require dealing with the details of the system. Instead one needs to work top-down.
be used to evaluate, for instance, maintainability, by changing the implementation according to change scenarios and measuring the required effort.

3.3.3 Mathematical modelling
Various research communities, e.g. high-performance computing [25], reliability [22], real-time systems [15], etc., have developed mathematical models, or metrics, that can be used to evaluate especially operational NFRs. Different from the other approaches, the mathematical models allow for static evaluation of architectural design models.

Mathematical modelling is an alternative to simulation since both approaches are primarily suitable for assessing operational NFRs.

3.3.4 Objective Reasoning
A fourth approach to assessing NFRs is through reasoning based on logical arguments. Experienced software engineers often have valuable insights that may prove extremely helpful in avoiding bad design decisions. Although these experiences generally are based on anecdotal evidence, most can often be justified by a logical line of reasoning.

This approach is different from the other approaches in that the evaluation process is less explicit and more based on subjective factors as intuition and experience.

3.4 Architecture Transformation
Once the architecture properties have been assessed, the estimated values are compared to the requirements specification. If one or more of the NFRs are not satisfied, the architecture has to be changed to achieve these non-functional requirements also. In the architectural reengineering method discussed in this paper, changes to the architecture are performed as architecture transformations. Each transformation leads to a new version of the architecture that has the same functionality, but different values for its quality attributes.

Five categories of architecture transformations have been identified. In the sections below, each category is discussed in more detail.

3.4.1 Impose architectural style
Shaw and Garlan [23] and Buschmann et al. [7] present several architectural styles that improve certain quality attributes for the system the style is imposed upon and impair other NFRs. Certain styles, e.g. the layered architectural style, increase the flexibility of the system by defining several levels of abstraction, but generally decrease the performance. With each architectural style, a fitness for each system property is associated. The most appropriate style for a system depends primarily on its non-functional requirements. Transforming an architecture by imposing an architectural style results in a complete reorganisation of the architecture.

3.4.2 Impose architectural pattern
A second category of transformations is the use of architectural patterns. An architectural pattern is different from an architectural style in that it is not predominant in the architecture. It is also different from a design pattern since it affects the complete architecture, or at least the larger part of it. Architectural patterns generally impose a rule [20] on the architecture that specifies how the system will deal with one aspect of its functionality, e.g. concurrency or persistence.

3.4.3 Apply design pattern
A less dramatic transformation is the application of a design pattern on a part of the architecture. For instance, an abstract factory pattern [9] might be introduced to abstract the instantiation process for its clients. The abstract factory increases maintainability, flexibility and extensibility of the system since it encapsulates the actual class type(s) that are instantiated, but decreases the efficiency of creating new instances due to the additional computation, thereby reducing performance and predictability. Different from imposing an architectural style or pattern, causing the complete architecture to be reorganised, the application of a design pattern generally affects only a limited number of classes in the architecture. In addition, a class can be involved in multiple design patterns without creating inconsistencies.

3.4.4 Convert NFRs to functionality
Another type of transformation is the conversion of a NFR into a functional solution that consequently extends the architecture with functionality not related to the problem domain but used to fulfil a NFR. Exception handling is a well-known example that adds functionality to a component to increase the fault-tolerance of the component.

3.4.5 Distribute requirements
The final type of transformation deals with NFRs using the divide-and-conquer principle: a NFR at the system level is distributed to the subsystems or components that make up the system. Thus, a NFR X is distributed over the n components that make up the system by assigning a NFR $x_i$ to each component $c_i$ such that $X=x_1 + ... + x_n$. A second approach to distribute requirements is by dividing the NFR into two or more functionality-related NFRs. For example, in a distributed system, fault-tolerance can be divided into fault-tolerant computation and fault-tolerant communication.

4 MEASUREMENT SYSTEMS
The domain of measurement systems denotes a class of systems used to measure the relevant values of a process or product. These systems are different from the, better known, process control systems in that the measured values are not directly, i.e. as part of the same system, used to control the production process that creates the product or process that is measured. A measurement system is used for quality control
on produced products that can then be used to separate acceptable from unacceptable products or to categorise the products in quality grades.

The goal of the reengineering project was to define a domain-specific software architecture (DSSA) that provides a reusable and flexible basis for instantiating measurement systems. Being a rather prototypical instance of a measurement system, the software architecture of the beer can inspection system introduced in section 2 was used as a starting point. This architecture, obviously, did not fulfill the non-functional requirements that should be fulfilled by a DSSA. Consequently, it needs to be re-engineered and transformed to match the requirements.

The challenge of reengineering projects is to decide when one has achieved the point where the reengineered architecture fulfills its requirements. The functional requirements generally can be evaluated relatively easy by tracing the requirements in the design. Non-functional requirements such as reusability and robustness, on the other hand, are much harder to assess. In the next section, the approach to evaluating some of the NFRs put on the measurement system DSSA is described.

4.1 Non-functional requirements
The DSSA for measurement systems should fulfill a number of requirements. The most relevant non-functional requirements in the context of this paper are reusability and maintainability. Measurement systems also have to fulfill real-time and robustness requirements, but we leave these out of the discussion in this paper.

As was described in section 3, development NFRs are generally easiest assessed using scenarios. The assessment process consists of defining a set of scenarios for each NFR, manually executing the scenarios for the architecture and subsequently interpreting the result. The assessment can be performed in a ‘complete’ or a statistical manner. In the first approach, a set of scenarios is defined that combined cover the concrete instances of the NFR. For instance, for reusability, all relevant ways of reusing the architecture or parts of it are represented by a scenario. If all scenarios are executed without problems, the reusability of the architecture is optimal. The second approach is to define a set of scenarios that not covers all possible cases but that forms a representative subset. The ratio between scenarios that the architecture can handle and scenarios not handled well by the architecture provides an indication of how well the architecture fulfills the NFR. For example, if eight out of ten reusability scenarios are handled by the architecture, one can conclude that the reusability of the architecture is 80%. Both approaches, obviously, have their disadvantages. A disadvantage of the first approach is that it is generally impossible to define a complete set of scenarios. The definition of a representative set of scenarios is the weak point in the second approach since it is unclear how does one decide that a scenario set is representative.

Despite these disadvantages, scenarios are a useful technique for evaluating development NFRs. As Poulin [18] concluded for reusability, no predominant approach to assessing the quality attribute exists. Although scenario-based evaluation depends on the objectivity and creativity of the software engineers that define and execute them, in our projects we have not experienced this to be a major problem. In the next two sections, the scenarios used for evaluating reusability and maintainability are defined.

4.1.1 Reusability
The reusability quality attribute provides a balance between two properties: generality and concreteness. First, the architecture and its components should be general because they should be applied in other similar situations. For example, a weight sensor component can be sufficiently generic to be used in several applications. Secondly, the architecture should provide concrete functionality that provides considerable benefit when it is reused.

To evaluate the existing application architecture we use scenarios. However, reusability is a difficult software property to assess. We evaluate by analysing the architecture with respect to each scenario and assess the ratio of unchanged, slightly changed and new components. Note that the scenarios are presented as vignettes [13] for reasons of space.

R1. Product packaging quality control. For example, sugar packages that are both measured with respect to intact packaging and weight. This scenario requires multiple values to be measured about the item.

R2. Surface finish quality control where multiple algorithms may be used to derive a quality figure to make a decision from. Algorithms can be replaced at run-time.

R3. Quality testing of microprocessors where each processor is either rejected or are given a serial number and test data logged in a quality history database in another system. This scenario requires actions even for accepted products.

R4. Product sorting and labelling. For example parts are sorted after tolerance levels and labelled in several tolerance categories and are sorted in different storage bins.

R5. Printing quality assurance. For example, using pattern recognition to detect problems with printed results and where the system rejects occasional misprints but several misprints in a sequence might cause rejection and raising an alarm. This scenario uses historical data for the actuation strategy.

All presented scenarios require behaviour not present in the initial software architecture. However, the scenarios are realistic and measurement systems exist that require the functionality defined by the scenarios.
4.1.2 Maintainability

In software-intensive systems, maintainability is generally considered important. A measurement system is an embedded software system and its function is very dependent on its context and environment. Changes to that environment often inflict changes to the software system. The goal for maintainability in this context is that the most likely changes in requirements are incorporated in the software system against minimal effort.

Also maintainability of the DSSA is assessed using scenarios. For the discussion in this paper, the following scenarios are applied on the DSSA. Again, the scenarios are presented as vignettes for reasons of space.

M1 The types of input or output devices used in the system might change and the corresponding software needs to be updated. Modifications of this category should only affect the component interfacing the hardware device.

M2 New actuation devices might be added to a measurement system in order to support additional actions upon actuation. For example, in addition to removing the item from the conveyor belt, the system may also classify items in different categories and store them in different containers.

M3 One or more sensors of a measurement system may be replaced with sensors using a fundamentally different technology for measuring.

M4 Introduction of external systems interface to exchange data with other systems related to the process. For example, a database for logging the measurement data and subsequent analysis.

M5 Extending an existing measurement system with advanced fault isolation and correction to increase robustness.

These are the scenarios we have found to be representative for the maintenance of existing measurement systems. There are, of course, other changes may possibly be required are less likely to appear.

5 APPLYING THE METHOD

In this section the application architecture is transformed in an iterative process in order for it to fulfill the non functional requirements put on the DSSA. First, the NFRs for the application architecture is evaluated. Subsequently, the most prominent deficiency is identified and the architecture is transformed to remove the deficiency. This process is repeated until all quality attributes have satisfactory levels for the defined NFRs.

5.1 Application evaluation

Based on the defined scenarios and the initial architecture, the first design can be evaluated:

- Reusability is very poor in the initial architecture. The classes are tightly coupled and the reuse scenarios do not show any reuse possibilities without modifications to the components.
- Maintainability is slightly better, since actually one of the scenarios (M1) is satisfied. The input and output hardware devices may change their interfaces without necessarily affecting the other components. The other scenarios however, are not supported satisfactory.

The results of the analysis indicates that the reusability and maintainability attributes of the architecture are not satisfying. In the remainder of this section, transformations for improving these attributes are presented.

5.2 Transformations

5.2.1 Class level abstractions

Problem. Even though the beer can application has a small set of classes, the task of changing or introducing a new type of items or devices require the source code of most classes to be changed.

Alternatives. This problem is addressed by improving modularity of the system. Two alternatives exist. The first alternative is the most natural for object oriented designers, i.e. find the real-world correspondents of these classes and define relevant abstractions. Another alternative could be to decompose the system into a pipe&filter architecture with sensing, data interpreting and actuation filters.

Transformation. The first alternative is selected, since the second alternative requires changes to more than one filter for several scenarios. First, the measurement system may not always measure beer cans, but could also measure other items. The DSSA should be able to deal with the different kinds of items in a uniform way, and therefore the abstraction MeasurementItem is introduced.

Subsequently, abstractions for the RemoveLever and the LineCamera are defined. The abstraction defined for the RemoveLever is the abstract class Actuator. Different types of actuators can be defined as subclasses of class Actuator. Analogous to class Actuator, a class Sensor is defined. Class Trigger is defined as a specialized sensor. The redesigned architecture is presented in figure 4. In addition, the affected classes are marked with (1) in figure 5.

Problem Resolved. The abstractions introduced reduced the coupling between the classes by use of the principle programming towards an interface and not an implementation. One of the remaining problems, however, is that object instantiation is dependent of the actual types of sensors, actuators and measurement items.
5.2.2 Separation of object creation from trigger

**Problem.** The introduction of abstract types facilitates the introduction of several sub-types but does not remove the problem of type dependence at object creation. Every time a client creates an object it needs to know the actual sensor, actuator or measurement item types.

**Alternatives.** One can identify two alternatives to address this problem. The first is to introduce preprocessor macros in the source code to easily change all instances of object creation of the affected types. The second alternative is to use the Abstract Factory [9] to centralise the information about concrete types.

**Transformation.** There are a number of drawbacks with the macro alternative, for example, it is a static solution that to be changed must be recompiled. The ItemFactory could have an interface for changing the instantiation scheme. Therefore the Abstract Factory pattern is selected and a factory class is introduced to handle the instantiation of actuators, sensors and measurement items. See (2) in figure 5.

**Problem Resolved.** The trigger need no longer to know the actual type of measurement item to be created, but instead requests a new measurement item from the ItemFactory. The problem has now more satisfyingly been resolved.

---

**Figure 4. Model after first transformation**

5.2.3 Changing to strategies

**Problem.** Different classes in the measurement system perform similar tasks. Changes to these tasks require updating the source code of every class containing the same implementation for the task. For example, the measurement of a measurement item aspect could be performed by different methods; the measurement item may pull data from sensor, the sensor may push data to measurement item or the sensors may pass on data whenever they are changed. These alternatives are common to all sensors, independent of their types, and a way of reusing this aspect of the sensor is desired.

**Alternatives.** The Strategy pattern [9] allows reuse of common methods for multiple clients and it also allows the method to be changed dynamically if needed.

**Transformation.** The Strategy pattern is applied to all the instances where we can identify a similar need;

- **Sensor update strategy.** The three variations (push, pull, on change) apply to how sensors pass their data. The strategies are: OnChangeUpdate, ClientUpdate, and PeriodicUpdate.
- **Calculation strategy:** Calculation of derived data and decisions also show that there is similar methods to perform different tasks of different objects. Different calculation strategies can be defined. These strategies can be used by sensors, measurement items and actuators.

The newly introduced classes are marked with (3) in figure 5. The classes are organized in an inheritance hierarchy.

**Problem Resolved.** The implementation of different methods to update sensor values can now be reused and dynamically changed in a black box fashion. Since strategies are defined using the abstract interface of class Sensor, a strategy can acquire the values without knowing the actual sensor type. The above also holds for calculation strategies.

5.2.4 Unification of actuation and calibration

**Problem.** Calibration is performed by letting the system measure a manually inspected item and the values are stored as ideal. In normal service, deviations from the ideal values measured are considered faults. The calibration are supposed to be carried out at different times in run-time. Calibration of the measurement system is very much considered a special service that is not implemented using the concepts of the redesigned architecture. This makes the system more tightly coupled, in the sense that the factory, the measurement item and the sensors need to be aware of the calibration state. This is inhibiting the reuse and maintenance properties.

**Alternatives.** Two alternatives to decreasing the coupling due to calibration exist. First, the calibration is defined as a separate service that all sensors and measurement items must implement. Each entity checks if its current state is calibration and then call its calibration services instead of the normal services. The second alternative is to centralise the calibration behaviour in a single entity, influencing as few entities as possible. This can be achieved by defining calibration as an actuation strategy.

**Transformation.** The first alternative is selected since calibration can be viewed as a kind of actuation behaviour, the standard actuation behaviour should be replaced by calibration behaviour whenever the system is in calibration mode.
Consequently, the actuation behaviour of a measurement item is transformed into a separate entity through the use of the Strategy pattern. This allows for run-time replacement of the actuation behaviour. See (4) in figure 5.

**Problem Resolved.** The use of the calibration strategy as a special type of actuation removes the knowledge about calibration from all entities except the factory class. Since this required the actuation to be separated from the measurement item class, (run-time) changes to the actuation do no longer affect the measurement item.

5.2.5 Adding prototype to factory

**Problem.** The introduction of the calibration strategy addressed most of the identified problems. However, there is a problem remaining, i.e. the calibration strategy is dependent on the concrete class of the measurement item. This couples these two classes, so that the calibration strategy cannot be reused separate from the measurement item.

**Alternatives.** One alternative solution is to decrease the coupling by introducing an intermediate data storage, e.g. a configuration file, where the ideal values are stored. The second alternative is to store the ideal measurement item in the ItemFactory and to use it as a template for subsequent measurement item instances. This is known as the Prototype pattern [9].

**Transformation.** We decide to apply a variant of the prototype pattern. The calibration of the system is performed by setting the ItemFactory into calibration mode. When a real-world item triggers the itemFactory, it creates a new measurement item with the normal sensors associated. The measurement item collects the data by reading the sensors, but when invoking the actuation strategy, instead of invoking the actuators, the calibration strategy causes the measurement item to store itself as a prototype entity at the item factory. The affected relation is marked (5) in figure 5.

**Problem Resolved.** The Prototype pattern was applied to the ItemFactory. Consequently, the calibration strategy no longer needs to know the concrete class of the measurement item. The only class in the system with knowledge about concrete class identifies is the ItemFactory. The information has been localized in a single entity.

<table>
<thead>
<tr>
<th>Non-functional Property</th>
<th>Iteration no.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Reusability</td>
<td>R1</td>
</tr>
<tr>
<td>R2</td>
<td>-</td>
</tr>
<tr>
<td>R3</td>
<td>-</td>
</tr>
<tr>
<td>R4</td>
<td>-</td>
</tr>
<tr>
<td>R5</td>
<td>-</td>
</tr>
<tr>
<td>Maintainability</td>
<td>M1</td>
</tr>
<tr>
<td>M2</td>
<td>-</td>
</tr>
<tr>
<td>M3</td>
<td>-</td>
</tr>
<tr>
<td>M4</td>
<td>-</td>
</tr>
<tr>
<td>M5</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1: Analysis of architecture

5.3 Final Assessment

In table 1, the results from the final assessment of the domain specific software architecture for measurement systems is presented. The DSSA itself is shown in figure 5. As shown in the evaluation, the scenarios R5 and M5 are not supported by the DSSA. However, the conclusion from the assessment was that the DSSA fulfilled its requirements sufficiently.
6 RELATED WORK

To the best of our knowledge, no architecture reengineering methods exist to date. Traditional design methods often focus on the system functionality [2,11,21,26] or on a single non-functional requirement, e.g. high-performance [25], real-time [15] and reusable systems [12]. Architecture design methods have been proposed by Shlaer & Mellor [24], Kruchten [14] and Bosch & Molin [6]. Different from architecture reengineering, architecture design methods start from the requirement specification only. Boehm [3] discusses conflicts between, what he calls, quality requirements, but focuses on identifying these conflicts and solving them during requirement specification rather than during architectural design or reengineering.

Architecture evaluation is discussed by Kazman et al. [13]. Their SAAM method also uses scenarios, but does not discuss other techniques for architecture evaluation. In addition, no relation to architecture (re)design is made. Also the work described in [1] primarily uses scenarios for architecture evaluation.

Architecture transformation uses the notions of architectural styles [23], architectural patterns [7] and design patterns [9]. However, rather than viewing these concepts as ready designs, we treat styles and patterns as active entities transforming an architecture from one version to another.

7 CONCLUSIONS

A method for reengineering software architectures has been presented that provides a practical approach to evaluating and redesigning architectures. The method uses four techniques for architecture evaluation, i.e. scenarios, simulation, mathematical modelling and objective reasoning. To improve the architecture, five types of architecture transformations are available, i.e. impose architectural style, apply architectural pattern, use design pattern, convert non-functional requirements (NFRs) to functionality and distribute NFRs. The method was illustrated using a concrete system from the measurement systems domain, i.e. a beer can inspection system. This system was reengineered into a domain-specific software architecture for measurement systems. The focus in this paper was on the reusability and maintainability requirements on the DSSA. Scenarios were defined for assessing each requirement. Although no predominant approach to assessing quality attributes of a systems exists, this paper shows that scenarios provide a powerful means for practitioners. Architecture transformations provide an objective way to redesign since associated with each transformation, a problem is explicitly addressed, alternative transformations are investigated and the rationale for the design decisions is captured.

Future work includes the extensions of the reengineering method for more non-functional requirements and the application of the method in more industrial case studies and projects.

References


