ASAAM: Aspectual Software Architecture Analysis Method

Bedir Tekinerdoğan
Department of Computer Science, University of Twente,
P.O. Box 217 7500 AE Enschede, The Netherlands
bedir@cs.utwente.nl

Abstract

Software architecture analysis methods aim to predict the quality of a system before it has been developed. In general, the quality of the architecture is validated by analyzing the impact of predefined scenarios on architectural components. Hereby, it is implicitly assumed that an appropriate refactoring of the architecture design can help in coping with critical scenarios and mending the architecture. This paper shows that there are also concerns at the architecture design level which inherently crosscut multiple architectural components, which cannot be localized in one architectural component and which, as such, can not be easily managed by using conventional abstraction mechanisms. We propose the Aspectual Software Architecture Analysis Method (ASAAM) to explicitly identify and specify these architectural aspects and make them transparent early in the software development life cycle. ASAAM introduces a set of heuristic rules that help to derive architectural aspects and the corresponding tangled architectural components from scenarios. The approach is illustrated for architectural aspect identification in the architecture design of a window management system.

Keywords
aspect-oriented software architecture design, scenario-based aspect-identification, scenario based architectural evaluation.

1. Introduction

Software architecture forms one of the key artifacts in the entire software development life cycle since it embodies the earliest design decisions and includes the gross-level components that directly impact the subsequent analysis, design and implementation [4][2]. Accordingly, it is important that the architecture design supports the software system qualities required by the various stakeholders. For ensuring the quality factors the common assumption is that identifying the fundamental concerns for architecture design is necessary and various architecture design methods have been introduced for this purpose [16]. To verify that the right concerns have been identified generally static analysis of formal architectural models are applied [13] or a set of architecture analysis methods as described in [7] are adopted. In this paper we focus on software architecture analysis methods that utilize scenarios for evaluating architectures. In general, these analysis methods take as input the architecture design and measure the impact of predefined scenarios on it in order to identify the potential risks and the sensitive points of the architecture. This helps to predict the quality of the system before it is built, thereby reducing unnecessary maintenance costs.

A scenario is considered to be a brief description of some anticipated or desired use of the system [1]. Scenarios that can be directly supported by the architecture(s) are called *direct scenarios*. Scenarios that require the redesign of the architecture are called *indirect scenarios*. In that case the software architecture design needs to be refactored to turn the indirect scenarios into direct scenarios. In that case generally it is tacitly assumed that the redesign of the software architecture design can be mostly successful.

In this paper we argue that some concerns, even at the architectural design level, can not be easily localized and specified in individual architectural components. Similar to the notion of aspect at the programming level, we say that these concerns are crosscutting and denote so-called *architectural aspects*. Since the crosscutting property of architectural aspects is inherent we claim that these cannot be undone simply by redefining the software architecture using conventional architectural abstractions. In fact, we believe that like various aspect-oriented programming abstractions [8][12] we need explicit mechanisms to identify, specify and evaluate aspects at the architecture design level.

Current software architecture analysis methods do not make an explicit distinction between conventional architectural concerns that can be localized using current architectural abstractions and architectural concerns that crosscut multiple architectural components. The risk is that potential crosscutting concerns might not be detected

as aspects and as such remain unsolved at the design and programming level. This may lead to tangled code in the system and consequently the quality factors that the architecture analysis methods attempt to verify will still be impeded.

To overcome this issue, we propose the Aspectual Software Architecture Analysis Method (ASAAM) as an approach for evaluating the architectural aspects in the available software architecture design. ASAAM builds on the Software Architecture Analysis Method (SAAM) as described in [1] and [11]. ASAAM is complementary to SAAM and includes explicit mechanisms for identifying architectural aspects and the related tangled components.

The remainder of this paper is organized as follows. In section 2 a short overview of the software architecture analysis method, SAAM will be given. In section 3 we present a running example, the design of a Window Management System. In section 4 we describe the steps of ASAAM to explicitly identify architectural aspects. Finally, in section 5 we provide our conclusions.

2. Software Architecture Evaluation

In [7] a comprehensive survey is given of the various software architecture design analysis methods that have been proposed so far. Among these methods the Software Architecture Analysis Method (SAAM) can be considered as a mature method which has been validated in various cases studies. Other methods such as SAAMCS, ESAAMI, SAAMER and ATAM are based on or adopt the concepts used in this method [7]. The Aspectual Software Architecture Analysis Method (ASAAM) in this paper also defines an extension and refinement to SAAM. Before explaining ASAAM we will describe the steps of SAAM first. The basic activities of SAAM are illustrated in Figure 1.

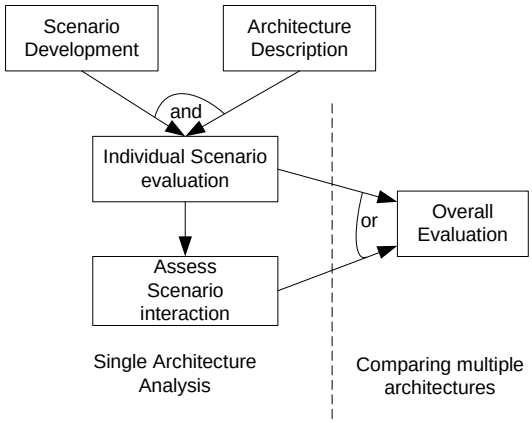

Figure 1. SAAM inputs and activities [1]

SAAM takes as input a problem description, requirements statement and architecture descriptions. The steps of SAAM are as follows [11]:

1. Describe candidate architecture: The candidate architecture is described which includes the system's computation and data components, as well as all component relationships, sometimes called connectors.

2. Develop scenarios: Development of scenarios for various stakeholders; the scenarios illustrate the kinds of activities the system must support and the anticipated changes that will be made to the system over time.

3. Perform scenario evaluations: Scenarios are categorized into direct and indirect scenarios. For each indirect task scenario the required changes to the architecture are listed and the cost of performing these changes is estimated. A modification to the architecture means that either a new component or connection is introduced or an existing component or connection requires a change in its specification.

4. Reveal scenario interaction: Different indirect scenarios that require changes to the same components or connections are said to *interact* at the corresponding component. Determining scenario interaction is a process of identifying scenarios that affect a common set of components. Scenario interaction measures the extent to which the architecture supports an appropriate separation of concerns. Semantically close scenarios should interact at the same component. Semantically distinct scenarios that interact indicate an improper decomposition.

5. Overall evaluation: Finally, each scenario and the scenario interactions are weighted in terms of their relative importance and this weighting used to determine an overall ranking. The weighting chosen will reflect the relative importance of the quality factors that the scenarios manifest.

3. Example: Window Management System

In this section we will provide an example of the design of a window management system architecture, which we will use in subsequent sections to present the ASAAM.

A window management system is a type of interactive user interface that enables users to work with multiple separate applications at the same time[1]. This is achieved through the use of a desktop metaphor in which each process is associated with a graphical window. A window management system provides the functionality to create and manipulate the display of multiple processes.

[1] Inspired from example in [11]

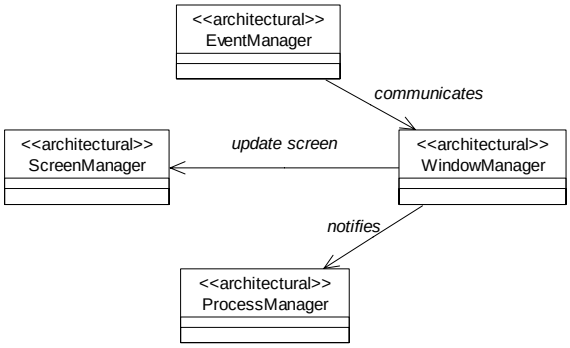

Figure 2. Top-level architecture of window management system.

A window management system includes several major components such as *EventManager* for controlling I/O, e.g. keyboard and mouse events; *Process-Manager* for scheduling and managing application processes; *ScreenManager* for maintaining the integrity of the screen; and *WindowManager* for managing the windows that are related to the application processes. The top-level architecture design of the window management system is given in Figure 2. *EventManager* in this architecture handles only keyboard and mouse input. *ProcessManager* includes functionality to manage multiple processes. *EventManager* communicates the events to *WindowManager* which notifies *ProcessManager*. Depending on the event *ProcessManager* can activate, delay or kill application processes. Events that are related to the updating of the screen notify the *ScreenManager*.

By applying a scenario-based software architecture analysis method, such as SAAM, we can now identify, for example, the following direct and indirect scenarios for the given architecture in Figure 2:

Direct Scenarios:

S1. Start multiple processes at the same time.
S2. Change color of widgets in a window.
S3. Close all open windows
S4. Change screen resolution
S5. Enter a command to start application process
S6. Move the main window
S7. Screen saver is activated
S8. Resize a window
S9. Terminate a process
S10. Interrupt a process

Indirect Scenarios:

S11. Change look-and-feel style at run-time.
S12. Add voice control
S13. A failure occurs and the system shuts down.
S14. Provide dual display screen.
S15. Use multiple desktops.
S16. Monitor activities of the user
S17. Provide touch screen and light pen as input
S18. A memory overflow due to too many opened windows
S19. Port system to command-based operation system
S20. Minimize windows after idle time

This means that scenario S1 to S10 can be directly performed by the given architecture. Scenarios S10 to S20 require some architectural changes to perform the required functionality. As described before, this can consist of a change to the functionality of one or more architectural components, the addition of an architectural component to perform new functionality, the addition of an architectural relation between the architectural components, or a combination of these changes [11].

Table 1 shows the direct and indirect scenarios together with the scenario interactions at each component. Direct scenarios are used to understand a component's complexity. Indirect scenario interactions are considered as good if they are semantically close. This shows the cohesiveness of the component. In case the scenarios are semantically distinct then this either means that the component needs to be composed to fulfill subsets of semantically close scenarios or it is assumed that the component is badly designed and a critical refactoring of the architecture is needed.

Table 1. Scenario Interactions for Window Manager architecture

Component	Direct Scenarios	Indirect Scenarios
EventManager	S5	S12,S13,S16,S17,S19
ScreenManager	S4, S7	S13, S14,S19
WindowManager	S2, S3, S6, S8	S11, S15, S16, S18, S19, S20
ProcessManager	S1, S9, S10	S13, S16,S19

We argue that the architecture may include potential architectural aspects and these should be explicitly and distinctly considered. However, since the state-of-the-art architecture analysis methods do not consider architectural aspects it is not possible to detect these at the architecture design level. Since potential aspects will not disappear by themselves these will still pop up later in the detailed design and programming level. Consequently this might easily lead to the known problems such as scattered concerns and tangled code, thereby increasing complexity and introducing maintenance problems. Within the context of software architecture analysis methods, this actually means that the verification of the concerns is not complete and as such the desired quality factors cannot be appropriately predicted. Aspects should be considered as first class

concerns and be identified right at the architectural design level [17].

4. Approach for architectural aspect identification

In this section we will extend and refine the steps in the SAAM to provide a method that can also identify architectural aspects using scenarios. There are several similarities between aspects and scenarios. First, aspects are crosscutting concerns, that is, concerns that interact over many components. In the same way, (indirect) scenarios also require changes to several components and can be said to 'crosscut' components. Second, aspects are relative to the problem description and the given decomposition of the design. Similarly, the categorization of scenarios into direct and indirect scenarios is completely dependent on the problem description and the given architecture design. Based on these observations we think that scenarios provide potential architectural aspects.

The basic activities for the ASAAM are illustrated in Figure 3.

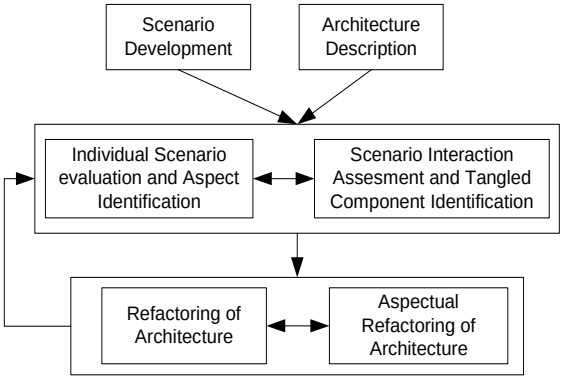

Figure 3. Activities for ASAAM

ASAAM, like the SAAM, takes as input a problem description, a requirements statement and architecture descriptions. In addition to developing a candidate architecture and scenarios, ASAAM comprises the following steps:

1. *Candidate architecture development*

   A (candidate) architecture design is provided that will be analysed with respect to the required quality factors and potential aspects.

2. *Develop scenarios*

   This is similar to SAAM. Scenarios from various stakeholders are collected, which represent both important uses and anticipated uses of the software architecture.

3. *Individual scenario evaluation and aspect identification*

   Scenarios are firstly categorized into direct and indirect scenarios. Complementary to SAAM, the scenario evaluation also searches for potential architectural aspects. The application of the heuristic rules will result in a further classification of the scenarios into *direct scenarios*, *indirect scenarios*, *aspectual scenarios* and *architectural aspects*. *Aspectual scenarios* are derived from direct or indirect scenarios and represent potential aspects. By *aspect domain analysis* the corresponding aspect for the scenario is searched and described.

4. *Scenario interaction assessment and component classification*

   The goal of this step is to assess whether the architecture supports an appropriate separation of concerns. This includes both non-crosscutting concerns and crosscutting concerns, i.e. aspects. For each component both direct and indirect components are analyzed and categorized into *cohesive component*, *tangled component, composite component*, or *ill-defined component*.

5. *Refactoring of architecture*

   Based on the scenario interaction assessment and component classifications a refactoring of the architecture is proposed. The refactoring can be done using conventional abstraction techniques, such as design patterns, or using aspect-oriented techniques. The architectural aspects and the tangled components are explicitly described in the architecture.

In the following we will describe each of these steps in more detail by giving explicit heuristic rules and applying this to the given example.

4.1 Individual Scenario Evaluation and Aspect Identification

SAAM considers the set of scenarios to be complete when the addition of a new scenario no longer disturbs the architecture design. However, there is a risk that for a given set of scenarios this process will never be completed, that is, the architecture will need to be disturbed all the time. This may even the case if all the architectural components in the system are modular. The reason why the set of scenario analysis will not be completed is the fact that a scenario might include a potential architectural aspect. As we have described before, an architectural aspect is a concern that crosscuts across multiple architectural components.

In the ASAAM we have defined a set of heuristic

rules to categorize scenarios into direct scenarios, indirect scenarios and architectural aspects. The heuristic rules are expressed using conditional statements in the following form [18]:

IF <condition> THEN <consequent>

The condition part includes an artifact that is analyzed. In this context, an artifact is a work product in the software architecture design process. The basic artifacts in ASAAM are as follows: ARCHITECTURE, PROBLEM DESCRIPTION, SCENARIO, ASPECT, and COMPONENT. As we will see, each of these artifacts is further specialized into intermediate representations to support the software engineer in processing the heuristic rules of ASAAM. The consequent part of each rule includes an action, which usually includes a transformation of an artifact to another artifact.

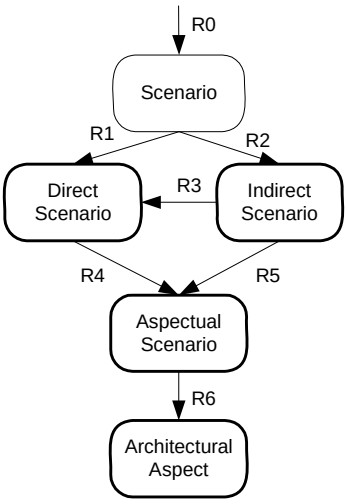

Figure 4. Scenario artifact diagram

Figure 4 shows the *scenario artifact diagram* of ASAAM, which defines the relations among the heuristic rules and the scenario artifacts. The rounded rectangles represent artifacts, the labeled arrows the heuristic rules. The bold rounded rectangles represent the artifacts that are finally delivered. In the individual scenario analysis and aspect identification process the delivered artifacts are the following:

1. *Direct Scenario*, describes a scenario that can directly perform the provided scenarios.
2. *Indirect Scenario*, describes a scenario which requires some changes to the component that it interacts with.
3. *Aspectual Scenario*, a direct or indirect scenario which is scattered over different architectural components and which cannot be captured in a single component.
4. *Architectural Aspect*, a well-defined concern transformed from an aspectual scenario based on the domain models derived from a domain analysis process.

The heuristic rules for scenario evaluation in ASAAM are shown in Figure 5. The rules are numbered from R0 to R6 and include the manipulation of the artifacts shown in the scenario artifact diagram in Figure 4 .

R0:
Develop SCENARIO artifacts based on PROBLEM DESCRIPTION

R1:
IF SCENARIO does not require any changes to architectural description
THEN SCENARIO becomes DIRECT SCENARIO

R2:
IF SCENARIO requires changes to one or more ARCHITECTURAL COMPONENTs
THEN SCENARIO becomes INDIRECT SCENARIO

R3:
IF INDIRECT SCENARIO can be resolved after refactoring
THEN INDIRECT SCENARIO is DIRECT SCENARIO

R4:
IF DIRECT SCENARIO is scattered and cannot be localized in one component
THEN DIRECT SCENARIO is ASPECTUAL SCENARIO

R5:
IF INDIRECT SCENARIO is scattered and cannot be localized in one component
THEN INDIRECT SCENARIO is ASPECTUAL SCENARIO

R6:
Derive ARCHITECTURAL ASPECT from ASPECTUAL SCENARIO

Figure 5. Heuristic rules for scenario evaluation

Rule R0 defines the scenario development from the problem description. The result of this rule is typically the set of scenarios as defined in section 3. Rules R1 and R2 categorize the scenarios into direct and indirect scenarios. Rule R3 states that an indirect scenario can become direct in case the architecture can be appropriately refactored.

Rule R4 and R5 are used to identify *aspect scenarios* that are scenarios for which the required changes are scattered over many architectural components, and which can not be just easily localized into one component. For example, in the window management system architecture, by applying rule R5, the scenarios S13 (failure occurs), S16 (monitor user activities), and S19 (port to command-based operating system) interact at various components, they are scattered over many

components and it is difficult to localize these in one component, since they are inherently crosscutting. ASAAM explicitly identifies these scenarios as aspectual scenarios. As such in ASAAM a third category that represents architectural aspects for scenarios is introduced:

Aspectual Scenarios
S13. A failure occurs and the system shuts down.
S16. Monitor activities of the user
S19. Port system to command-based operation system

In rule R6 these aspectual scenarios are more thoroughly inspected by analyzing the related domain(s) using domain analysis [2], and a corresponding architectural aspect is identified. For the above scenarios we might come up with architectural aspects of *Failure Management Aspect, Monitoring Aspect* and *OperatingSystem Aspect.* By doing so, the software engineer may anticipate on these during the detailed design and programming phases.

To guide the software engineer in processing these heuristic rules in ASAAM we order and specify these rules as follows:

```
R0; R1||R2; R3; R4||R5; R6
```

Here ';' is used to denote sequence, '||' defines the parallel activity. In this case, rule R0 is executed first, followed by R1 and/or R2 which may be processed in parallel, followed by R3, followed by R4 and/or R5 in parallel, and finally R6.

4.2 Scenario Interaction Assessment and Tangled Components Identification

After scenarios have been classified ASAAM focuses on scenarios at individual components. Figure 6 shows the *component artifact diagram* in ASAAM which shows the relation between the various corresponding heuristic rules and the intermediate forms of component artifacts. ASAAM delivers the following type of components (bold rounded rectangles in the figure):

1. *Cohesive Component*, which is a component that is well defined and performs semantically close scenarios.
2. *Composite Component*, a component consisting of several sub-components that each perform a semantically close set of scenarios.
3. *Tangled Component*, a component that performs an aspectual scenario which is either directly or indirectly performed by the component.
4. *Ill-defined Component,* a component that includes semantically distinct scenarios but which cannot be decomposed or does not include an aspectual scenario.

Further, the artifacts *Component,* and *Tentative Tangled Component* represent the intermediate artifacts that help to identify the above four types of components. The heuristic rules R7 to R18 are shown in Figure 7.

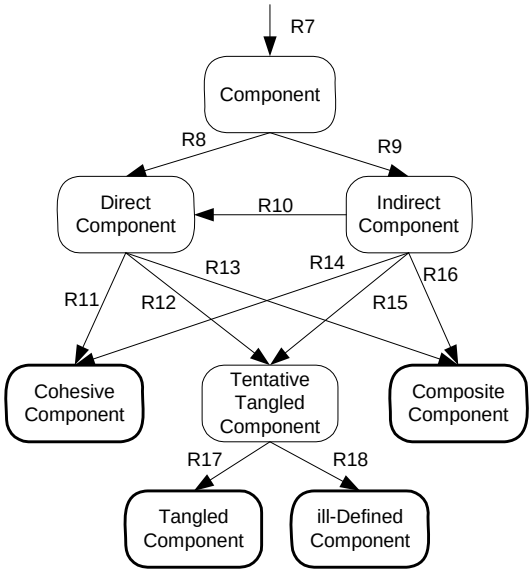

Figure 6. Component artifact diagram

Rule R7 lists the architectural components. Typically, these are the four architectural components as provided in Figure 2.

Rules R8 and R9 categorize the components into direct and indirect components to distinguish between components that perform direct scenarios and indirect scenarios, respectively. The characterization of the component as direct and indirect is relative to the scenarios. The component can be direct for one set of scenarios and indirect for another set. This is explicitly described as shown in Table 1.

Rule R10 defines the situation in which an indirect component can be refactored to perform an indirect scenario. In this case the indirect scenario becomes direct and as such the component will be direct. For example, scenario S12 (adding voice control) is not supported by the *EventManager*. However, the functionality of *EventManager* can be rather easily adapted to also accept voice input so that S12 can become direct.

Rules R11, R12 and R13 analyze the direct components and decide whether they perform either semantically close scenarios or semantically distinct scenarios.

In case of semantically close scenarios (R11) the component is identified as a cohesive component. This covers scenarios S3, S6 and S8, for example, which are all window management operations and can be directly supported by *WindowManager*. We say that *WindowManager* is a cohesive component with regard to scenarios S3, S6 and S8.

If the performed scenarios are semantically distinct but if the component can be decomposed into sub-components in which each subcomponent performs a set of semantically close scenarios then the component is identified as a composite component (R13). This is, for example the case for scenario S4, changing the screen resolution, and scenario S7, activating a screen saver; both are performed by *ScreenManager* but can be regarded as semantically distinct scenarios. Nevertheless if we assume that these functionalities are provided by separate subcomponents in *ScreenManager* in the current architecture, then *ScreenManager* is a composite component regarding scenarios S4 and S7. If the semantically distinct set of scenarios can not be decomposed into sub-components to eliminate the semantically distinct behavior then rule R12 states that the component is a *tentative tangled component*. A tentative tangled component can eventually become a *tangled component* or an *ill-defined component*, as we will explain shortly.

Rules R14, R15 and R16 define the transformation from an indirect component to either a cohesive component, a tentative tangled component or a composite component. In essence, the rules are similar to rules R10, R11 and R12 for direct components, except the transformations take place from an indirect component. Of particular interest is rule R15 which concerns the identification of tangled components from indirect components. For example, consider the indirect scenario S19 which concerns porting the window management system to a different operating system platform. This scenario requires changes to all the components in the architecture and cannot be localized. It will also interact with the scenarios in the corresponding components. As a result the affected components become tentative tangled components of this aspectual scenario.

Note that in the case of direct components the architecture does not require to be changed, since it already fulfills all the related scenarios. The issue here is that using the above rules we can detect whether the component is still a tangled component or composite component, despite the fact that it might accidentally perform the scenarios. In case of indirect scenarios the change is always required.

R7:
Select COMPONENT from ARCHITECTURE

R8:
IF COMPONENT is not affected by a SCENARIO
THEN component is DIRECT COMPONENT for SCENARIO

R9:
IF COMPONENT is affected by one or more SCENARIO
THEN component is INDIRECT COMPONENT for SCENARIO

R10
IF INDIRECT COMPONENT can be refactored to perform INDIRECT SCENARIO
THEN INDIRECT COMPONENT is DIRECT COMPONENT

R11
IF DIRECT COMPONENT performs semantically close scenarios
THEN DIRECT COMPONENT is COHESIVE COMPONENT

R12
IF DIRECT COMPONENT performs semantically distinct scenarios
AND cannot be decomposed
THEN DIRECT COMPONENT is TENTATIVE TANGLED COMPONENT

R13
IF DIRECT COMPONENT performs semantically distinct scenarios
AND can be decomposed
THEN DIRECT COMPONENT is COMPOSITE COMPONENT

R14:
IF INDIRECT COMPONENT includes semantically close scenarios
THEN INDIRECT COMPONENT is COHESIVE COMPONENT

R15:
IF INDIRECT COMPONENT includes semantically distinct scenarios
AND cannot be decomposed
THEN COMPONENT becomes TENTATIVE TANGLED COMPONENT

R16:
IF INDIRECT COMPONENT includes semantically distinct scenarios
AND can be decomposed
THEN INDIRECT COMPONENT is COMPOSITE COMPONENT

R17:
IF TENTATIVE TANGLED COMPONENT includes ASPECTUAL SCENARIO
THEN TENTATIVE TANGLED COMPONENT is TANGLED COMPONENT for given aspectual scenario

R18:
IF TENTATIVE TANGLED COMPONENT does not include ASPECTUAL SCENARIO
THEN TENTATIVE TANGLED COMPONENT is Ill-DEFINED COMPONENT

Figure 7. Heuristic rules for identifying tangled components.

Rule R17 and R18 inspect the tentative tangled component on aspectual scenarios that might have been identified in the individual scenario evaluation and aspect identification process. In case an aspectual scenario is involved (R17) then the tentative tangled component becomes a tangled component of the architectural aspect that is derived from the corresponding aspectual scenario. Otherwise (R18) the tentative tangled component becomes an ill-defined

component, which means that the component includes semantically distinct scenarios because of an improper decomposition of the component.

The rule ordering for the scenario interaction assessment and tangled component identification can be specified as follows:

```
  R7; (R8||R9); R10;
((R11||R12||R13); || (R14;R15;R16)); R17
```

After executing these heuristic rules for each component we can identify the related aspects and characterize the components more specifically. The results are shown in Table 2. This table provides a guideline for the refactoring of the architecture. The column *Cohesive* lists the direct and indirect scenarios that show the cohesiveness of the corresponding component. The column *Aspect* lists the set of aspectual scenarios that are tangled in the component. Finally, column *Ill-def.* lists the scenarios that the architectural component can not cope with at all.

Table 2. Characterization of Components

Component	Cohesive	Aspect	Compos.	Ill.def
EventManager	S5	S13,S16,S19	S12,S17	-
ScreenManager	S14	S13,S19	S4,S7	-
WindowManager	S2,S3,S6, S8,S20	S16,S19	S11,S18,S15	-
ProcessManager	S1,S9,S10	S13,S16,S19		-

4.3 Architectural Aspect Specification

The previous two sections have described the steps of ASAAM for identifying architectural aspects and the related tangled components. This information will be used to redesign the given architecture in which the architectural aspects are made explicit. This can be basically done in two ways: (1) using a software architecture description language [13] or (2) using a visual modeling language to represent aspects in the architecture. The latter is usually based on extensions of UML [5]. We will not elaborate on this issue due to space limitations.

As an example, in Figure 8 we show a UML diagram of the refactored architecture which explicitly includes the architectural aspect derived from the aspectual scenario S13 (failure occurs and system shuts down). We have used stereotypes, the built-in extension mechanism of UML [5]. Architectural components are identified using the stereotype <<architectural>>. Architectural aspects are represented using the sterotype <<arch-aspect>>. To represent the relations with the tangled components we represent <<pointcut>>. This stereotype identifies, similar to the concept used in AspectJ [12], a *pointcut designator* referring to the set of components that the aspect crosscuts. In this case the pointcut is called *recover()* which has as join points the architectural components *EventManager*, *ScreenManager* and *ProcessManager*. At this phase we do not specify other aspect issues such as the related advice. These might be explored during the detailed design and implementation of the system.

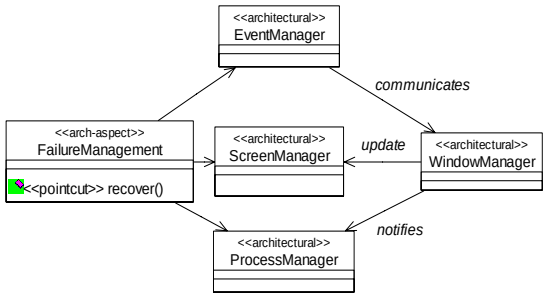

Figure 8. Window management system architecture after aspectual refactoring

5. Related Work

Various architecture analysis methods have been proposed in the literature. A comprehensive overview of these architecture analysis methods is given [7]. In this paper we have basically adopted SAAM as a representative for scenario-based analysis methods. The reason for this is that SAAM includes a core set of steps that is adopted by most of these methods. Further SAAM has been validated in various case studies. ASAAM extends and refines SAAM to include steps for architectural aspect identification. As such, we think that ASAAM also provides complementary techniques to the other software architecture analysis methods.

In [14] an approach is proposed for modularizing and composing crosscutting aspectual concerns at the requirements analysis phase. The approach is based on separating the specification of aspectual requirements and non-aspectual requirements. Composition rules are defined which specify how aspectual requirements impact non-aspectual requirements. The approach helps to identify early trade-offs between aspectual requirements and support the decision of the various stakeholders in the requirements engineering process. This approach can be considered complementary to ASAAM, and vice versa, since it focuses on requirements engineering, whereas ASAAM focuses more on the architecture design.

In our earlier work we have stated that aspects should

be identified at the requirements analysis and software architecture design phases [17]. In addition we have thereby stated that aspects should be derived from domain models. This work on ASAAM is an extension to our previous ideas. The novelty here is that we apply scenarios to guide the identification of aspects and the related domain analysis to aspects.

Aspectual design is emerging and we think that ASAAM can benefit from the various proposals such as described in [6][9][10] and [15]. In [6], composition patterns are introduced which use UML templates to extend UML for aspect-oriented design. In [9] examples of architectural capture of aspects and their representation using UML is provided. In [10] an approach is described for aspect-oriented component engineering techniques together with the corresponding notations. In [15] a UML-based Aspect-Oriented Design notation is provided for AspectJ as the target implementation language. The ideas in these publications on aspect modeling might be used in the architectural aspect specification process as described in section 4.3.

6. Conclusion

The contribution of this paper is twofold. First of all we have shown that several architectural concerns crosscut multiple architectural components. That is to say, *architectural aspects* exist. For example in the window management system example presented in section 4 we have seen that the *Failure Management Aspect, Monitoring Aspect* and *OperatingSystem Aspect* are inherently crosscutting concerns.

Secondly, we have defined the method ASAAM which is a systematic scenario-based architecture analysis method that is both able to identify concerns that can be easily localized and specified in architectural abstractions, and identify concerns that crosscut various architectural components. The method includes a set of heuristics for identifying these architectural aspects. For this, scenarios have been classified into *direct scenario, indirect scenarios, aspectual scenarios,* and *architectural aspects*. In addition we have provided a detailed analysis of scenario interactions and provided a characterization of the various components into *cohesive component, composite component, tangled component,* and *ill-defined components.*

We have illustrated our ideas for evaluating the architecture of window management system architecture and identified the architectural aspects of *failure management*, *monitoring* and *operating system* aspects. We have discussed that these aspects should be made explicit at the architecture design level so as to prepare it for aspect-oriented design and aspect-oriented programming. In addition we have proposed a simple way of specifying architectural aspects, or pointed to using existing more advanced aspect modeling techniques.

ASAAM builds on scenario-based architecture analysis methods, and as such, should be considered as a complementary approach to these methods. The benefit of ASAAM is due to the systematic support for the management of architectural aspects in an explicit manner. In this paper we have applied ASAAM for a single structure of an architecture. In fact, architectures are composed of several different structures, each focused on a single concern or set of related concerns, as described in [4], and analysis must take these different structures into account. We will focus on applying ASAAM to multiple structures of the architecture in our future work.

Acknowledgements

I would like to thank the anonymous reviewers for their comments and suggestions. This research has been carried out in the *Aspect-Oriented Software Architecture Design project* which is funded by the Dutch Scientific Organisation in the *Jacquard Software Engineering Program*.

References

[1] G. Abowd. *Analyzing Development Qualities at the Architecture Level: The Software Architecture Analysis* Method. in: L. Bass, P. Clements, and R. Kazman (eds.). *Software Architecture in Practice*, Addison-Wesley 1998.

[2] M. Aksit (Ed.), *Software Architectures and Component Technology: The State of the Art in Research and Practice*, Kluwer Academic Publishers, 2001.

[3] G. Arrango. *Domain Analysis Methods*. In Software Reusability, Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, pp. 17-49, 1994.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*, second edition, Addison-Wesley 1998.

[5] G. Booch, J. Rumbaugh & I. Jacobson. *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[6] S. Clarke, R.J. Walker. *Composition Patterns: An Approach to Designing Reusable Aspects.* In proceedings of the 23rd International Conference on Software Engineering (ICSE), Toronto, Canada, May 2001.

[7] L.Dobrica & E.Niemela. A survey on software architecture analysis methods. IEEE Trans. on Software Engineering, Vol. 28, No. 7, pp.638-654, July 2002.

[8] T. Elrad, R. Fillman, & A. Bader. *Aspect-Oriented Programming.* Communication of the ACM, Vol. 44, No. 10, October 2001.

[9] J. Grundy & R. Patel. *Developing Software Components with the UML*, Enterprise Java Beans and Aspects, In Proceedings of the 2001 Australian Software Engineering Conference, Canberra, Australia, 26-28 August 2001.

[10] J. Grundy. *Multi-perspective specification, design and implementation of software components using aspects*, International Journal of Software Engineering and Knowledge Engineering, Vol. 10, No. 6, December 2000, pp. 713-734.

[11] R.Kazman, G.Abowd, L.Bass & P.Clements. *Scenario-Based Analysis of Software Architecture.* IEEE Software, pp. 47-55, Nov. 1996.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. *An Overview of AspectJ*. In J. Lindskov Knudsen (ed.), ECOOP 2001 Object-Oriented Programming 15th European Conference, Budapest Hungary, pages 327-353. Volume 2072 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, June, 1997.

[13] N.Medvidovic & R.N. Taylor. *A classification and comparison framework for Software Architecture Description Languages,* IEEE Trans. on Software Engineering, Vol. 26, No.1 pp. 70-93, 2000..

[14] A. Rashid, A. Moreira, J. Araujo. *Modularisation and Composition of Aspectual Requirements*. In proceedings of Second Aspect-Oriented Software Development Conference, Boston, pp. 11-20, March, 2003.

[15] D.Stein, S. Hanenberg & R. Unland. *A UML-based Aspect-Oriented Design Notation for AspectJ*, in Proceedings of First Aspect-Oriented Software Development Conference, Enschede, The Netherlands, April, 2002.

[16] B. Tekinerdoğan & M. Akşit. *Classifying and Evaluating Architecture Design Methods*, in: Software Architectures and Component Technology: The State of the Art in Research and Practice, M. Aksit (Ed.), Kluwer Academic Publishers, 2001.

[17] B.Tekinerdoğan & M. Akşit. *Deriving design aspects from conceptual models*. In: Demeyer, S., & Bosch, J. (eds.), Object-Oriented Technology, ECOOP '98 Workshop Reader, LNCS 1543, Springer-Verlag, pp.410-414, 1999.

[18] B. Tekinerdogan & M. Aksit, Providing Automatic Support for Heuristic Rules of Methods, in Object-Oriented Technology, S. Demeyer and J. Bosch (Eds.), LNCS 1543, ECOOP'98 Workshop Reader, Springer Verlag, pp. 493-499, July 1998.