## Using UML in Architecture-Level Modifiability Analysis

Lassing, N.; Rijsenbrij, D.; van Vliet, H.

***published in***
ICSE 2001 Workshop on Describing Software Architecture with UML
2001

***document version***
Publisher's PDF, also known as Version of record

# Using UML in Architecture-Level Modifiability Analysis

**Nico Lassing, Daan Rijsenbrij and Hans van Vliet**
Vrije Universiteit, Faculty of Sciences
De Boelelaan 1081A
1081 HV Amsterdam
The Netherlands
+31 20 4447768
{nlassing, daan, hans}@cs.vu.nl

**ABSTRACT**
In our scenario-based method for software architecture-level modifiability analysis of business information systems, we use architectural views to determine and express the effect of change scenarios. We distinguish four architectural views. We used the Unified Modeling Language (UML) notation to formalize the information captured in these views. This paper reports on the experience we gained in this process.

**Keywords**
Software architecture, software architecture analysis, architectural views, view model, modifiability

## 1 INTRODUCTION

We have defined a scenario-based method for architecture-level modifiability analysis of business information systems, ALMA [2], based on the software architecture analysis method (SAAM) [6]. ALMA consists of five steps: goal setting, architecture description, change scenario elicitation, change scenario evaluation and interpretation. In the architecture description step, we make use of architectural views that are used in subsequent steps to determine and express the effect of change scenarios. An architectural view is a representation of a system from a certain perspective. Views are constructed according to viewpoints that establish the techniques used for its creation [5]. We distinguish four viewpoints that are useful in modifiability analysis.

We found that in the description step of ALMA it is important to make a distinction between the internals of a system, i.e. its micro architecture, and the role of a system in its environment, i.e. its macro architecture [8]. At both levels we identify two viewpoints, the conceptual viewpoint and development viewpoint at the micro architecture level and the context viewpoint and the technical infrastructure viewpoint at the macro architecture level. The micro architecture level viewpoints rougly coincide with viewpoints identified earlier by Kruchten in his 4+1 View Model [7] and by Soni et al. [12]. The viewpoints at the macro architecture level are new.

Initially, we used informal notation techniques to express the information in views. However, we noticed that this leads to lack of clarity about the semantics of the views. We next used the Unified Modeling Language (UML) notation [3] to formalize the information captured in these views [9]. This paper reports on the experiences that we gained in this process. Section 2 introduces the viewpoints, section 3 discusses the experiences and section 4 contains a summary and some concluding remarks.

## 2 VIEWPOINTS FOR MODIFIABILITY

In our method for architecture-level modifiability analysis architectural description plays an important role. The description of a system's software architecture allows us to determine the effect of change scenarios and, in addition, guides the change scenario elicitation process [10]. These activities require a number of architectural views. Based on our experiences with architecture-level analysis, we have identified four viewpoints that capture decisions concerning modifiability [9]. Two of these viewpoints concern the system in its environment, the macro architecture level, and two concern the internals of the system, the micro architecture level.

The viewpoints identified at the macro architecture level are the context viewpoint and the technical infrastructure viewpoint. The **context viewpoint** gives an overview of systems in the environment with which the system communicates. The **technical infrastructure viewpoint** addresses the relationships of the system to its technical environment. At the micro architecture level, we identify the conceptual viewpoint and the development viewpoint. The **conceptual viewpoint** gives an overview of the high-level design elements of the system, representing concepts from the system's domain. These elements may be organized according to a specific architectural style. The **development viewpoint** concerns decisions related to the structure of the implementation of the system. These decisions are captured in prescriptions for the building blocks that will be used in the implementation of the system.

The micro architecture level viewpoints coincide with the logical and the development viewpoint in Kruchten's 4+1

View Model [7] and the conceptual and the code architecture identified by Soni et al. [12]. The macro architecture viewpoints were not identified before and are discussed in this section. A more elaborate treatment can be found in [9].

**Context View**
The context viewpoint gives an overview of the system and the systems in its environment with which it communicates. This communication can take the form of file transfer, a shared database or 'call/return' (see [4]). In the analysis this view is used to assess which systems have to be adapted to implement a change scenario. This view also includes an identification of the owners of the various systems, which is useful for determining who is involved in the changes brought about by a change scenario. Figure 1 gives an overview of the concepts used in a context view and their notation technique.

SYSTEM. A system is a collection of components organized to accomplish a specific function or set of functions. A system is depicted using the standard UML-notation for a component with the stereotype «system».

SHARED DATABASE. A shared database is a database that is used by several systems. The type of dependency this exposes is that when adaptations to one of the systems using this database requires (structural) adaptations to this database, other systems may have to be adapted as well. The notation for a shared database is the symbol for a data store with the stereotype «shared». The fact that a system uses the database is indicated through a dashed arrow (UML-notation for dependencies).

OWNER. An owner is an organizational unit financially responsible for the development and management of a system, or simply put the entity that has to pay for adaptations to the system. Ownership is an important notion with respect to modifiability, because the involvement of different owners in adaptations complicates the required changes (for a more elaborate treatment of this topic see [8]). The owner of a system or shared database is indicated as an attribute of the object.

FILE TRANSFER. File transfer denotes that one system (asynchronously) tranfers information to another system using a file. The dependency created by this type of communication mostly concerns the structure of the files transferred: if the structure of the information exchanged between the systems changes, the file structure has to be adapted, requiring the systems to be adapted as well. Another type of dependency is the technology/protocol used for transferring the file. File transfer between two systems is depicted using a directed link with stereotype «file transfer».

CALL/RETURN. A call/return relationship between systems denotes that one system calls one or more of the procedures of another system. This entails direct communication between systems. This type of relationship brings about a number of dependencies. They include the technology used, the structure of the parameters and, additionally, the systems have to be able to 'find' and 'reach' each other. A call/return relationship between systems is depicted using a directed link with stereotype «call/return».

**Technical Infrastructure View**
The technical infrastructure viewpoint contains an overview of the dependencies of the system on elements of the technical infrastructure (operating system, database management system, etc.). The technical infrastructure is often shared by a number of systems within an organization. Common use of infrastructural elements brings about dependencies between these systems: when a system owner decides to make changes to elements of the technical infrastructure, this may affect other systems as well.

Additional dependencies are created when an organization decides to define a standard for the technical infrastructure. Such a standard prescribes the products to be used for infrastructural elements like the operating system, middleware, etc. A standard is often defined to make sure that systems function correctly in an environment in which the infrastructure is shared, but, at the same time, it limits the freedom of the individual owners to choose the products to be used for their systems. These influences are also captured in this viewpoint.

Figure 2 gives an overview of the concepts used in a technical infrastructure view and their notation technique.

DEPLOYMENT ELEMENT. A deployment element is a unit of a software system that can be deployed autonomously. A deployment element is represented using the UML-notation for a component with stereotype «deployment».

STANDARD. Standards prescribe the use of certain deployment elements. A standard is represented by the UML-notation for a package with stereotype «standard». Elements that are prescribed by this standard are indicated with a dashed arrow.

DEPENDENCY. A dependency exists between two elements if changes to the definition of one element may cause changes to the other [3]. A dependency between two deployment elements is indicated using the standard UML notation for dependency, i.e. a dashed arrow.

NODE. A node is a computer on which a number of deployment elements are physically located. A node is represented using the standard UML notation for node, i.e. a shaded rectangle.

**Example: Dutch Tax Department**
One of the case studies that we conducted concerns a system at the Dutch Tax Department. We performed architecture analysis for the system that will be used by Dutch Customs to process supplementary declarations. Figure 3 shows the context view of this system. The figure shows that supple-

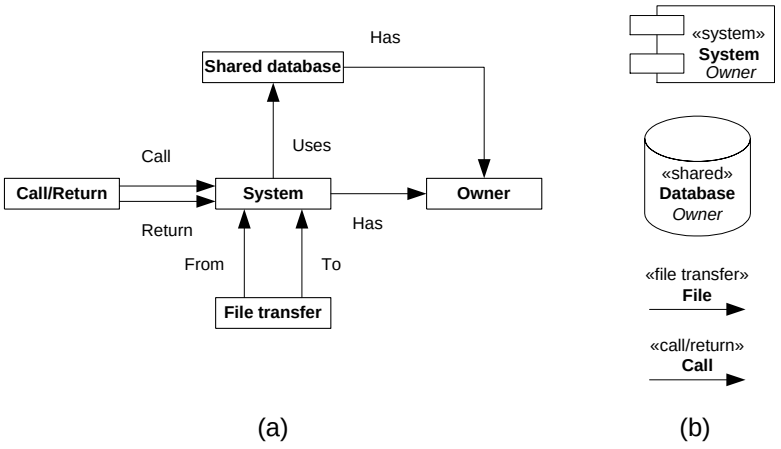


Figure 1: (a) Concepts of the context viewpoint and (b) their notation technique

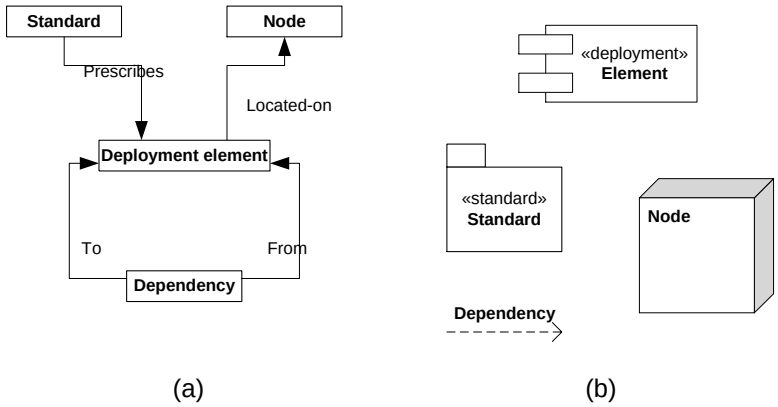


Figure 2: (a) Concepts of the TI viewpoint and (b) their notation technique

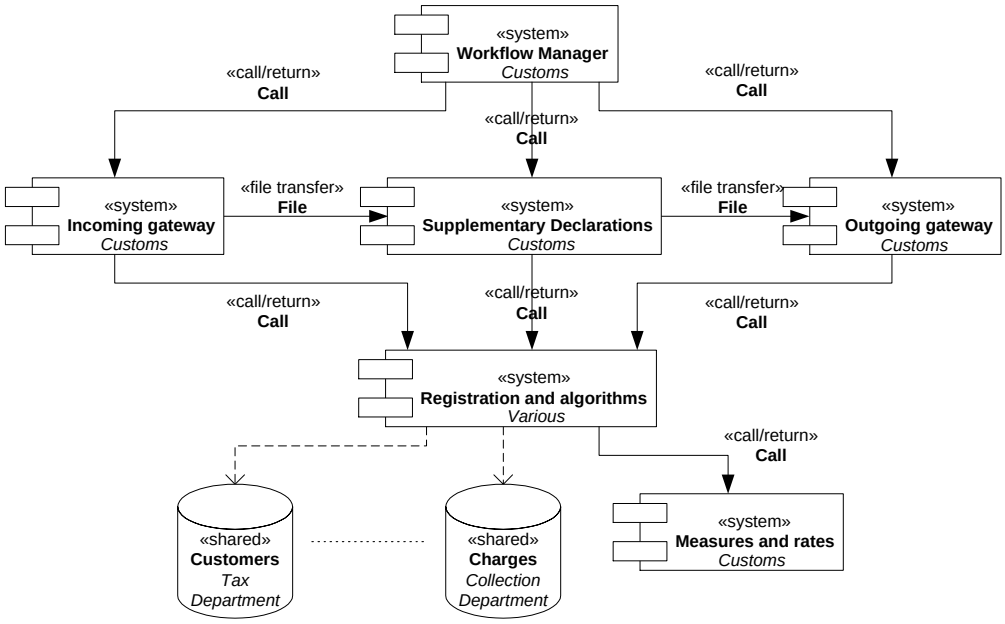


Figure 3: The context view of the system for processing supplementary declarations

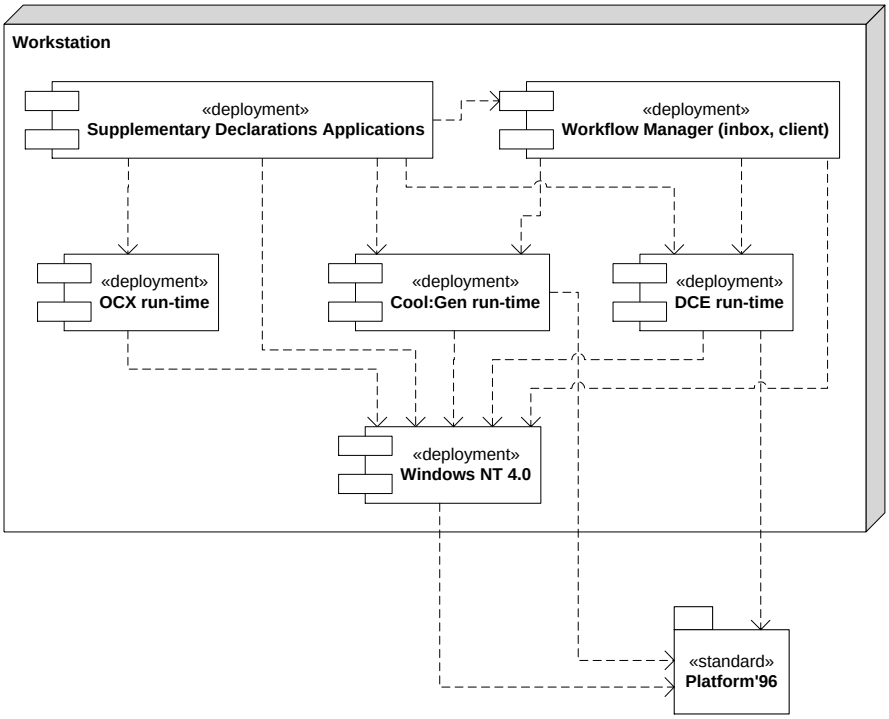


Figure 4: The technical infrastructure view of the system for processing supplementary declarations

mentary declarations that are submitted by clients are collected at the incoming gateway system. This systems converts the declarations to a generic format and transfers them to the system 'supplementary declarations'. The declarations are then processed and messages are sent to clients through the outgoing gateway system. This process is controlled by a workflow manager and the systems make use of a number of common data sources and algorithms.

Figure 4 shows part of the technical infrastructure view of the system. This figure only shows the workstations of the system, the complete view also addresses application servers and database servers. Although this view concerns the same system, it focuses on a different aspect resulting in a completely different figure. It shows that there are several infrastructural elements with a number of dependencies between them. The OCX run-time files, for instance, are dependent on the operating system used. If the operating system changes, the OCX run-time files may need to be replaced too. This influences the system's modifiability.

## 3 EXPERIENCES WITH UML

When formalizing our viewpoints using UML, we gained a number of experiences. These are discussed in this section.

**Defined Semantics**

The main reason why we chose to use UML to describe the viewpoints was that we experienced that the informal notation techniques we used before resulted in lack of clarity about the concepts used in these viewpoints. The lines and boxes used in this informal notation proved to be open for misinterpretation. Using UML meta models to describe the concepts of the viewpoints forced us to consider and define their semantics explicitly.

Formalizing the context viewpoint, for example, revealed that not only systems have owners, but that databases have owners as well. We had not considered this before.

**Detail versus Precision**

A system's software architecture is an abstraction of the system. At the software architecture level a high-level description of the software architecture is created. This means that not all aspects of the system are specified down to the smallest detail. Using a formalized notation technique such as UML for architectural description may suggest that these details are included. We should be careful not to confuse *precision* with *detail*. Using UML leads to architectural models with *precisely* defined semantics; we can be precise about concepts without going into their details.

In the conceptual architectural view of a system, for instance, the high-level design elements of the system and their connectors are captured [9]. At that level, we do not concern ourselves with the details of the communication that takes place between the components distinguished; those are addressed at lower levels.

**Symbol Overload**

In general, software architecture has to do with components. A widely used definition of software architecture is 'the structure or structures of the system, which comprise soft-

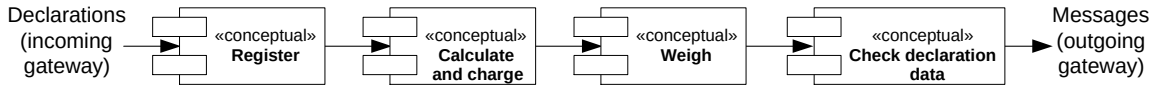


Figure 5: Conceptual view of the system for processing supplementary declarations

ware components, the externally visible properties of those components, and the relationships among them' [1]. Most architectural 'structures' or 'views' include the notion of *component*, although of a different meaning. All of the viewpoints that we identified for modifiability analysis include some kind of 'component'. At the macro architecture level, for instance, the context view includes systems as components and the technical infrastructure view includes deployment elements as components. UML includes a single symbol to represent components. This symbol is used in all views to represent components in their different meanings. Different pictorial elements for different types of 'component' would increase the legibility of the various views.

**Architectural Styles**
Architectural styles are an important tool to communicate the rationale that is used for a system. Architectural styles include among others a pipe-and-filter architecture and a blackboard architecture [11]. One of the downsides of UML is that it does not provide any facilities for showing that a system follows a specific architectural style.

For instance, for the system 'supplementary declarations' at the Dutch Tax Department a pipe-and-filter architecture is used for processing declarations. In UML notation this leads to the model shown in Figure 5. This model shows that there are four components that communicate in some way with each other. However, it is not apparent from this figure that these components are organized using a pipe-and-filter style. This has to be pointed out in the textual description that accompanies this view.

**Suitability for Stakeholders**
One of the reasons for using software architecture is that it is a vehicle for stakeholder communication [1]. Stakeholders include both technical people such as designers and developers, and non-technical people such as clients and possibly future users. The nature of UML models is mostly technical. A relevant question is whether such technically oriented architectural models are suitable for all stakeholders. Perhaps, another notation technique is more suitable for communicating the architecture to the non-technical stakeholders.

## 4 CONCLUSIONS

In the method for architecture-level modifiability analysis that we advocate, four viewpoints are used to capture the information required. We used the UML to formalize the notation technique for these viewpoints. This paper discusses the experiences we gained in this process.

We experienced that formalizing the viewpoints using UML is useful, because it forces us to consider the semantics of the concepts used. The downside of UML is that it does not provide any facilities for representing architectural styles. With respect to the notation techniques provided by UML, we experienced that the nature of UML diagrams may suggest detail that is not yet present at the architecture level. In addition, we found that it may be confusing that some symbols are used in a number of viewpoints representing different concepts. And finally, we found that UML diagrams may not be appropriate for all stakeholders in a system's software architecture.

**ACKNOWLEDGEMENTS**
This research is mainly financed by Cap Gemini Netherlands.

## REFERENCES

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. 1998. Addison Wesley, Reading, MA.

[2] P. O. Bengtsson, N. Lassing, J. Bosch and H. van Vliet. *Analyzing software architectures for modifiability*. Technical Report HK-R-RES–00/11-SE, Högskolan Karlskrona/Ronneby. 2000.

[3] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 2nd edn. Addison Wesley, Reading, MA. 1999.

[4] V. Gruhn and U. Wellen. Integration of heterogenous software architectures - an experience report. In: P. Donohoe, ed. *Software architecture: Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, pages 437–454, Kluwer Acadamic Publishers, Dordrecht, The Netherlands. 1999.

[5] IEEE. *IEEE Recommended Practice for Architecture Description*. IEEE Std 1471, 2000.

[6] R. Kazman, G. Abowd and L. Bass. Scenario-Based analysis of software architecture. *IEEE Software* 13(6): 47–56, 1996.

[7] P. Kruchten. The 4+1 view model of architecture. *IEEE Software* 12(6): 42–50, 1995.

[8] N. Lassing, D. Rijsenbrij and H. van Vliet. Towards a broader view on software architecture analysis of flexibility. In *Proceedings of the 6th Asia-Pacific Software*

*Engineering Conference '99 (APSEC'99)*, pages 238–245. 1999.

[9] N. Lassing, D. Rijsenbrij and H. van Vliet. *Viewpoints on modifiability*. Technical report, Vrije Universiteit, Amsterdam. 2000.

[10] N. Lassing, D. Rijsenbrij and H. van Vliet. *Scenario Elicitation in Software Architecture Analysis*. Technical report, Vrije Universiteit, Amsterdam. 2000.

[11] M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of the 21st International Computer Software and Application Conference (CompSac)*, Washington, D.C., 1997.

[12] D. Soni, R. L. Nord and C. Hofmeister. Software architecture in industrial applications. In R. Jeffrey and D. Notkin, eds, *Proceedings of the 17th International Conference on Software Engineering*, pages 196–207, ACM Press, New York, 1995.