Viewpoints on Modifiability

Nico Lassing, Daan Rijsenbrij and Hans van Vliet
Division of Mathematics and Computer Science
Faculty of Sciences, Vrije Universiteit
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
{nlassing, daan, hans}@cs.vu.nl

Abstract

Software architecture is generally regarded as an important tool to achieve systems of higher quality. It is claimed that the foundation for a system's quality is laid by the decisions made in the software architecture. A question that is occupying both researchers and practitioners is *in which areas should decisions be made in the software architecture?* We believe architectural view models play an important role in the answer to this question. View models consist of a coherent set of architectural views. These view models have both a prescriptive and a descriptive role in the development process. Their prescriptive role is that they call for a number of aspects to be considered when defining a software architecture and their descriptive role is that they provide a framework to document a software architecture. Currently, a number of view models exist, the most important of which are the 4+1 View Model of Kruchten and the four views by Soni et al.

In our experience with modifiability analysis for business information systems we found that the views in current view models do not include all information required. In this paper we discuss the views we found useful for architecture level impact analysis of business information systems. They are illustrated using a case study we performed for the Dutch Tax Department. We claim that when these views are required for architecture level impact analysis, the decisions they capture should also be considered during architecture development.

Keywords: software architecture, software architecture analysis, architectural views, view model, modifiability

1 Introduction

Many organizations today operate in a world in which change is ubiquitous. Business developments and new technology force them to adapt their information systems regularly. Studies have shown that a large part of the costs associated with an information system is spent on such adaptations [16]. Software architecture is seen as an important tool to reduce these costs.

A system's software architecture is the first design artifact in the development process, capturing the very first design decisions for a system [1]. Examples of these decisions include the distribution of functionality over components and the dynamic behaviour of components. Such decisions have a large influence on the quality of the resulting system. The distribution of functionality over components, for instance, influences the number of components that has to be modified when the system is adapted (related to modifiability), but it also influences the required communication between the components (related to performance). So, different aspects should be considered in the software architecture and the decisions made on these aspects influence different quality attributes. This makes the software architect's task quite complex.

Fortunately, the architect is not entirely in the dark. View models have been introduced to help the architect focus on the important aspects of the software architecture. The most important of these view models are the 4+1 View Model by Kruchten [10] and the four architectures by Soni et al. ([18] and [6]). A view model constitutes a coherent set of architectural perspectives, or viewpoints, that should be considered when defining a system's software architecture. Each of these viewpoints emphasizes different aspects of the software architecture, such as the run-time perspective (the dynamic behaviour of components) or the deployment perspective (the mapping of software components to hardware). The architect takes decisions on each of these aspects and documents these using a number of models, one for each viewpoint. The result is an architectural description, which captures the architect's decisions and thereby allows for analysis of the results. This type of analysis aims to answer the question whether the software architecture supports the quality requirements that were defined for the system. This is the area of software architecture analysis.

We have defined ALMA, a method for Architecture-Level Modifiability Analysis [2]. ALMA uses change scenarios to analyze a system's modifiability. The principle of such a scenario-based approach is that the analyst interviews various stakeholders and asks them to come up with events that may occur in the future and require the system to be adapted. These events are recorded in change scenarios and the analyst then investigates the software architecture to assess the impact of these scenarios. Architectural views are used in this process to determine for each scenario the components that have to be adapted. An important question is which views are required for such an architecture-level impact analysis.

Based on our experiences with applying ALMA to business information systems (see for instance [12] and [13]), we have identified four viewpoints that provide full insight into the effect of realizing change scenarios. Two of these viewpoints roughly coincide with viewpoints earlier identified by Kruchten [10] and Soni et al. [18]. These viewpoints concern the internals of the system whose modifiability is assessed. The two other viewpoints address the relationships between the system being analyzed and its environment. These latter viewpoints are not explicitly included in existing models; yet, we found them to be essential in the analysis of business information systems. Apparently, the aspects reflected in those viewpoints are relevant for the modifiability of systems in this domain and should therefore also be considered by the architect when creating the software architecture.

In this paper we discuss the four viewpoints using Sagitta 2000, one of the systems for which we performed an architectural assessment. The remainder of the paper is divided into six sections. In section 2 we discuss the role view models play in the development of business information systems. In section 3 we give an overview of ALMA, our method for software architecture analysis of modifiability. In section 4 we introduce the four viewpoints required when using ALMA to assess business information systems. In section 5 we give a brief introduction to Sagitta 2000 and describe its software architecture using the four views. In section 6 we list the change scenarios that we identified in our analysis and show how architectural views are used for assessing their effect. Section 7 contains our conclusions.

2 Software architecture, views, viewpoints and view models

A number of definitions of software architecture exist. One of the most widely used definitions is given by the Software Engineering Institute [1]:

> The software architecture of a program or computer system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

A similar, but subtly different definition is formulated in IEEE Standard 1471 [7]:

> Architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.

The main difference between these definitions is that the latter not only focuses on the components of the system and their relationships, but also takes the environment of the system into account. In the area we are working in, business information systems, this is very important, because such systems are seldomly isolated. So, throughout this paper we will use the IEEE 1471 definition.

The 'system's fundamental organization' is the result of a series of design decisions. These decisions put constraints on the building process as well as the product delivered. They address both structural issues and the dynamics of the system. The structural aspects include the decomposition of the system in components and the distribution of components over various nodes in the network. Examples of decisions related to the system's dynamics are the life-cycle of components and the communication between components. In an architecture-based development process, the result of these decisions, i.e. the software architecture, is represented in an architectural description.

An explicit representation of the software architecture is important for various reasons. First, it makes the architect's considerations concrete, forcing him to think about the important aspects of the system. Secondly, an explicit description of the software architecture enables communication between stakeholders (such as the architect, the development team, the customer and the maintenance organization). Finally, such a description allows for early analysis of the system [1], which is called software architecture analysis. In this paper we focus on the first and third reason: which aspects of the systems should be considered in order to *build* modifiable systems and how should they be recorded in order to be able to *analyze* the modifiability of the software architecture.

Several proposals have been made for describing a software architecture, the most important of which are the view models as introduced by Kruchten [10] and Soni et al. [18], architecture description languages (ADLs) and the IEEE Standard 1471 on architectural description [7]. View models are based on the principle that a software architecture cannot be captured in a single view. View models define a number of aspects of a software architecture that should be addressed in a software architecture description and suggest a number of techniques to describe these aspects. Kruchten's 4+1 View Model, e.g. contains the following views: (1) the *logical view*: the key abstractions of the system, (2) the *process view*: the set of the independently executing processes, (3) the *physical view*: a mapping of the software onto the hardware, (4) the *development view*: the static organization in the development environment, and (+1) the *scenarios*: these scenarios show how the elements of the four views work together. Soni et al. define the following four views: (1) the *conceptual architecture view*: the description of the architecture in domain elements, (2) the *module architecture view*: the decomposition of the software and its organization into layers, (3) the *execution architecture view*: the mapping of modules to run-time images, defining the communication between them and assigning them to physical resources and (4) the *code architecture view*: the mapping of modules and interfaces to source files and of run-time images to executable files. The semantics of the views in both models are defined rather informally.

With respect to formality, ADLs are on the other end of the spectrum (for an overview of ADLs, see [15] and [3]). They provide formal techniques which are aimed at creating a representation of the architecture that can be analyzed using automated tools and may serve as a basis to generate an implementation of a system. This requires that a lot of lower-level information is also recorded, which makes them less suitable for architecture description in the area of business information systems, where a system's software architecture is often not more than a rough sketch of the system.

IEEE Standard 1471 defines a framework for architectural description which is also based on the notion of views. However, while Kruchten and Soni et al. use the notion 'view' to address both the semantics of a view and its instantiation for a specific system, IEEE 1471 makes a clear separation between the two. The standard introduces the term viewpoint to refer to the semantic aspect and the term view to refer to the instantiation for a specific system. A viewpoint is then defined as *a specification of the conventions for constructing and using a view* and a view as *a representation of a whole system from the perspective of a related set of concerns*. A viewpoint acts as a pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis. This distinction allows for more accurate reasoning about these concepts and we will use it throughout the remainder of this paper.

IEEE 1471 does not prescribe any specific views that should be included in an architectural description, it merely provides a framework for description. The viewpoints included in an architectural description indicate which aspects the architect has considered when defining the software architecture. Each viewpoint entails a number of decisions that influence certain qualities of the system. An important issue then is *which* viewpoints to include in an architectural description.

We have experienced that for software architecture analysis of modifiability in the area of business information systems, the viewpoints included in existing view models do not provide sufficient information. This led us to believe that there is also a need for additional viewpoints to be used when *developing* systems in this area. These viewpoints are introduced in section 4, but first we give an overview of our method for modifiability analysis.

3 Architecture-Level Modifiability Analysis (ALMA)

The generic method for modifiability assessment of software architecture that we advocate is based on the Software Architecture Analysis Method (SAAM) [8]. Both methods are scenario-based, which means that they use concrete changes to assess the modifiability of a system based on its software architecture. The main difference between ALMA and SAAM is that SAAM does not explicitly address the need for differences in the techniques used as part of the analysis depending on the goal of the analysis. In ALMA this is an essential part and the need to clearly distinguish the goal of the analysis and to only have one goal for each analysis is emphasized [11].

Another scenario-based architecture analysis method is the Architecture Trade-off Analysis Method (ATAM) [9], the successor of SAAM. ATAM is a framework for architecture analysis of a number of (conflicting) quality attributes and as such does not prescribe any techniques for evaluating the individual quality attributes. ALMA could be used in combination with ATAM to analyze modifiability. We define modifiability as the ease with which a system can be adapted to changes in the functional specification, in the environment, or in the requirements. We explicitly exclude the correction of implementation errors and changes in the quality requirements of the system. Changes in the quality requirements are left aside, because the assessment of their effect requires other techniques (and, mutatis mutandis, other viewpoints). Changes in the required processing capacity of a system, for example, relate to scalability, which is a field of study in itself (see [19]).

At the architecture level, modifiability has to do with allocation of functionality to components and dependencies between these components. The allocation of functionality determines the components that have to be adapted for a change scenario and dependencies determine ripple effects of these adaptations. The aim of ALMA is to assess whether the decisions taken with regard to these aspects result in the required modifiability.

ALMA consists of five steps. These steps are not always performed in the indicated sequence, it

is often necessary to iterate over the various steps. The steps are:

1. Set goal: determine the aim of the analysis
2. Describe software architecture: give a description of the relevant parts of the software architecture
3. Elicit change scenarios: find the set of relevant change scenarios
4. Evaluate change scenarios: determine the effect of the set of change scenarios
5. Interpret results: draw conclusions from the analysis results

In this section, we give a brief overview of the steps in ALMA. A more elaborate discussion of the full method is given in [2].

3.1 Goal setting

The first step to take in software architecture analysis of modifiability is to set the analysis goal. The goal determines the type of results that will be delivered by the analysis. In addition, the goal influences the choice of techniques to be used in subsequent steps. Different goals ask for different techniques. With respect to modifiability, the following goals can be pursued: (1) risk assessment, (2) maintenance prediction and (3) software architecture comparison. The aim of risk assessment is to find types of changes for which the system is inflexible. We are then interested in scenarios which are particularly difficult to accomplish. When performing maintenance cost prediction the aim is to estimate the cost of (adaptive) maintenance effort for the system in a given period. Finally, when the goal is software architecture comparison, we are comparing two or more candidate software architectures to find the most appropriate one. The difference with the two aforementioned goals is that in comparison we make relative statements about a number of candidate software architectures, while with the other goals we make absolute statements about a single candidate architecture.

3.2 Software architecture description

After the goal of the analysis is set, the next step is to create a representation of the software architecture. The information acquired in this step is recorded in a number of architectural views. These views should enable us to do architecture level impact analysis for the set of change scenarios, i.e. assess and express their effect on the software architecture. To this end, the description should provide an overview of the dependencies between the system and its environment and the mutual dependencies between the components of the system. Section 4 discusses the viewpoints that we use to capture these dependencies for business information systems.

3.3 Scenario elicitation

An important step in a scenario-based approach to modifiability analysis is to collect a set of events, change scenarios, that will require the system to be adapted. This set of change scenarios captures the events that stakeholders expect to occur in the future of the system. The main technique to elicit this set is to interview stakeholders, because they are in the best position to predict what may happen in the operational life of the system. The aim of the scenario elicitation step is to come to a set of change scenarios that supports the goal that we have set for the analysis [14].

3.4 Scenario evaluation

After eliciting a set of change scenarios, we determine their effect on the software architecture. This means that we perform an architecture level impact analysis for each of the scenarios individually. To do so, we first have to determine the components of the system and components of other systems that have to be adapted to implement the change scenario. This task is typically performed in collaboration with members of the development team. After we have determined the components that should be adapted, the results are expressed in a way that supports the goal of the analysis.

3.5 Interpretation

After we have determined the effect of the change scenarios, we can interpret these results to come to a conclusion about the system under analysis. The way the results are interpreted is again dependent on the goal of the analysis. However, for each goal it is important that the likelihood of the scenarios is considered in this interpretation process.

If the goal of the analysis is risk assessment the results of the scenario evaluation are investigated to determine which change scenarios pose risks, i.e. for which scenarios the product of probability and costs is too high. The criteria for determining which values are still acceptable should be based on managerial decisions by the owner of the system. When risks are found, various risk mitigation strategies are possible: avoidance (take measures to avoid that the scenario will occur or take action to limit their effect, for instance, by use of tools), transfer (choose another software architecture) and acceptance (accept the risks).

If the goal of the analysis is to do maintenance prediction, the aim of this step is to come to an estimate of the amount of effort that is required for maintenance activities in the coming period. Again, it is up to the system's owner to decide which values are still acceptable. If the goal of the analysis is architecture comparison, we compare the results of the evaluation of the two sets of scenarios and choose the most appropriate candidate architecture.

4 2+2 Viewpoints on Business Information Systems

The previous sections explained the importance of architectural description in the architecture design and analysis process. An architectural description consists of a number of views, i.e. models of the system's software architecture from certain viewpoints. Each of these views captures a number of decisions. This section discusses four viewpoints on business information systems that capture decisions related to modifiability.

The rationale for these viewpoints is found in the systems we assessed. We have no formal proof that the the four viewpoints defined below are sufficient to assess the impact of change scenarios. Our experience with analyzing the architecture of four different systems, however, strongly suggests that these viewpoints capture the relevant modifiability-related decisions.

At the software architecture level modifiability has to do with separation of functionality and dependencies, i.e. *how do we distribute the functionality over components?* and *how are these components related?* Allocation of functionality determines which components have to be adapted to realize certain changes and dependencies determine how changes to a component affect other components. In an architectural description in which modifiability is addressed these decisions should be made explicit.

The aforementioned questions focus on the system's internals. For business information systems it is not sufficient to study only the internals of the system. Such systems are rarely isolated, they are

often part of a larger suite of systems. At the systems level questions similar to the ones at the component level recur: *how do we distribute functionality over systems?* and *what are the dependencies between these systems?* These decisions also affect modifiability. Therefore, we split the description of a system's software architecture into two parts: (1) the *macro architecture level*: the software architecture at the systems level, and (2) the *micro architecture level*: the software architecture of the internals of the system.

Each of the viewpoints captures the notions of components and dependencies between components. Changes result in adaptations to one or more of these components and, because of ripple effects that result from dependencies identified, other components may be affected as well. Dependencies identified at the architectural level are often specified at a rather course level. For instance, in the example discussed in section 5 we identify a dependency between the type of middleware used and the underlying operating system. It requires deep technical expertise to make these dependencies more precise. For this reason we assume the presence of experts during the scenario evaluation step.

This section is split up into two subsections. In section 4.1 we introduce the viewpoints at the macro architecture level. To do so, we define the concepts that are used in these viewpoints, their semantics and their relationships. We do not aim to give a logically sound definition of the viewpoints, because architecting is not a formal activity, but it should be clear to the reader what the concepts mean. Section 4.2 does the same for the viewpoints at the micro architecture level.

4.1 Macro Architecture Level

Neither Kruchten [10] nor Soni et al. [18] identify viewpoints at the macro architecture level; they focus on the internals of a system. We found the viewpoints at the macro architecture level to be essential for business information systems, because these systems are rarely isolated and, therefore, it is essential to include the system's environment in the description of the software architecture. From a modifiability perspective, the environment is not only a source of changes, but it can also be a complicating factor for implementing the changes associated with a change scenario (for a more elaborate discussion see [13]).

We identified two viewpoints at the macro architecture level that capture decisions related to modifiability. This section gives an overview of these viewpoints, the concepts used and their notation technique. The notation used for the viewpoints is based on the Unified Modeling Language (UML) [4]. These viewpoints are:

- The **context viewpoint**: 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 [5]). 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».

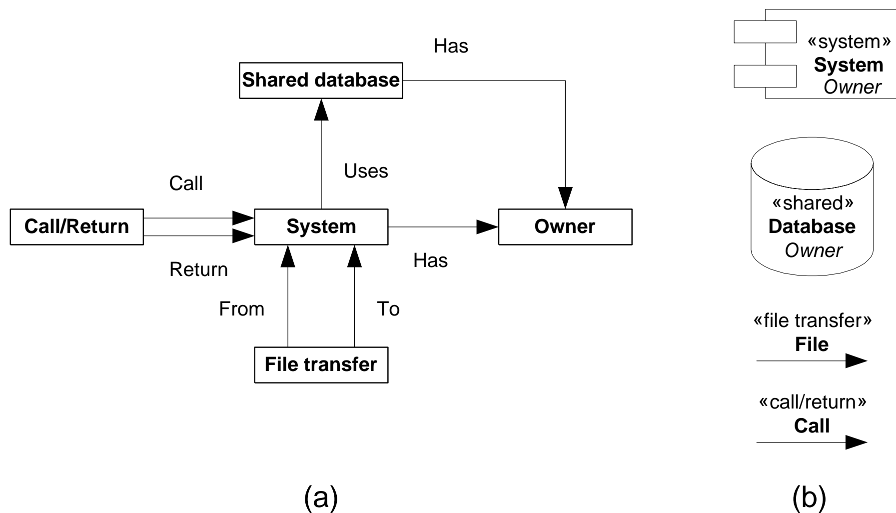

Figure 1: (a) Concepts of the context viewpoint and (b) their notation technique

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 [13]). 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».

- The **technical infrastructure viewpoint**: 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.

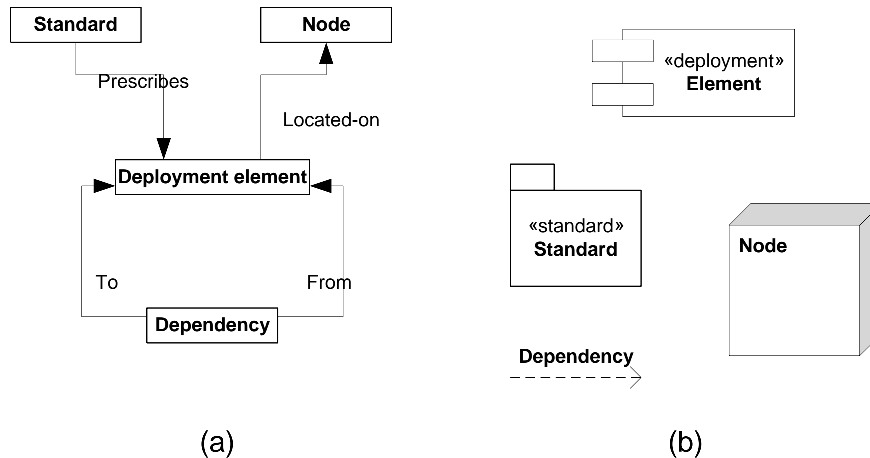

Figure 2: (a) Concepts of the TI viewpoint and (b) their notation technique

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 [4]. 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.

The context viewpoint does have some similarities with the conceptual architecture viewpoint defined by Soni et al. [18], when we consider systems as components. Both give an overview of major design elements. However, the design elements included in the context viewpoint are systems whose boundaries can be clearly delineated (both conceptually and in the implementation), in contrast to the elements of the conceptual architecture viewpoint that need not recur as such in the actual implementation.

The technical infrastructure viewpoint does have some similarities with the physical viewpoint defined by Kruchten [10]. They both concern the distribution of system elements over machines. However, the physical viewpoint does not show the relationships between a system and the other

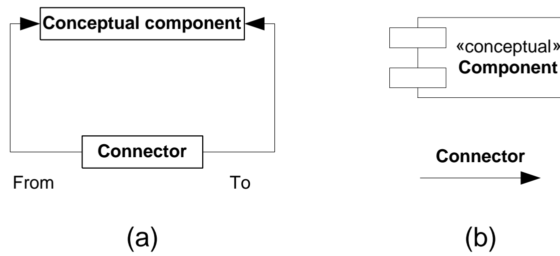

Figure 3: (a) Concepts of the conceptual viewpoint and (b) their notation technique

infrastructural elements that are present on these machines. For modifiability, it is essential to have an overview of these dependencies.

4.2 Micro Architecture Level

The viewpoints at the micro architecture level focus on the internals of the system. At this level modifiability concerns the effort that is required to make changes to the system's internals. We found two viewpoints that capture decisions influencing modifiability at the micro architecture level. These viewpoints roughly coincide with viewpoints that are also recognized by Kruchten and Soni et al.

In this section we discuss the two micro architecture level viewpoints: the type of decisions they capture and their relationship to modifiability and the concepts they use and their notation technique. In addition, we discuss their relation with Kruchten's 4+1 View Model and the four views by Soni et al.

The micro architecture level viewpoints are:

- The **conceptual viewpoint**: 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. Examples of architectural styles include the pipe-and-filter architecture, in which a number of successive transformations is performed on an item, and the blackboard architecture, in which a number of autonomous components manipulate shared data. A classification of architectural styles can be found in [17].

  For modifiability this viewpoint allows us to judge whether the high-level decomposition of the system supports future changes. Figure 3 gives an overview of the concepts used in a conceptual view and their notation technique.

  CONCEPTUAL COMPONENT. A conceptual component is a high-level design element of the system. A conceptual component is represented using the UML-notation for a component with stereotype «conceptual»

  CONNECTOR. A connector indicates a relationship between two conceptual components. A connector is represented using a directed arrow.

- The **development viewpoint**: an overview of 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. For instance, the decision to separate business logic and data access may result in the prescription that some components may contain business

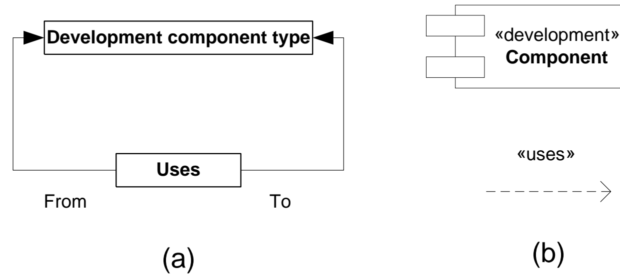

Figure 4: (a) Concepts of the development viewpoint and (b) their notation technique

logic and some components access data, but that such functionality is never combined in one component.

The prescriptions may be enforced by the development environment (CASE-tool, programming environment, programming language, etc.), for instance by the type of implementation units that it supports. Examples of such units are modules, files and classes, but also screens, windows, and dialog boxes.

We use this viewpoint in the analysis of modifiability to determine whether adaptations to a system are limited to a certain type of component. For instance, if business logic and data access are implemented in separate components, then the impact of changes to the underlying data model can be confined to the data access components. Figure 4 gives an overview of the concepts used in a development view and their notation technique.

DEVELOPMENT COMPONENT TYPE. A development component type is a type of unit that may be used in the development environment. A development component type is represented using the UML-notation for a component with stereotype «development».

USES. When it is permitted that one development component type makes use of facilities provided by another development component type, a uses relationship exists between the two. A uses relationship is indicated using the UML-representation of a dependency (dotted arrow) with stereotype «uses».

The conceptual viewpoint introduced above equals the conceptual viewpoint introduced by Soni et al. [18] and the logical viewpoint included in Kruchten's 4+1 View Model [10]. They all concern the high-level structure of the system.

The development viewpoint is also contained in the view models of both Kruchten's and Soni et al. Kruchten recognizes a viewpoint with the same name and Soni et al. recognize it as the code architecture. The main difference between their approaches and ours is that we limit ourselves to the *types* of components used in the development environment. Capturing the instances of these components as well is not feasible in our domain, because this would result in a very large and complex model.

5 Sagitta 2000

To illustrate the viewpoints discussed in the previous section, we show the four architectural views for Sagitta 2000. Sagitta 2000 is a large information system that is being developed by the IT department

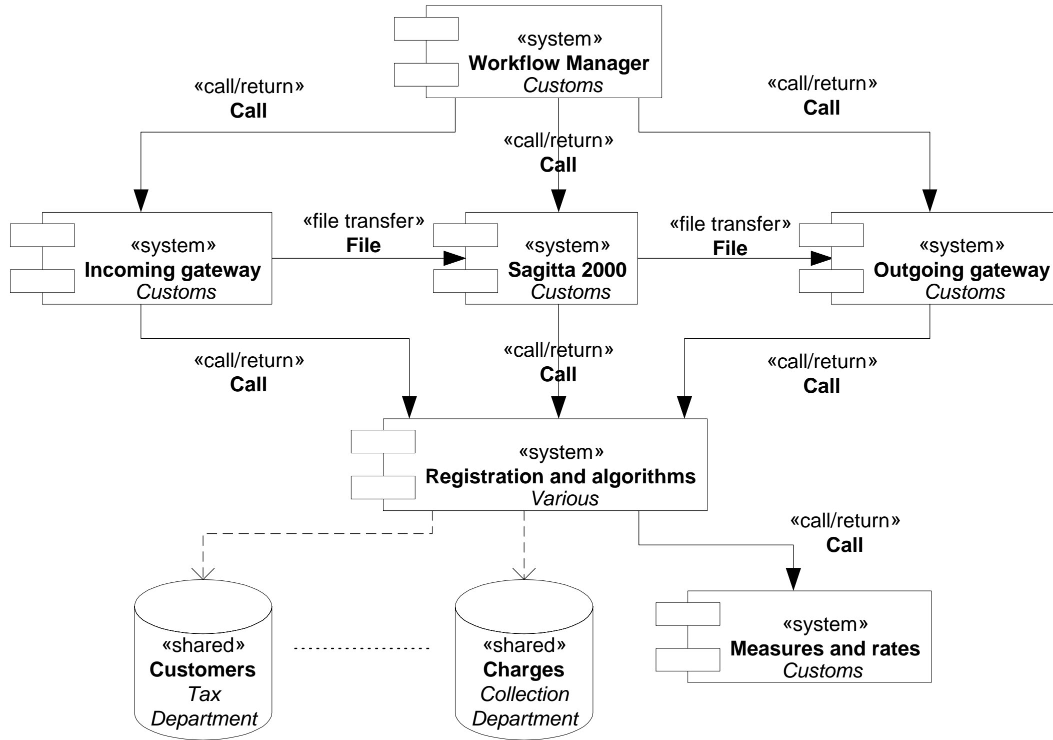

Figure 5: Context view of Sagitta 2000

of the Dutch Tax Department on behalf of the Dutch Customs. Sagitta 2000 supports processing of supplementary declarations of large companies regarding their imports and exports of goods. We were asked to analyze its modifiability.

This section discusses the four views for Sagitta 2000. Section 5.1 shows the macro architecture level views of Sagitta 2000 and section 5.2 shows the views at the micro architecture level. Section 6 demonstrates how these views were used in the analysis of Sagitta 2000.

5.1 Macro Architecture of Sagitta 2000

The context view of Sagitta 2000 is shown in Figure 5. Sagitta 2000 is surrounded by four systems: two gateway systems, a workflow manager and a system that provides access to central data registration and algorithms.

The incoming gateway receives all incoming declarations from customers. It translates these declarations to a generic format, after which they are transferred to the systems that handle the declarations, which is Sagitta 2000 for supplementary declarations. The outgoing gateway handles all communication with external entities. When a message has to be sent to an external entity, it is transferred to the outgoing gateway. The outgoing gateway then translates the message to the format and medium (for instance e-mail or snail mail) used to communicate with the addressee and sends it.

Currently, temporary gateways are used that were developed specifically for Sagitta 2000. It is the intention of the Dutch Tax Department to develop a common incoming gateway and a common outgoing gateway for all systems of the Dutch Tax Department (including Customs). When these systems are developed, this will be done under responsibility of a central unit. This approach has

both advantages and disadvantages. The main advantage is that a single version of each gateway has to be maintained. A disadvantage could be that the owner of Sagitta 2000 does not have full power of decision over the functionality of the gateways. The Dutch Tax Department has decided that the advantages outweigh the disadvantages.

The workflow manager controls the flow of declarations through the systems. Just like the gateways, the workflow manager that is currently used is only temporary and was developed specifically for Sagitta 2000. In the near future, a common workflow manager will be used for all systems of the Dutch Tax Department.

The registration and algorithms system is a kind of broker system. It provides access to a number of shared databases and systems that provide functions used by several systems. These databases and systems are not directly accessible by other systems, but their data and functions are made available through the services of the registration and algorithms system. Examples of these services are the customer service, which is used to access data about customers, and the rates and measures service, which determines the tax rates and measures that apply to a group of goods. These underlying systems already exist and the services that access these systems are developed under responsibility of their respective owner. So, the registration and algorithms system has various owners.

The other view at the macro architecture level is the technical infrastructure view, shown in Figure 6. This figure includes two types of nodes (or machines) on which Sagitta 2000 operates: workstations and application servers. The third type of node, database servers that store the data of the system, is ommitted to enhance readability.

All three types of nodes adhere to the standard that was defined by the Dutch Tax Department for the technical infrastructure. This standard is called Platform '96 and it prescribes the products that have to be used for the technical infrastructure of systems. This way, the Dutch Tax Department hopes to simplify systems management. Platform '96 currently prescribes Windows NT 4.0 as operating system for workstations, AIX UNIX as operating system for servers, DCE (= Distributed Computing Environment) as middleware, Encina as transaction monitor and Cool:Gen as development tool.

Figure 6 shows that each node includes a number of elements of the technical infrastructure in the form of run-time libraries. The DCE run-time files are included on each node, because they are required for communication between nodes. The run-time files of Cool:Gen are required on each node that runs applications of a system that is developed in Cool:Gen: in this case the workstations and application server. Something similar applies to the Encina run-time files, which are required on all machines participating in transactions, which are the application server and database server in Sagitta 2000. The workstations also include the run-time files required for an OCX-control[1] that is used in the workstation applications of Sagitta 2000.

The infrastructural elements shown in Figure 6 are also not independent of each other. Some dependencies are between infrastructural elements on the same node, others are between infrastructural elements on different types of nodes. An example of the first type is that the Cool:Gen run-time files only operate on a specific version of the operating system. An example of the second type of dependency is that the middleware components (DCE run-time files) must be the same on all nodes. Figure 6 includes dependencies of both types.

Another issue demonstrated in Figure 6 is that a part of Sagitta 2000 as well as a part of the Workflow Manager run on the workstations. This means that the dependencies between the Workflow Manager and Sagitta 2000 are not limited to the 'call/return' relationship indicated in Figure 5, but also include elements of the technical infrastructure running on the workstation. These dependencies are also important to consider when building modifiable systems.

[1]An OCX-control is component for the Windows environment

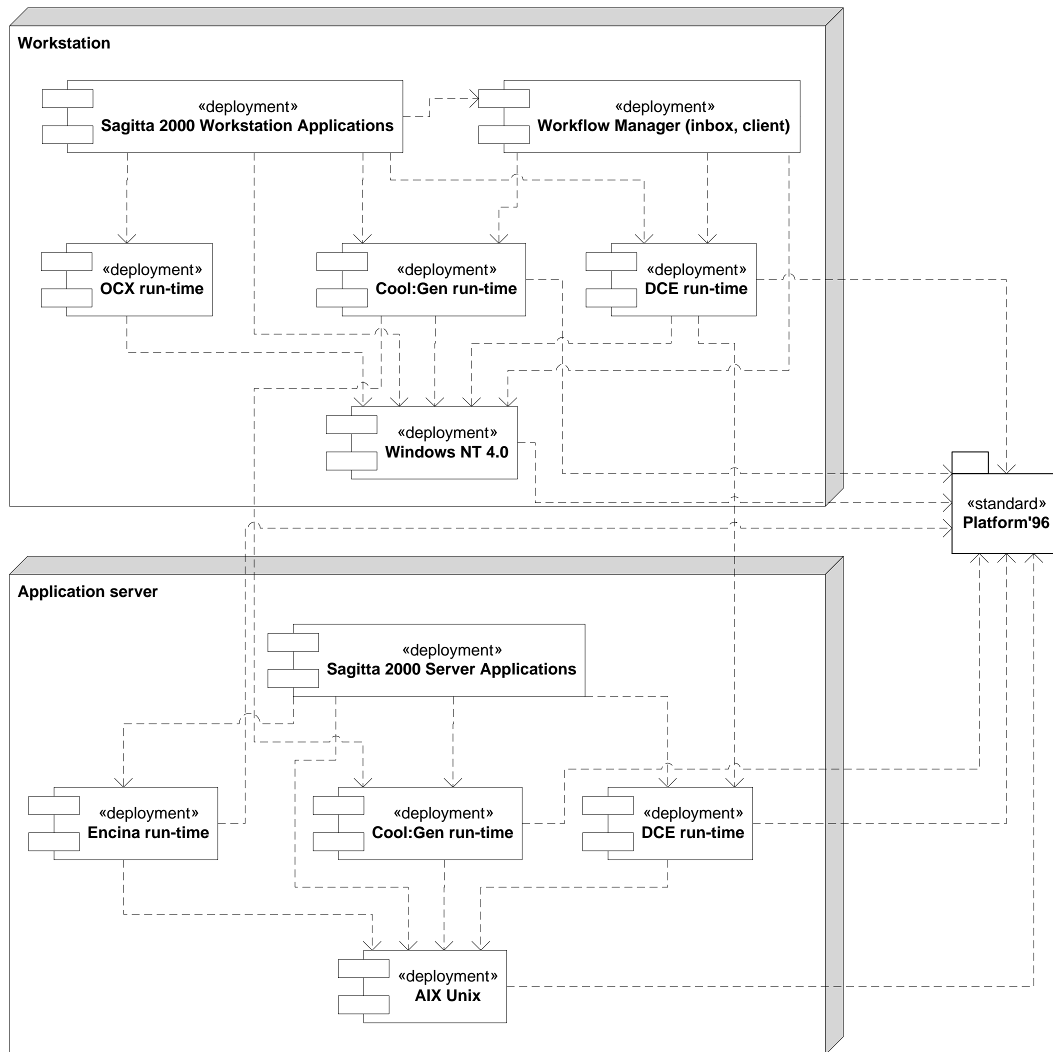

Figure 6: Technical infrastructure view of Sagitta 2000

5.2 Micro Architecture of Sagitta 2000

The conceptual view is shown in Figure 7. We see that a pipe-and-filter architecture is used for Sagitta 2000. The supplementary declarations enter the system from the incoming gateway, after which they undergo a number of transformations, and finally customers are informed about their declarations.

The transformations that the declarations undergo are the following. First, the declarations are converted to an internal format and registered in the system. The next step is to determine the sum of the charges, which is registered so that it can be recovered from the customer. After that, the procedure to be followed for the inspection is determined. The thoroughness of this inspection is determined by the risk of the declaration, which is caused by a number of factors, such as the highness of the charge. Subsequently, the actual check of the data of the declarations is performed according to the procedure determined in the previous step. Finally, messages are sent to the customers about their declarations.

The development view of Sagitta 2000 is shown in Figure 8. This view is a meta-model, it prescribes the *types* of components to be used in the implementation. For instance, access to data has

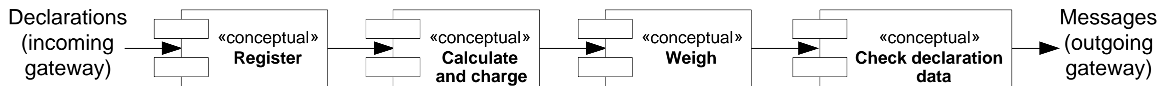

Figure 7: Conceptual view of Sagitta 2000

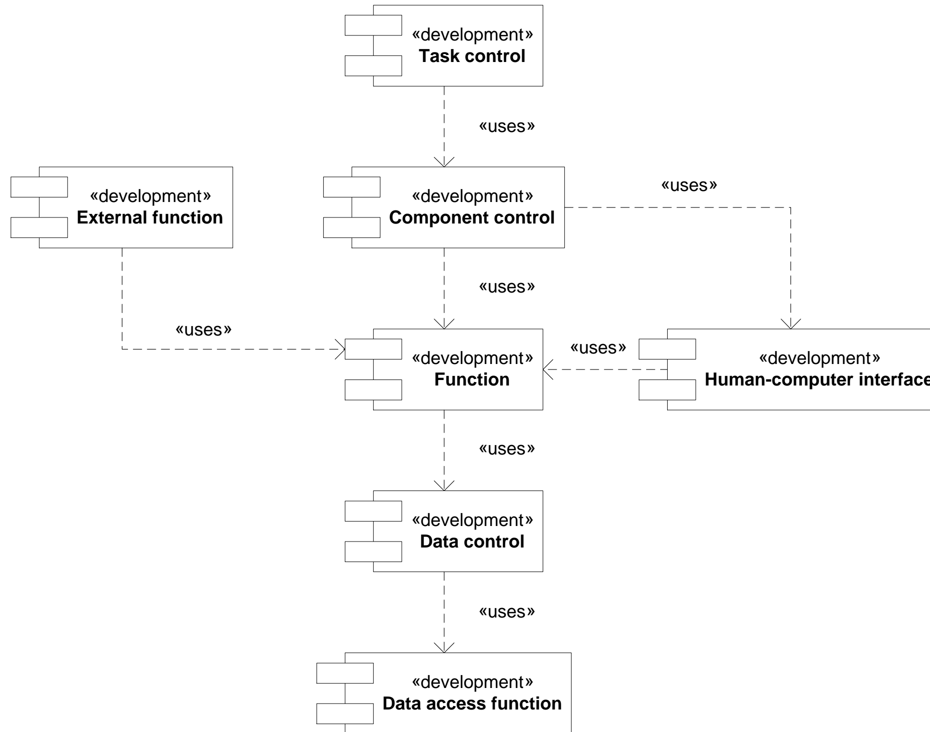

Figure 8: Development view of Sagitta 2000

to be implemented in separate components that do not perform any processing. Similarly, processing has to be separated from presentation. Since these rules cannot be inferred from the meta-model, this model has to be augmented by narrative text that explains the meaning of the various elements. This additional information should be known for a full analysis of modifaibility.

The components that were distinguished in the conceptual view, 'Register', 'Calculate and charge', 'Weigh' and 'Check declaration data', are implemented using component types distinguished in the development view. Some of the development components (instances of development component types) are shared by several conceptual components. An example of such a development component is the function component for calculating the highness of charges. This component is used by both 'Calculate and charge', to initially calculate the highness of charges, and by 'Check declaration data' to recalculate in case of adjustments to the declaration. So, the relationship between the conceptual view and the development view is not completely straightforward.

6 Using the Views in the Analysis of Change Scenarios

In this section we show how the above-mentioned views were used in the scenario evaluation step to assess the impact of change scenarios. Views that are required to assess the impact of change scenario

Table 1: The views required for evaluating changes in the functional specification

	Macro arch.		Micro arch.	
	Cont.	Techn.	Conc.	Dev.
F.1 What needs to be changed to include support for administrative audits in Sagitta 2000?	C/E		C/E	C/E
F.2 Is it possible to include support for fully automated processing of supplementary declarations? (without human intervention)	C		C/E	C/E
F.3 What needs to be changed to enable customers to submit their supplementary declaration through e-mail, through EDI or on paper?	C/E			
F.4 What needs to be changed to register additional information about clients?	C/E		C/E	C/E
F.5 What happens when the new European code-system is adopted?	C		C/E	C/E
F.6 What happens when the underlying datamodel of Sagitta 2000 is changed?	C		C	C/E
F.7 What changes are necessary to adapt the process belonging to the processing of supplementary declarations?	C		C	C/E

capture decisions related to the system's modifiability.

Based on the documentation of Sagitta 2000 and interviews we had with various stakeholders of the system (the architect, designers, and a representative of the customer) we drew up a number of change scenarios. These change scenarios are listed in Table 1, 2 and 3. For each of these change scenarios these tables indicate the views that were used to assess the impact of that scenario. Not all views are equally important for the evaluation of each change scenario. Sometimes, a view is irrelevant for a change scenario, sometimes it is only *consulted* to determine its impact, i.e. to determine which components/systems had to be changed to realize the change scenario. Sometimes a view is also used to *express* the change scenario's impact, i.e. the scenario affects decisions captured in the view. The letter "C" in a table indicates that the view was consulted to assess the impact of a change scenario. The letter "E" indicates that the view is used to express the impact of a change scenario.

In our analysis of Sagitta 2000, we identified three categories of change scenarios: (1) changes in the functional specification, (2) changes in the technical infrastructure and (3) other changes. The following subsections elaborate on these categories.

6.1 Changes in the Functional Specification

The first category consists of change scenarios that affect the functionality of the system. Such scenarios may require adaptions to existing functions of the system, the introduction of new functions in the system or the removal of obsolete functions from the system. Table 1 lists the change scenarios identified that affect the functionality of Sagitta 2000 and which views were required to assess their impact.

For assessing the effect of change scenario F.2, for instance, we use the context view, the conceptual view and the development view. The context view (Figure 5) is used to determine which systems

Table 2: The views required for evaluating changes in the technical infrastructure

	Macro arch.		Micro arch.	
	Cont.	Techn.	Conc.	Dev.
TI.1 What happens when the operating systems for workstations is changed in the standard for the technical infrastructure (Platform '96)?		C/E		C/E
TI.2 What happens when the operating systems for application servers is changed in the standard for the technical infrastructure (Platform '96)?		C/E		C/E
TI.3 What is needed to switch to another database management system?		C/E		C/E
TI.4 Is it possible to use 'thin-clients' for Sagitta 2000?		C		C/E
TI.5 What needs to be changed when in Platform '96 the prescribed middleware is replaced by another type?		C/E		C/E

are used for processing a supplementary declaration and which of these systems require any human intervention. It turns out that Sagitta 2000 is the only system that requires any user intervention, so the environment is unaffected by this scenario. Then we move on to the conceptual view (Figure 7) at the micro architecture level and notice that the component 'Check declaration data' is the only interactive component. To include support for fully automatic processing, an additional 'Check declaration data' component is needed that is capable of checking a supplementary declaration without any user intervention. In addition, the component 'Weigh' has to be adapted to include an assessment whether a declaration can be checked automatically, or that it has to be checked by hand. These changes have to be realized in the development view (Figure 8), which means that new task control components, component control components and function components have to be defined, and others have to be adapted. So, for change scenario F.3, we consult three views. The effect of the scenario is expressed in two views, viz. the conceptual view and the development view.

For assessing the effect of change scenario F.3, which explores the changes that are required for offering other forms of submission, we only use the context view (Figure 5). This view reveals that the supplementary declarations enter the Tax Department through the incoming gateway, which translates them to a generic format and transfers them to Sagitta 2000. When alternative forms of submission are to be supported, these have to be implemented in the incoming gateway. The interface between the incoming gateway and Sagitta 2000 remains unaffected, because the format of the declarations that flow between them is independent of the form of submission. So, this scenario does not affect the internals of Sagitta 2000 and the effect of the scenario is expressed using only the context view.

6.2 Changes in the Technical Infrastructure

The second category of change scenarios stem from changes in the technical infrastructure. These changes may come from a variety of sources, such as innovation of products used or changes in the organization's IT-policy. Table 2 lists the change scenarios of this type that we identified for Sagitta 2000. We will elaborate two of these scenarios, TI.1 and TI.5.

Change scenario TI.1 explores what happens when the operating system of the workstations

changes. To assess the effect of this event, we start by looking at the technical infrastructure view. Figure 6 shows that a number of elements depend on the operating system of the workstations. For each of these elements we have to determine whether they have to change for the new operating system. One of these elements is part of Sagitta 2000, namely the Sagitta 2000 workstation applications. These applications are generated by Cool:Gen based on a set of 'development components'. Cool:Gen can do so for a number of operating systems. If Cool:Gen supports the new operating system, we have to examine the development view (Figure 8) to assess which of these components have to be adapted and/or regenerated for this scenario. If Cool:Gen does not support the new operating system, the whole system has to be rebuilt for this scenario. The context view and the conceptual view are not consulted for this scenario because they do not include any information concerning the distribution of components over machines. So, for change scenario TI.1 two views are consulted, and both views are affected by the change.

Change scenario TI.5 deals with the changes necessary when another type of middleware is prescribed. The technical infrastructure view (Figure 6) indicates the elements that are dependent on DCE (the middleware currently used). These elements are the Sagitta 2000 workstation applications and the client inbox of the Workflow Manager on the workstations, the Sagitta 2000 server applications on the application server and Sybase DBMS on the database server. For all of these elements we have to determine whether they have to be adapted and whether that is possible at all. Sybase DBMS, for instance, has to support the new middleware, otherwise a major problem emerges. Adapting Sagitta 2000 to the new type of middleware is a matter of regenerating the system using Cool:Gen, provided the generator supports this new type of middleware. Again, we have to examine the development view to determine which development components have to be adapted and/or regenerated for this scenario. So, we use two views to determine and express the effect of the change scenario, viz. the technical infrastructure view and the development view.

6.3 Other Changes

The third category is a reservoir of change scenarios that cause changes that cannot be classified in one of the other categories. For Sagitta 2000 these scenarios are listed in Table 3. Most of these scenarios cause changes in one of the external components used by Sagitta 2000, and these changes may influence Sagitta 2000 as well.

Change scenario O.1, for example, concerns the integration of a new European system for handling transport declarations, called Transit, in Sagitta 2000. Transit is a standard package that is developed on the authority of the European Union and will be used by a number of countries of the European Union. The first view that is used to assess the effect of this scenario is the context view (Figure 5). This view helps to determine whether the functionality of this new system overlaps with the functionality of any other system. In addition, this view helps us to investigate the interfaces between Transit and other systems. It turns out that Transit has support for receiving incoming declarations and workflow management built-in. The first can be disabled in Transit but the latter cannot. This means that this issue has to be addressed when Transit is adopted. Another issue is that Transit has its own facilities for managing customer data. To avoid that this data has to be maintained twice, integration with the central customer database is desirable.

The next step is to explore the relationship between Transit and the technical infrastructure using the technical infrastructure view (Figure 6). This reveals that Transit uses some products that differ from those prescribed by the Platform '96 standard used by the Dutch Customs. Transit uses, for example, Oracle DBMS instead of Sybase DBMS that is prescribed by Platform '96. This issue can be addressed in a number of ways; e.g. Transit could be altered to use the products prescribed by Platform

Table 3: The views required for evaluating other changes

	Macro arch.		Micro arch.	
	Cont.	Techn.	Conc.	Dev.
O.1 What needs to be changed in order to integrate Transit in Sagitta 2000?	C/E	C/E	C	C/E
O.2 What is needed to make parts of Sagitta 2000 available to other systems?	C	C	C	C/E
O.3 What needs to be changed when RIN (new tax recovery system) is adopted?	C/E	C	C	C
O.4 What needs to be changed when BVR (new relation management system) is adopted?	C/E	C		
O.5 What needs to be changed when the new workflow management sytem is adopted?	C/E	C	C	C/E
O.6 What needs to be changed when the incoming gateway comes under the responsibility of the central department concerned with all inputs?	C/E	C		
O.7 What needs to be changed when the final incoming gateway uses another format than the temporary incoming gateway?	C	C	C/E	C/E

’96 or one could relax the obligation that Transit should adhere to the Platform ’96 standard. It is up to the Dutch Customs management to decide which of these solutions is chosen. Our next step is to investigate the conceptual view (Figure 7), in order to determine the overlap in functionality between Transit and Sagitta 2000. It turns out that Transit and Sagitta 2000 have no overlap in functionality, because Transit is aimed at other types of declarations than those supported by Sagitta 2000. However, it turns out that Sagitta 2000 and Transit do use common data. This means that it has to be decided where this data will be maintained: (1) in the databases of Sagitta 2000 or (2) in those of Transit. To investigate the effect of these decisions, the development view (Figure 8) should be investigated. When the first option is selected it is probably necessary that new data access components are added to Sagitta 2000 that fit the way in which Transit accesses this common data. However, if the second option is chosen the existing data access components of Sagitta 2000 have to be adapted so that they obtain their data from Transit. So, all four views are consulted for change scenario O.1, and three of them are used to express the effect of the required changes.

The impact of change scenario O.4 can be assessed using only two views, viz. the context view and the technical infrastructure view. The new relation management system replaces one of the systems from the registration and algorithms layer. The context view (Figure 5) is used to determine which systems are affected when this system changes. A number of systems use data from this system, but they do not access the system directly, they always use the associated service. Thus, when it is replaced by the new relation management system, the internals of this service probably have to be adapted but its interface can remain the same. This means that other systems, including Sagitta 2000, are unaffected by this change. The technical infrastructure view (Figure 6) is then used to assess whether this change affects one or more elements of the technical infrastructure. It turns out that the old relation management system, as well as the new relation management system, are installed on a different part of the technical infrastructure, and, therefore, that the elements of this view are

unaffected. So, the effect of this scenario can be expressed using only the context view.

6.4 Evaluation

Each of the aforementioned viewpoints is required in the assessment of at least some change scenarios. So, each of them captures decisions related to modifiability. We do not have a formal proof that the four viewpoints defined in this paper are always sufficient to assess the impact of change scenarios. Our experiences with software architecture analysis for three different systems (MISOC 2000 at the Dutch Department of Defense [13], EASY at DFDS Fraktarna [11] and Sagitta 2000 at the Dutch Tax Department), however, do suggest that these four viewpoints capture all relevant information.

The tables with the scenarios show a certain pattern in the views that are required to assess their impact. This pattern is not coincidental. In this case, the evalutation of change scenarios that affect functional requirements requires all views except for the technical infrastructure view. For scenarios that affect the technical infrastructure, we need to investigate the technical infrastructure view and the development view. For the remaining category we investigate all views. Again, we have no formal proof that these rules hold in general, but our other experiences show a similar pattern.

7 Conclusions

Modifiability is an important quality for business information systems. The foundation for a system's modifiability is laid by the decisions made in the software architecture. However, it is not always apparent in which areas decisions should be made in order to achieve modifiable systems. A solution to this problem is offered by view models: view models constitute a set of viewpoints that should be addressed in the description of a system's software architecture. This focuses the architect on a number of important decision areas. In particular, a view model that captures modifiability-related decisions highlights the decisions regarding this quality attribute.

In this paper we introduce four architectural viewpoints on business information systems that capture decisions related to a system's modifiability: the context viewpoint, the technical infrastructure viewpoint, the conceptual viewpoint and the development viewpoint. The latter two viewpoints coincide with viewpoints introduced before by other authors. All four viewpoints capture decisions that concern the allocation of functionality to components and dependencies between components.

The viewpoints are illustrated using the software architecture of a system whose modifiability we analyzed for the Dutch Tax Department. In this analysis we used ALMA, a scenario-based method for architecture-level modifiability analysis. Architecture description is an important part of this method: a description of the architecture is used to determine the impact of the change scenarios.

We have shown that each of the aforementioned viewpoints is required in the assessment of at least some change scenarios. So, each of them captures decisions related to modifiability. Our other experiences with software architecture analysis (MISOC 2000 at the Dutch Department of Defense [13] and EASY at DFDS Fraktarna [11]) suggest that these four viewpoints capture all relevant information. We claim that when these views are required in the analysis, the decisions they capture should also be considered during development.

In future research we want to further validate whether the identified viewpoints capture all relevant dependencies between systems and within systems, or that additional views are needed. In addition, our aim is to refine the semantics of the views used.

Acknowledgements

This research is mainly financed by Cap Gemini Netherlands. We thank the Dutch Tax Department and its IT-department for their cooperation. We are especially grateful to Harm Masman, Lourens Riemens, Jos Tiggelman and Harald de Torbal, for their valuable comments and input.

References

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

[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] P. C. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, March 1996.

[4] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1999.

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

[6] C. Hofmeister, R. Nord, and D. Soni. *Applied software architecture*. Addison-Wesley, Reading, Massachusetts, 1999.

[7] IEEE recommended practice for architecture description. IEEE Std 1471, 2000.

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

[9] R. Kazman, M. Barbacci, M. Klein, and S. J. Carrière. Experience with performing architecture tradeoff analysis. In *Proceedings of International Conference on Software Engineering '99 (ICSE99)*, pages 54–63, 1999.

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

[11] N. Lassing, P. O. Bengtsson, H. van Vliet, and J. Bosch. Experiences with software architecture analysis of modifiability. Technical Report HK-R-RES–00/10-SE, Högskolan Karlskrona/Ronneby, 2000.

[12] N. Lassing, D. Rijsenbrij, and H. van Vliet. Flexibility of the ComBAD architecture. In P. Donohoe, editor, *Software architecture: Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, pages 341–355, Dordrecht, The Netherlands, 1999. Kluwer Academic Publishers.

[13] 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.

[14] N. Lassing, D. Rijsenbrij, and H. van Vliet. Scenario elicitation in software architecture analysis. Technical report, Vrije Universiteit, Amsterdam, 2000.

[15] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1), January 2000.

[16] J. Nosek and P. Palvia. Software Maintenance Management: Changes in the Last Decade. *Journal of Software Maintenance*, 2(3):157–174, 1990.

[17] 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.

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

[19] M. van Steen, S. van der Zijden, and H. Sips. Software engineering for scalable distributed applications. In *Proceedings of the 22nd International Computer Software and Applications Conference (CompSac)*, 1998.