Complexity Management by well defined interfaces

Master thesis by Marijke van der Vliet

Department of Computer Science, Rijksuniversiteit Groningen
Research performed at Philips Medical Systems, Best

December 1st 2003

Supervisors:
Prof.dr.ir. J. Bosch (RuG)
Prof.dr. G.R. Renardel (second reader, RuG)
Dr. R.L. Krikhaar (Philips Medical Systems)

"The main thing is that everything becomes simple, easy enough for a child to understand" – Albert Camus.

Contents

1. Executive summary

Our general objective was to manage or reduce the complexity of large software systems. The definition and management of interfaces can play an important role in tackling the complexity problem. Therefore, interfaces should be well-defined, i.e. they should have clear *Interface Specifications*, be not dependent on (many) other interfaces, and be generic enough such that they don't need to be changed often. The *Interface Specification* makes the interface comprehensible, and usable for external users without needing to know the internals of the entity that implements the interface.

We have investigated what information should be given in so-called interface specifications, and how we can embed the description of these specifications in a natural way in the development process. We have studied two cases of interfaces at Philips Medical Systems (PMS), at the Magnetic Resonance (MR) department. We had interviews with the 'owners' of the interfaces concerned (who are responsible for the interfaces' quality) to learn about the context and contents of the interfaces. Then we had interviews with the external users of the interfaces to identify what information they needed to use the interfaces. Finally we had interviews with a group of designers and architects to identify how the management and specification of interfaces could be embedded in the development process.

From the results of the cases, we created a new template for Philips MR [Appendix D], which indicates what issues need to be described for each interface and what issues can be described optionally. Important is that the interface specification describes all a user needs to know in order to use the interface. This not only concerns syntactic information, but also semantic information, error handling, usage restrictions and so on. Such a complete interface specification enables the deployment of the entity that implements the interface as a black box, i.e. only providing executables, the interface and specifications and other deliverables, and hiding the internals (implementation and internal documents) to the environment in which it is deployed. This way of working facilitates people who were not involved in the development process to use the functionality provided, which is in particular useful when development is done in a geographical multi-site development, because then other communication possibilities are limited.

To ensure the interfaces and interface specifications are defined properly, the interface specification and interface management should be embedded in the organisation's development process, such that the developers know what to describe, when and where. Furthermore, to enable control of the dependencies and contents of the interfaces, these must be visible (in an overview) to the designers and architects. Then, it is important having the right decomposition of software functionalities; each composite having an interface that does not need to be changed often. Also, the interface should preferably be generic, and have configurability options (parameters) to make it more flexible for changes.

2. Motivations and research problem

2.1 Introduction

This research was initiated to improve the complexity management of software at Philips MR, by improving the software interfaces. It covers the definition of interface specifications, and the way in which their description and usage can be facilitated.

In section 2.2 we first explore the problem area of large software system development, and then we focus on the problems of software complexity. In section 2.3 we describe the current situation with respect to complexity at Philips MR. In section 2.4 we describe how interfaces can play a role in the management of software complexity. Then, in section 2.5 the current problems at Philips MR are described, followed by the assignment, which includes the concrete research questions. Finally, in section 2.6 we expose the generic research questions, and discuss which questions already have been answered by existing work.

2.2 Problem area of software engineering

2.2.1 Background: Increasing demands to software

The role of software development in world's industry has been changing from a supporting towards a prominent one; the hardware development still accelerates and improves the hardware, but the real distinction between final products is in the added value that the software provides [14,16]. This added value comes from e.g. fine-tuning and correcting hardware that contains noise or is not precise enough, or even from replacing hardware as the dominant component in many complex systems [33]. Thus software can be seen as the competitive edge of development. As a result, the requirements to the diversity of functions to be implemented grow rapidly, and consequently, the size and complexity of software increase very fast. When these software systems grow larger and larger, it becomes increasingly difficult and time consuming, to understand, maintain, and improve them. Also the trend to have international cooperation leads to even bigger systems, with even more developers working on it.

2.2.2 General problems in the development of large software systems

In Figure 1 we sketch problems that have emerged from the development of large systems with growing requirements. If possible, we also show solutions to those problems. As one can see, almost each solution gives birth to new problems asking for other solutions and so on.

One way to produce more functionality (in less time) seems to contract more developers. However, this is not only costly, but economic studies have shown that more developers on a project may even slow down the project, e.g. if the development of the product can't be partitioned further in subtasks [9]. Another way that seems to reduce development time is to reduce the effort in updating documentation. This may be successful for products with a short life cycle, but for (larger) products that evolve through many years, the lack of good documentation may lead to misunderstandings and therefore, result in delays of development, in the end.

A real solution is to use more efficient development methods. Known development methods which reduce development time – without reducing the software's quality! – are the following:

- Reuse of system parts [16]
- Development at a higher level of abstraction [18]
- Usage of a framework to implement commonalties between products [5]
- Independent development of system parts [37]

We elaborate on these development methods in chapter 4. Of course – as mentioned – the solutions given have their consequences too. One consequence is, most of the methods require a decomposition of the system into smaller components. Furthermore, the reuse of system parts and the parallel development of system parts require measures to control the integration of those parts in the system, or agreements upon standards of design and implementation such that the various developers understand each other's work. Otherwise, less consistent and less comprehensible code may be the result.

2.2.3 Problems regarding the software complexity

Now we have mentioned the consistency and comprehensibility of the software, we have entered the domain of software complexity. A common definition of software complexity says this is the degree of difficulty in analysing, maintaining, testing, designing and modifying software [22]. The software complexity is made up of the following parts [22]:

- Problem complexity: measuring the complexity of the underlying problem
- Algorithmic complexity: reflecting the complexity of the algorithm implemented to solve the problem
- Structural complexity: measuring the structure of the software used to implement the algorithm
- Cognitive complexity: measuring the effort required to understand the software

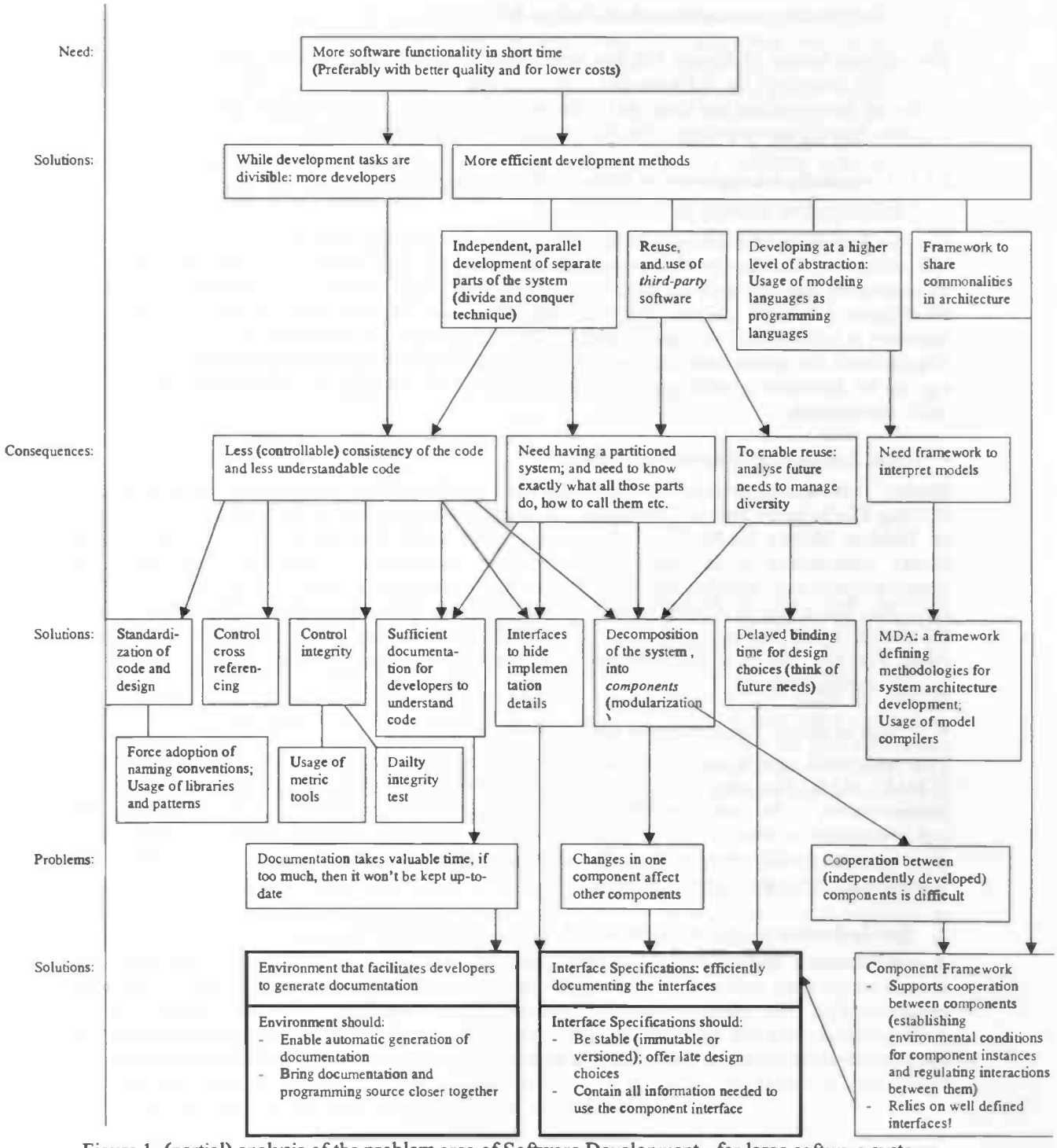


Figure 1: (partial) analysis of the problem area of Software Development - for large software systems

In this research, we focus on managing the structural and cognitive complexity, i.e. the complexity in software engineering. Our aim is to have software that is clearly structured (e.g. not too many interdependencies between software parts) and that is very comprehensible (e.g. usage of intuitive, consequent names and agreements on notations).

Two primary ways for humans to deal with complexity are divide-and-conquer (also called separation of concerns[1]) and abstraction [2]. The trend over the last thirty years in software engineering has been towards greater code modularity and greater information hiding. This lead e.g. to the partitioning of problems into simpler pieces, interfaces to be defined, and the implementation details to be hidden behind the interfaces [2].

2.3 Complexity management at Philips MR

The software system of Philips MR has been evolving over the last 20 years into a large complex system that comprises the software of a family of MR scanners. In these years of development, the number of functionalities has risen, and some parts of the software have become legacy code. In order to reduce the software complexity, the following measures have been taken:

2.3.1 Complexity management at Philips MR: historical overview

- Modularisation: showing part-of relations.

The idea behind modularisation is to breakdown software in manageable parts (i.e. components). The MR software system has been restructured about two years ago, which has resulted in a functional decomposition into so-called Building Blocks [52], which represent the functionalities that belong to the different MR units. Because these Building Blocks are organised hierarchically, also the whole code base is reflected this way; each Building Block being a directory that holds the code contents.
Together with this modularisation, goes the introduction of object oriented programming (if applicable, e.g. not for hardware specific code), the adoption of COM technologies and nowadays the usage of .NET environment.

- Cross Referencing: showing use relations

Another restructuring process has been initiated to make the use dependencies between different Building Blocks better visible. Therefore an abstraction step was added to the building block structure: all Building Block's header files (which among other things describe from which other Building Blocks functionality is included), which belonged functionally or conceptually together, were categorised into one "interface directory" within that Building Block. Now referring to a header file of a Building Block only is allowed through the corresponding interface directory of that Building Block. Which building blocks have permission to use which other building blocks, is registered in a special global directory, and these use dependencies are checked, of course. This concept is called (enforced) include scoping [50].

- Usage of Metric Tools: analysing code consistency

Two web-based architecture trend analysis tools, the Code and Module Architecture Dashboard (CMAD) and the Execution Architecture Dashboard (EXAD) are used to monitor changes, and detect inconsistencies, to the code base [30]. CMAD is used to examine changes of different code, module, and interconnection metrics. EXAD helps to keep track of changes in system performance and to relate them to recent modifications in the code architecture. When inconsistencies are found, a high priority should be given to repair these.

- Standardisation of code and design: analysing naming conventions

A simple measure to keep the system understandable for human beings is to set up agreements that prevent people from writing code or comments in all kinds of different ways (and, ergo, provide some consistency). At MR, coding standards [49] and design standards [51,43] have been introduced as rules to obey. These rules are not strict obligatory yet, but the results are being measured, in terms of the number of present violations. Thus, one is at least alerted to the present quality of code and design.

[1] See e.g. the Workshop on Advanced Separation of Concerns in Software Engineering at ICSE 2001.

- Consolidation process: guarding the system's integrity

To support a flexible cooperation of software developers, and to avoid interferences due to code changes, MR has defined a so-called Process Model [43], which prescribes to the MR software engineers how to cooperatively develop software, by working with the (version controlled) files from the "ClearCase" archive. To modify an element of the ClearCase archive, a developer first checks out the element from the Archive domain. Then he modifies the element in his own development domain (which is invisible to others!). When finished modifying, he has to submit his changes to the integrator. The integrator can accept or refuse the submitted changes, and if accepted, he can consolidate the newest element version (see *Figure 2*). The last consolidated element versions are the starting points for new development again.

To guard that the various parts of the systems that have been under development concurrently end up in a working system, the *daily build and smoke test* [20] is performed. This means that each day every file of the system is compiled, then all files are linked/combined into a single executable program, and finally that program is put through the smoke test – a test that exercises the entire system to expose any major problems. If the smoke test is not successful, then the first priority becomes to fix the problems! This way, the code integration risk is minimised, and incompatible code is identified early. Also it enables to add new functionalities step-by-step, which reduces the size of problems to be solved at once, and thus reduces the software complexity.

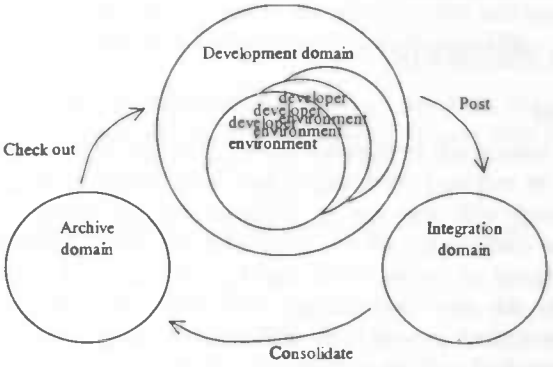


Figure 2: MR Process Model

2.3.2 Complexity issues left

The measures mentioned above have improved the comprehension of the code and its interdependencies. But they didn't solve the control problem: how to keep the system maintainable. The trouble when changing one of the system's parts is it often affects others. And because of the system size, and even more because of the concurrent development of releases, the consequences of such changes may be far-reaching, and difficult to survey. This yields even more when cooperation with international colleagues is involved, which is the case at the geographical multi-development sites with Philips' colleagues in Cleveland and Bangelore. Because the developers are not only miles away from each other, but also work in different time zones, the communication between them is very difficult. To make this concurrent development work, the functionality of software that is provided to others should be documented very clearly, and be as stable as possible.

Another aspect of maintenance comes from the fact that the developers of different interrelating parts of code do not necessarily know each other, or be familiar with each other's programming domain. This may also be the case when third party software is used or provided as such. The consequence is that the communication between these parts is required at a more abstract level than that of implementation. This would also facilitate the communication between other stakeholders, e.g. marketing and service engineers.

So a good communication medium, to communicate the software characteristics, and to control their stability, is needed. This medium should enable the different stakeholders to communicate the software functionalities between each other, including all side effects, e.g. the impact of changes in some part of the code on other parts of the code, the dependencies, interactions, and so forth.

2.4 The role of interfaces in complexity management

The good news is: such a medium already exists. We believe the proper media to communicate these functionalities between developers are the interface specifications. Sommerville [34] states that "clear and unambiguous subsystem specifications reduce the chances of misunderstandings between a subsystem providing some service and the subsystems using that service". And Clements [10] reinforces this, by stating that "well defined, unambiguous, interface specifications provide enough information to understand the implemented software", thus preventing that the software will be used in a wrong way or will be changed in such a way that other projects will be disturbed.

An additional advantage of good interface specifications, is the insight they provide to project planners. By having defined the way functionality is implemented, and what variations or configurations are possible yet, it is easier to estimate the effort needed to implement new functionality or changes to the present functionality.

Note that the interface specifications we are talking about now are beyond describing only syntactic information (such as e.g. IDL does). When necessary to make the distinction, we define that syntactic information as "the interface", and we all information that is needed to use that interface as "the interface specifications". These definitions are elaborated in paragraph 5.4. There, we also discuss different types of (syntactic) interfaces.

2.5 Problem area at Philips MR

2.5.1 Problem statement

Recently some effort has been made to improve the organisation of the MR software system by making the use dependencies of its software parts more clear. Also interface descriptions in UML have been created on subsystem level with the aim to facilitate the generation, updating and validation of documentation. Thus, the consistency of the code with the documentation and vice versa could be guarded. In addition, a model of the interface could provide a quick insight in the semantics of that interface – according to the say "one picture tells more than thousand words". However, the introduction of UML descriptions proved to be difficult; few designers knew how to use them, and the descriptions have never matched with the code archive totally.
Question was how the interface descriptions could be improved, and how they could be better embedded in the development process.

2.5.2 Assignment

The assignment at Philips MR was to clearly describe the required contents of the interface specifications for the MR subsystems, thereby taking the stakeholders wishes into account. Furthermore, the embedding of the creation of clear interface specifications in the current software development process must be described. This also might influence how the interfaces are described, e.g. in UML or in header files. With respect to the embedding in the development process, we would investigate the initial (creation) and evolutionary (modification) phase only, leaving the transition phase (from current to new situation), outside our scope.

As results we wanted to present a checklist with items that should be described in interface specifications, and an advice how to keep these specifications (more) up-to-date continuously.

2.6 Research questions

Our objective was to reduce software complexity by using well-defined interfaces, as described in section 2.4. The adjective "well-defined" can be interpreted in two ways: the syntactic interface contains the right functionality, and the interface is documented thoroughly. In this research we wanted to make both interpretations clear. Also, we wanted to study how the definition of interfaces and interface specifications could be placed in the software development process, and how the changes to interfaces could be managed. Therefore we posed the following questions:

1. What information is needed in interface specifications – such that it is enough information to use the interface, without knowing its implementation?
2. What is a good representation of the interface specifications? – such that the information is comprehensible?

3. How can interface specifications be embedded in the development process?
4. How can changes of interfaces be managed, and minimised?

Regarding the first question, we discovered very recent work had been done by Clements [10]. He suggests a standard organisation for interface documentation, which he presents as a list of all items that should be described. As this list was already very extensive, we decided to take it as a model for the interviews we planned to do at Philips MR, and evaluate if the list was usable there, or perhaps might be extended or shortened for different kind of interfaces.

Regarding the second question, we found no consensus on the right representation in literature. There exist languages such as IDL and UML that can be used to describe interfaces, but they cover mostly syntactic information. Thus, this issue was still open.

About the embedding of interface specifications in the development process, we haven't found many existing work. There is a lot of work done to describe waterfall, evolutionary, incremental and spiral models, e.g. in [34], but these are often not explicitly related to the software specifications. In [34] is described to design and specify interfaces between the development of an Abstract Specification (derived from Requirement Specifications), and Component design, but the reasons why interfaces should be specified then are not given. On the other hand, the Rational Unified Process (RUP), in [19], argues to accommodate changes early in the project, but then it doesn't mention to use interfaces to identify these changes. This leaves the issue to relate the specification and management of interfaces to development process models open. We have investigated this at Philips MR.

Regarding change management of interfaces, various studies have been done. We have described this in paragraph 5.5. In addition, we discussed change management issues with designers and architects at Philips MR.

Further details about the research approach are described in the next chapter.

3. Research Approach

3.1 Problem solving approach

Because this research has been done at Philips MR, the approach used was not only scientific, but also aimed at finding practical implications that could be used (to fund the theories). This asked of course for collaboration with the 'client' Philips, who was supposed to give feedback, so theories could be improved and we both would be satisfied. This approach is corresponding with that of 'building bridges' in [29], and also conforms to the so-called bathtub model[2]. This model prescribes to first translate the generic formulated problems to more concrete problems that can be identified within – for this research – Philips. Therefore we chose to study the general problems of instable and incomplete interfaces in the context of a case. We have formulated the concrete questions in the assignment. The answers retrieved from the case study we used to specify a generic solution, by considering the solutions in a more general context.

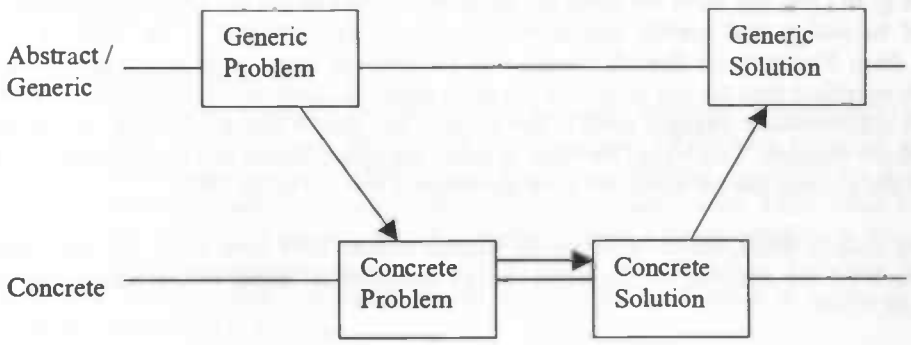


Figure 3: The bathtub model

3.2 Research design: turning research questions into projects

With respect to the first research question, we followed a fixed design. This means we defined a hypothesis or conceptual model before we got to collect data. When we started the research, we assumed that a good definition of the interface specifications would lead to a stricter separation of concerns in development, which would improve the software's maintainability. Our conceptual model was a list of items needed to describe in the Interface Specifications, according to Clements [10]. We used a non-experimental strategy, which means we didn't try to change the situation while we were studying the interfaces.

Because the specifications seemed likely to be different for various functionality domains within Philips MR, we tried to answer the questions given above, by applying them to two different cases, both in different functionality domains.

To answer the other research questions, we worked with a flexible design. That says, we didn't presume what the solutions could be, but worked exploratory, by learning from the developers what would facilitate their way of working, and by evaluating the ideas evolving from that, with other stakeholders (Philips designers and architects).

3.3 About the case studies

In a case study within the software engineering area, not only the technical solutions are important, but also the embedding in the software engineering process. To achieve this, the people working on the software system have to be convinced of the need for the solution. This requires insight to their way of working, as to facilitate the (preferably stepwise) introduction of the solution, and it also requires focusing on the advantages that the solution will give to them in the future.

To experience how our theories can be applied, we have done two case studies at Philips MR. A case study means a qualitative research method depending on sources such as observation, interviews,

[2] As teached by Prof. J. Bosch, RuG.

documents and the researcher's impression. The way, in which the theories appear to be useful, is concurrently a validation for those theories, thereby also resulting in a concrete solution for MR.

We have used convenience testing, which means we have selected those resources that were available. This means we would have no probability sample, and thus no random selection. Therefore no statistical generalisation to any population beyond the sample surveyed would be possible. However, we chose the interviewees such that they were a good representation of their group (of users of a specific interface). Therefore we first chose a domain (part of the software) to concentrate on, and then selected all kinds of stakeholders within MR, to interview them about their views on interface specifications. More detailed information about the selections can be found in paragraph 6.

3.4 Data collection

3.4.1 Questionnaires and checklist for the interviews

To get similar interviews, that can be compared to each other later on, we have made questionnaires. The questionnaire for the case interviews contained some closed questions, to ensure getting some answers that can be categorised, and also some open questions, to get ideas how to proceed. The questionnaire for the management interviews contained only open questions, which were meant to be guidelines for the interviews. That way we would get a survey of all management ideas. The contents of the questionnaires and checklist are discussed in section 6.2.

3.4.2 Organisation of the interviews

We started each interview with an introduction, in which we explained the goal of the interview and the context of the questions. For the cases, the context was the selected interface, for the other interviews the questions concerned the whole MR software system. After this introduction, we discussed the questionnaire. The duration of the interviews was about thirty minutes to an hour.

3.5 Interpretation of the data

Because of our choice for convenience test groups, we had to be very careful with generalising from the interviews' results. As for the results from the cases, they should be regarded in the context of those cases. For one of the cases we applied the results and ideas retrieved by then, to the Interface Specification of that case (see paragraph 7.4.2), and got that Interface Specification reviewed and authorised. As for the results of the interviews with other stakeholders, we wanted to present them as a survey, to show what the MR developers' and architects' ideas were with respect to interface change management.

3.6 Summary of activities

In Table 1 we have summarised all research activities in a timetable. The abbreviation "mgt" stands for management.

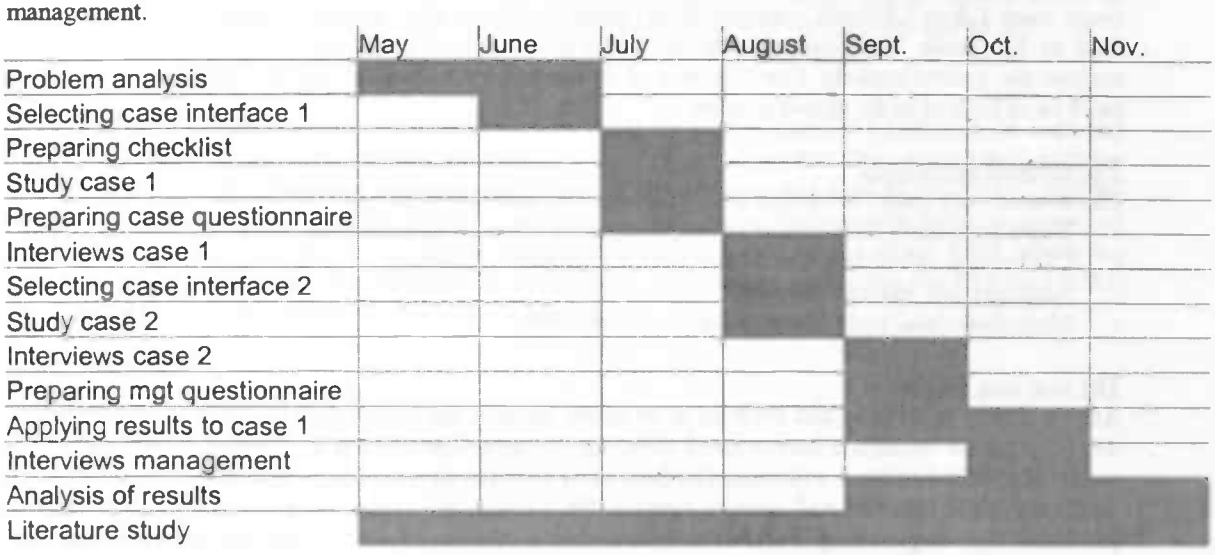


Table 1: Overview of research activities in time

4. Philips Magnetic Resonance

This chapter describes the research environment at PMS MR. First a description of Philips Medical Systems (PMS) and the Magnetic Resonance (MR) software system are given, followed by a description of the MR architecture and an overview of changes done to MR interfaces.

4.1 Magnetic Resonance background

4.1.1 Philips Medical Systems

PMS is a division of Royal Philips Electronics, Europe's largest electronics company. As a world leader in integrated diagnostic imaging systems and related services, PMS delivers portfolios of medical systems for diagnosis and treatment. Their product line includes technologies in X-ray, ultrasound, magnetic resonance, computed tomography, nuclear medicine, positron emission tomography (PET), radiation oncology systems, patient monitoring, information management and resuscitation products [45].

4.1.2 Magnetic Resonance technique

MR concerns scanning by using magnetic resonance. This technique originates from nuclear magnetic resonance (NRM), which was originally discovered as a technique for probing chemical structure, configuration and reaction processes. It identifies, and visualises, different chemicals on base of the frequency at which magnetic resonance absorption to those chemicals occurs. Only when Lauterbur modified a spectrometer in 1973, to provide encoded signals through a linear variation in the magnetic field, the first images of an inhomogeneous object (tubes of water) were produced. The development of Magnetic Resonance Imaging (MRI) scanners started a few years later, when it appeared that the technique could also be applied to humans.

The principle

Under the influence of the strong magnetic field and radio pulses, the hydrogen parts in the human body (or any other biological tissue) will send electromagnetic signals. A receiver and a computer interpret these signals by translating the relative response of specific nuclei to absorbed radio frequency into pictures with differences of contrast. In fact, one will get a tomographic map of distribution of protons. The big advantage of MRI (with regard to CT) is the possibility to show good contrast resolutions in weak parts of the human body. Another advantage is that the orientation of the scan is flexible.

The product

Philips has a large-scale software product family of Magnetic Resonance Imaging scanners. This scanner family is divided in two kinds of scanners: the Intera (closed MRI systems) scanners and the Panorama (open MRI systems) scanners. The different configurations of the closed systems have a range from 1.0 to 3.0 Tesla magnetic field strength, and the open systems' configurations range from 0.28 to 1.0 Tesla. The open systems are more patient-friendly, because the scanner tube does not enclose the patient entirely. Consequence of the open construction is that the magnetic field strength can't be as high as in the closed systems.

The medical application

The scanners are useful in various medical areas. The most important are:

- Neurology: early diagnosis of stroke, evaluation of dementia symptoms
- Cardiology: visualising blood flow, present cardiac morphology
- Angiography: vascular imaging
- Musculoskeletal and body: tumour characterisation

The scanning process

After a patient is selected and the scan to be made, defined, the patient that is laying at a table, will be moved stepwise through a human-sized tube. An electromagnet around this tube, consisting of coiled wires (also called coils) is activated. Gradient coils vary the strength of the magnetic field, while radio frequency coils transmit and receive signals. By varying the coils' types different images can be produced (by emphasising different physiochemical characteristics of specific protons, and thus assuring exceptional tissue contrasts). Applications are e.g. to view blood flow, and compensate for blurring effects of cardiac and respiratory motion. The data achieved is reconstructed to images, and

then archived. The process described here is also drawn in Figure 4, in the context of the MR software system.

4.2 The MR System

The execution of the scan can be translated to various functional steps. First the hardware (magnet and coils) has to be controlled to produce data, which then has to be acquired, and finally be transformed into images. Beneath the processes of the MR scan are shown in such a way that the relation of the functionality to the MR system becomes clear.

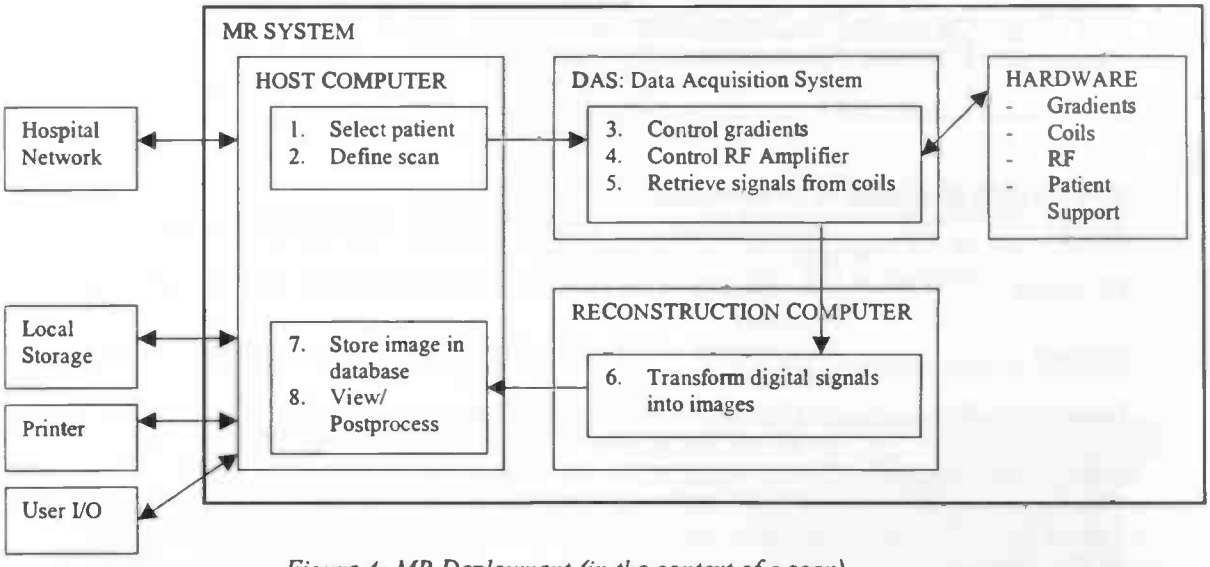


Figure 4: MR Deployment (in the context of a scan)

The host computer is connected to the hospital network. This host computer is the computer the employees at the hospital use. At this computer the patients can be selected, and the scan to be made is defined. When the scan is started, the host computer starts up the Data Acquisition System (DAS).

The DAS controls the hardware to acquire the scan data. It comprises:

- Control of the patient table
- Control of the gradients
- Control of the RF amplifier
- Retrieving signals from the coils

The Reconstruction computer transforms the received signals into images, which can be saved or viewed at the host computer (that is connected to the hospital computers and printers and so on).

4.2.1 MR Architecture view

Since 1996, all medical imaging systems of MR, from low-end (diagnostic systems) to high-end (systems used for complex medical interventions) are built upon one product line architecture, with the purpose to handle the increasing size and complexity of the systems, and also to reduce the time-to-market [8]. As described in this paper's introduction, the MR System has been decomposed functionally into building blocks. In *Figure 5* you can see all building blocks at top level, which are therefore called subsystems. The functionality described in the previous subsection can be found in this figure as well, although it might be in another shape.

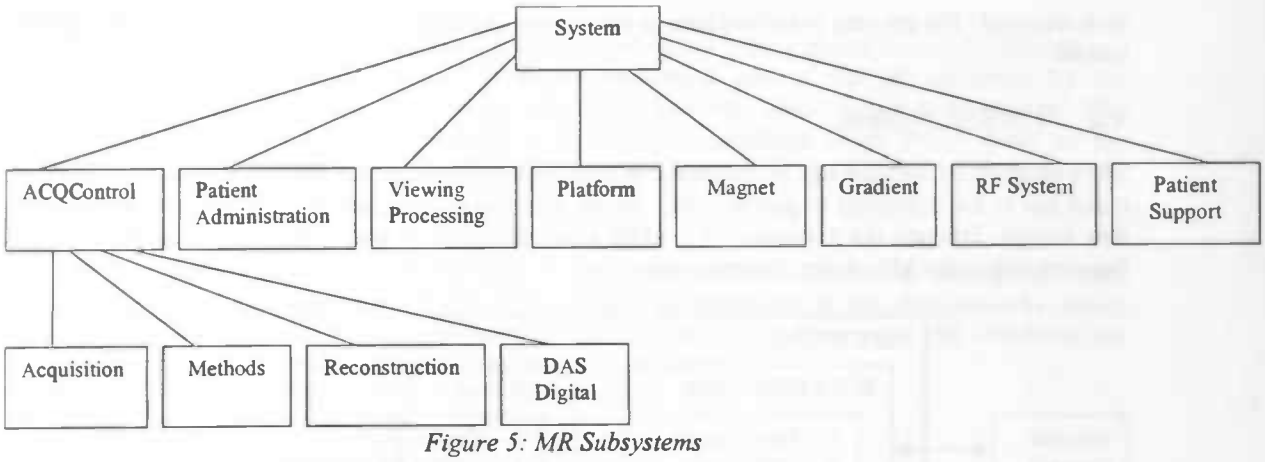


Figure 5: MR Subsystems

Here is a short description of the subsystems:

Magnet	All parts necessary to supply an accurate stationary homogeneous magnetic field.
RF System	The generator of RF pulses in the imaging volume, and receiver of the RF responses.
Gradient	The generator of dynamic gradients of the magnetic field in the imaging volume.
Patient Support	The device to carry the patient, the monitoring of patients physiology and the interaction with the patient.
Patient Administration	The maintainer of all patient related data.
Viewing Processing	The optimiser of raw data from the reconstructor (digital image processing)
Platform	The basic environment for the development of the application software.
ACQControl	The control and implementation of facilities to define and perform a scan.
- Acquisition	The facilities for defining a scan list and the execution of a scan.
- Methods	The handler of physics aspects of the definition and execution of a scan
- Reconstruction	The transformer from the digitised MR signals to images, and/or MR spectra.
- DAS Digital	The Data Acquisition System.

The building blocks that are most relevant to the software department are Acquisition, Methods, Reconstruction, Patient Administration, Viewing Processing and Platform.

4.3 Monitoring MR Interfaces

In a pre-research we have identified the number of interface modifications during a project's life cycle [56]. Therefore we chose to monitor one project that was in the last phase of development, so we would get an overview of recent changes. To retrieve and analyse the modifications, we developed a script program, CAPI (Change Analysis Program for Interfaces). Beneath, we have shortly described how we have analysed the interface modifications and at the end we have given the main results and conclusions.

4.3.1 Retrieving all interface modifications

For each file in the software archive, the configuration management system "ClearCase" registers what modifications have been done to it. An overview of these modifications can be represented by ClearCase in a graphical representation or in a text file. As it is difficult to analyse graphical representation automatically, we decided to analyse the modifications from text files. For the information extraction from the modification files we used the scripting language Perl, because that supports a quick interpretation of text fragments. We generated these modification files for the (external) interfaces of the six most important subsystems.

4.3.2 Analysing the interface modifications

The modification files we retrieved from ClearCase contained all modifications to all header files of the selected interface in chronological order. But this list of modifications also included modifications that were done in the developers' domains. Thus it could occur that files had been modified on the same

date, but only one of them was in the system archive (i.e. it was consolidated), while the other one was still in the developer's domain (for an overview of the MR Process Model that shows the relations between different domains, we refer to *Figure 2*). In that case, we only wanted to report the modification of the file that had been consolidated. Therefore we had to keep, for each period we wanted to report, a record of each file, in which we could administrate the number of modifications done during that period. As begin and end marks for a period we used the configuration baseline labels that mark the files that belong to the same version of the project.

We intended to make the CAPI script as flexible as possible. Therefore we used parameters to choose which project, which period and which 'level' (building block, interface or header files) should be monitored. We also provided to choose which kind of modifications had been done: an addition or removal of a header file, a merge of a header file (from another stream to the monitored project stream) or another change of a header file.

4.3.3 CAPI Results

As we wanted an overview of the interface modifications for the project's life cycle, we chose to show the total number of header files that had been changed for all the different consolidation-periods. We also generated a view of the total percentage of files that had been changed for all the different subsystems. This gave the following results for the projects Aquarius2 and its successor Taurus (see Figure 6 and Figure 7).

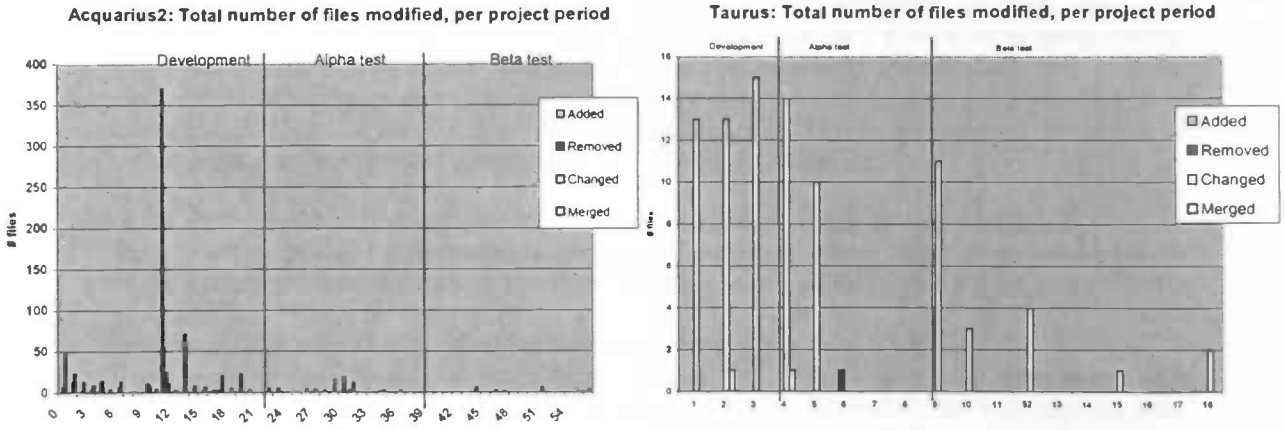


Figure 6: The numbers of files modified per project period, for resp. Aquarius2 (on the left) and Taurus (on the right).

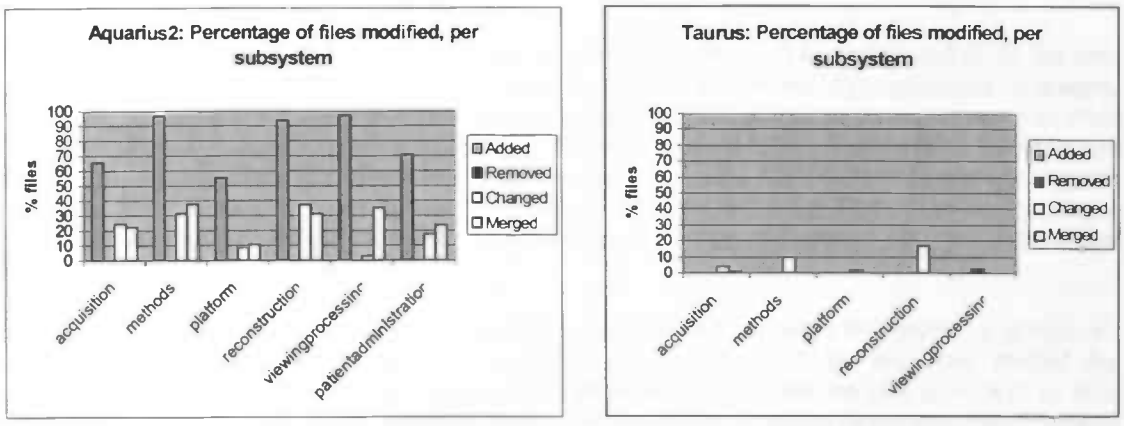


Figure 7: The percentages of file modified per building block, for resp. Aquarius2 (on the left) and Taurus (on the right).

4.3.4 Conclusions & Discussion

- A substantial amount, about 30% of the about 400 subsystems' interfaces were changed during a project's life cycle (of Aquarius2). Also in Taurus, which was a considerably smaller project, several changes to interfaces were done.

- The number of changes decreased in time, but changes during testing phases were not excluded.
- The high amount of added interfaces in the Aquarius2 project can be explained by the reorganisation of the system's external interfaces that was done during that period. The reorganisation concerned the movement of all external interfaces from lower levels to the subsystems they belonged to (which were exact those we monitored).

4.3.5 Application

The tool that we have developed will be used for monitoring the MR system, in combination with several other tools. Some guard the stability of the include-dependencies, or the compliance to coding standards, and our tool will show the stability of the interfaces. If at some point the interfaces are changed more often than normal, further investigation can be done.

5. State of the art in Interface Management & related work

This chapter describes existing work on complexity management and interface management. We start with a section in which we show how complexity can be decreased by increasing the abstraction level of code, design and architecture. Then we discuss the evolution of software development towards component oriented and domain oriented software. In the following sections we stress the role of well-defined interfaces, and explain how these interfaces can be managed. Finally, we summarise the techniques described and relate them to our research and to the situation at Philips MR.

5.1 Raising the level of abstraction in software engineering

The idea of increasing the level of abstraction is simple: it allows one to concentrate on the problem as a whole, without going into details, which leads to an increase of productivity and correctness. The consequence, of course, is that the details still have to be implemented too. Nevertheless, the benefits from designing at a higher level of abstraction dominate, especially since techniques exist to generate implementations (partly) automatically. In the next subsections we have elaborated further on the developments in designing at a higher level of abstraction.

5.1.1 Generations of programming languages

Programming languages have evolved through abstraction for years. First there only was machine code. Second, assembly languages used mnemonic coding for the individual machine dependent constructions, which made the code much easier to understand. The third generation of programming languages (3GL) used compilers and interpreters to convert high-level specifications into machine assembly code, enabling programmers to concentrate more on the essence of the application rather than on constraints and syntax of underlying hardware and technologies. This greatly reduced the number of lines of programming required, and again facilitated the code understanding.

Within the third generation languages the following programming techniques were developed:

- Generic programming; i.e. language independent programming (where the language is a parameter of the system, so instantiation with language definition is needed as to obtain a language-specific system) [7].
- Structured programming; e.g. object oriented programming (e.g. Java, C++)
- Event based programming; the programmer designs the user interface, and determines what happens at which event (e.g. mouse click on object, a certain keystroke); the user chooses what happens, in which order and when by using the user interface (e.g. Visual Basic).
- Programming closer to natural languages. Mainly to facilitate accessing databases, because a need arose to enable access to databases to more people, and this way also non-programmers could use them.

These techniques have improved and facilitated programming, but can't be considered to be the next abstraction step in programming. That step is made by the new generation of programming languages, the 4GL's. The 4GL's are non-procedural languages, which allow programmers to express just what they want, instead of describing a detailed picture of how to produce it [42]. For example, the language Prolog [36] accomplishes this by offering programmers to express their ideas in logical statements instead of using terms that define computer operations. Other 4GL's are in fact modelling languages used as programming languages rather than merely as design languages, thus performing the abstraction step from 3GL to 4GL [13].

For the interpretation of the developers' ideas that are expressed (visually) by models, a (graphical) development environment is needed that generates the appropriate code. We have studied the generation of code from models in paragraph 5.1.3. Because in the abstraction step from 3GL to 4GL even more implementation details (the high-level code) are hidden from the developer, the developed "code" (that is: models) will be much clearer and easier to understand, and is, thus, less complex.

To allow programmers designing even closer to the problem domain, a natural language component can be added to the graphical formalisms. This combination of (formal) visual representation of software with natural language is called a semiformal visual language [15]. Another software paradigm aimed at corresponding to the developers' natural view of concerns, and thus providing a new style of modularisation, is Aspect Oriented Programming (AOP) [6]. Aspects are abstractions that capture and

localise crosscutting concerns, e.g. code that cannot be encapsulated within one class, but that is tangled over many classes. Classical examples of aspects are synchronisation (concurrency handling), failure handling, logging, and memory management. For the developer the implementation of an aspect is a separate concern. During runtime, the separate aspect representations have to be processed in order to produce the concept that is described by the crosscutting aspects [11]. This compilation process is referred to as weaving. The goal of weaving is in fact the same as the goal of generators: to compute an efficient implementation for a high-level specification. Extra advantage is that this weaving can also be realised by run-time interpretation of the aspect programs [11].

The next step in programming languages (5GL), would involve computers that respond to spoken or written instructions, or natural language commands [21].

5.1.2 Frameworks

Because modelling languages represent concepts and data relationships in the problem domain space, a whole framework, rather than a simple conventional compiler, is needed for getting this high level language to be executed by computers directly [41]. Therefore, we have also discussed (modelling) frameworks and issues regarding the automatic generation of code from the models created by these languages, later on.

A (object oriented) framework, is a set of cooperating classes, some of which may be abstract, that make up a reusable design for a specific class of software [5]. Frameworks represent a large-scale form of reuse. They are generally used and developed when several (partly) similar applications need to be developed, e.g. in a software product-line. A framework allows capturing the commonalties between these applications, thus reducing the effort to build the applications, and increasing productivity. It can be seen as a frame with basic functionality on which the developer "mounts" components [35] (see Figure 8). For now it is enough to think of component as a piece of software, e.g. a module implementing some functionality. We have made a stronger definition of components later on.

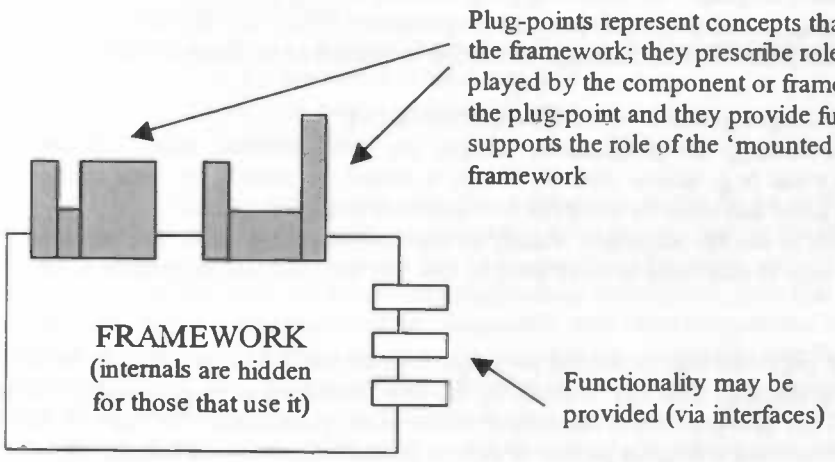


Figure 8: Conceptual model of a framework

Besides the advantage of reuse, frameworks can also be very useful to guarantee adaptability of the software, e.g. on various platforms, when they provide methodological and technical support for working directly on analysis and design models without being dependent on technological and infrastructural decisions.

5.1.3 Automatic code generation

Because designers are comfortable with the languages and concepts used in 4GL, and they don't need to specify how their ideas should be implemented, development can be realised very fast. But then, generation methods are required to generate code form the created models. The generated code should, preferably, be executable as it is. Thus, it can give a rough idea what the (final) application will look like, which is useful for evolutionary prototyping (generating the integral application, and let it evolve through a succession of increasingly refined phases). This prototyping encourages the participation of end users in the development process, thereby reducing the likelihood of rejections.

Currently, only a few tools are available to generate code from models. Formal model drive generators are used to produce 3GL code, HTML, XML, IDL etc. Bettin [3] remarks that generative techniques are only limitedly accepted in the majority of software development organisations. This may be due to the investments that have to be done (such as buying tools, educating the new techniques to the software developers), while the usage of generative techniques has not established in the current industry yet. Thus the benefits don't outweigh the investments yet.

Also the maturity of the modelling tools is still an issue. Bettin [3] warns to use (old) UML modelling tools that have the same level of abstraction as the implementation, because they will only be useful as a drawing tool for post-implementation documentation, as they can represent only little more than instances of design patterns used in implementation. However, UML design models that are implementation language independent, can be used to generate implementation structures in potentially more than one target language, if also qualitative code generation techniques are available to map the UML models to those target languages. Thus, in [3] the point is made that, the (higher) level of abstraction of the UML models determines the (higher) achievable gain in productivity.

About the maturity of UML, Skipper [33] states that there are still deficiencies to be addressed. E.g. UML does not support use cases with a high degree of interaction between the user and the system, use cases of dynamic simulation, use cases with logical constructs (such as loops or concurrent operations). Also UML does not provide the consistency and integrity between use cases and class models, thus in [33] is concluded that UML does not satisfy the criteria yet to be used for systems engineering. Organisations as OMG and Rational Rose are working to address the deficiencies of UML.

The newest development in code generation is the area of generative programming (GP) that is aimed at creating reusable software solutions. This area, however, is also very immature. Building on the product-line approach, GP complements object-oriented methods with notations and techniques to perform domain scoping and feature modelling. It also provides techniques for deriving a common family architecture, and for automatically assembling components [11].

5.2 Component oriented software

Another way to develop software more efficiently is to organise software functionalities into components, that are standardised and reusable. Standardisation and reuse accelerate the software development process. Reuse accomplishes not having to design and implement the same functionality again and again, and standards provide stability to the development of basic software. When this standard software domain grows, companies are encouraged to shift their expertise (competitive edge) to more specialised areas, which leads to a more mature software market. As a fact, the use of components is a law of nature in any mature engineering discipline [37].

5.2.1 Emerging from ad hoc programming to programming system's products

Now we have stated how useful it is to have standard reusable assets, how do we realise components that comply with these requirements? As mentioned, the development of components is a maturity step to be taken in the software engineering industry. In general, we can distinguish two directions in which programming can evolve: towards a programming product, or towards a programming system (see Figure 9) [9]. Both ways the product is converted into something more useful, but also more costly.

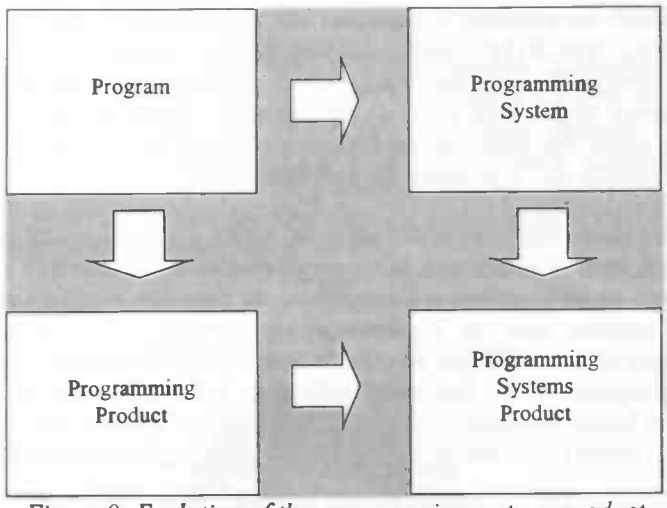


Figure 9: Evolution of the programming systems product

A programming product is a program that is generalised as much as the basic algorithm reasonably allows, is thoroughly tested and documented such that anyone may use it, fix it, and extend it. A programming system is a collection of interacting programs, which are coordinated in function and disciplined in format, such that they can interact with each other through precisely defined interfaces. When a program is evolved in both directions, the result is a programming system product, which is a collection of interacting programming systems. A rule of thumb says the development of a programming system product costs about nine times as much as just a program, but then it is a truly useful object [9]. In this context, we can see a component as a programming product with a contractually specified interface. In the next subsection we have defined components in more detail.

5.2.2 Software components defined

Szyperski [37] defines a software component based on the following characteristic properties:

- A component is a unit of independent deployment; and
- A component is a unit of third-party composition[3]; and
- A component has no persistent state.

This first part of the definition implies that the component needs to be well separated from its environment and from other components, and therefore must encapsulate its constituent features. Also, to be a unit of deployment, the component has to be machine executable without human intervention (extra information can be provided in so-called deployment descriptors), such that it can be installed in any environment. As the component must be composable with other components by a third party, clear specifications of what the component requires and provided are needed: the interface specifications that enable communication with the environment. Finally, the last part of the definition means that the component should be functioning as an immutable plan, and not as a mutable instance. Thus when a deployed software component gets installed on many systems, it remains invariant. Also, components live at the level of modules or classes, and not at the level of objects.

As one can see, this definition of a component guarantees the component to be independently developable and thus easier to reuse too. An important consequence to the development organisation for components is that it should be black box oriented, that says: only relying on interfaces and specifications. So implementation code and other internals are not available (which would be white box). The role of interfaces in the development of components is discussed in paragraph 5.4.

5.2.3 Component maturity

Currently the market of software components is rather premature [37]. Time and effort are needed to evolve the software engineering discipline, resulting in standard components. In hardware standard modules like pc's or memory charts are much more accepted. In software the best example of development towards component industry are operating systems: they already are accepted as standards

[3] In this context, a third party is one that cannot be expected to have access to the construction details of all the components involved

that are used as basic software that one doesn't write by oneself[4]. This is just the beginning of the component market development.

Szyperski [38] distinguishes four levels of component maturity:

1. Maintainability: modular solutions
2. Internal reuse: product lines
3. A) Composition: make and buy from (a-1) closed pool of organisations or (a-2) open market
   B) Dynamic upgrading
4. Open and dynamic

According to Szyperski the state of the art of many software solutions is at about level 1 or 2 of this component maturity model. The practical use of components today tends to end with the deployment of components (level 3). To make components truly deployable, especially the components' quality needs to be assured. Therefore component unit tests and other forms of quality assurance, such as verification and component, are of critical importance.

The different degrees of component maturity correspond with different uses of software architectures. The modular solutions can be applied in the software architecture of an individual software system, internal reuse can be realised in a product-line architecture, and finally the public component market concerns the development of standard architectures, which Szyperski calls component frameworks. In the next subsections we discuss both component frameworks and the usage of components in a product-line architecture.

5.2.4 Component Frameworks

A component framework is a partial design and implementation for an application in a given domain, which consists of components [5]. In essence it can be seen as a dedicated and focused architecture that supports components to certain standards, and allows instances of these components to cooperate. Therefore, the framework provides the required shared knowledge to couple components, define rules of interactions and possible configurations. This knowledge exists of standard component interfaces, connectors and composition rules. The final application(s) will exist of components and scripts that connect the components (generating so-called glue code).

5.2.5 Components in software product families

Components are not only interesting for third-party usage, but also for own usage, within software product families [4]. A software family typically concerns a set of related products or system. Because they are related, they can be built upon a common architecture that enables fine-tuning of the components to the different products. As each of the products is built upon the same architecture and makes use of shared components, this product line approach leads to increased productivity, time-to-market and software quality.

The development of the product line involves the definition of the product-line architecture, the set of components and the set of products [5]. The product-line architecture should cover all the products and include features that are shared between all products. The selection of the product features to be included in the product-line architecture is called scoping. For the definition of the components, the provided and required functionality should be made clear, and also the *variability* of the components should be made explicit (see 5.5.3), such that the differences between the various products can be taken into account. For the final products of the family, ideally the product-line architecture can be used 'as-is', by using only the variability of the components to express the differences in implementation between the products. But the product architecture may also be different, e.g. having its own component implementations or extending the product-line architecture.

5.3 Domain oriented software

Domain oriented software is about using meta-data to drive component oriented software. The idea is that the module and types can be defined in an abstract meta-language, and then the implementation is rendered using a code generator. However, domain oriented software is more than generative software, because it can also generate code for different implementations. And it is more than component oriented software, as it can generate different implementations for the same interface [46].

[4] Bits & Chips, "Bergson, Fancom en Nyquist zetten softwareproductiestraat op". Juli 2003.

5.3.1 Model driven architecture (MDA)

A MDA can be seen as a framework for defining system architecture development methodologies, by giving directions for making architectures, serving as a toolbox for system developers and system architects, and facilitating integration and interoperability between systems [31]. Thereby, MDA separates structure and function of a system from its technical realisation, to facilitate a unified process of analysis and design [39]. MDA is based on modelling languages (like UML) and other industry standards for visualising, storing and exchanging software design ands models. The idea is to enable designing without having to use technology specific code.

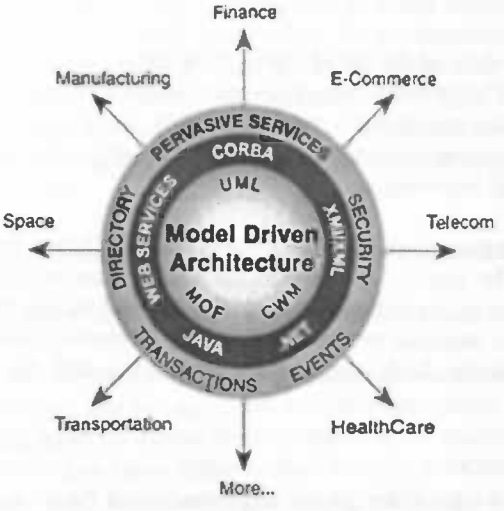


Figure 10: The Model Driven Architecture

By providing these methodology and technology, the MDA enables [44]:

- Technology neutral designing, independent from implementation technology: the specification of the functionality of a system is separated from the specification of the implementation of the functionality of a specific technological platform. This makes it easier to rapidly include emerging technological benefits into existing systems.
- Storing models in standardised repositories. There they can be accessed repeatedly and automatically transformed by tools into schemas, code skeletons, test harnesses, integration code and deployment scripts for various platforms.
- Addressing the complete life cycle of designing, deploying, integrating and managing applications as well as data using open standards *(better, faster, cheaper)*.

The main advantages of these features are higher productivity (thus reduced development time for new applications, and reduced development costs) and better maintainability (e.g. easier reuse of applications, and thus reduced management costs). We should note, however, that the development of the MDA is still very premature, and is only adapted within the Object Management Group (OMG).

5.3.2 Model-based development process

In a model-based development process, models are created for capturing not only requirements, but also designs and implementations, thus covering the full life cycle of software development. The results can be seen as living documents that are to be transformed into an implementation [1].

Three different types of model-driven development can be distinguished [40]:

- Conceptualisation: an object oriented domain model is designed in, e.g., UML.
- Blueprint: after conceptualisation, a blueprint (or framework) of the software is generated from that. Remaining pieces are programmed by hand.
- Specification: an object oriented domain model and execution specification are designed in UML; the software is directly generated from there with no manual programming involved.

Here we have studied the whole life cycle of model based development, that is resulting in a specification. The process is anchored on two models, the Platform Independent Model (PIM) and the Platform Specific Model (PSM). The designers first create a PIM of the design. This is an abstract model of the software design that omits any platform, i.e. implementation-specific, details. The PIM then is refined into a PSM (See Figure 11). This is a mapping, accomplished by associating the PIM to

the characteristics of the chosen platform. The PSM contains then, in fact, the same information as a coded application, but then this information is in the form of (e.g.) UML. The model transformation technology that connects platforms and modelling tools is an active area of research.
One could also apply mappings from PIM to PIM or from PSM to PSM (which are only model refinements during the development life cycle respectively during the deployment of components), or from PSM to PIM, which implies reverse engineering.

Finally, when the PIM is created, commercially available model driven generation tools depending on the chosen platform and specific technology can do the code generation. Unfortunately there is not much tooling available yet, beyond UML modelling and skeleton class generation. Also tools for verification and validation of the models are very rare.

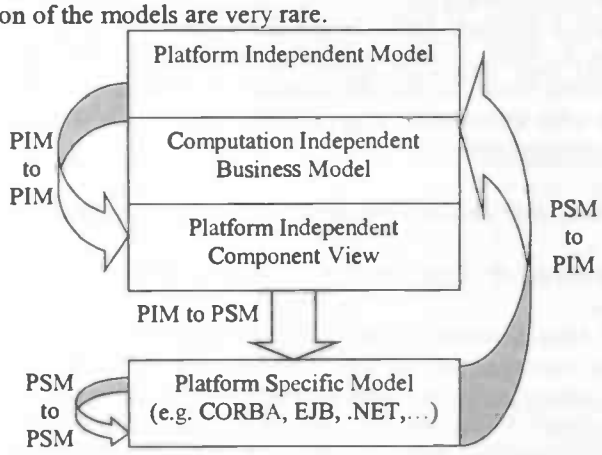


Figure 11: MDA process

5.4 Defining Interfaces

So far, we have described techniques to raise the level of abstraction, and to divide and conquer the software by modularisation. A next step in modularisation was to create standard modules, which could be developed independently, the so-called components. Now we consider the following issue. As components are intended to be developed independently, one should be able to combine them freely with other components. That requires that component implementations are unrelated to each other. The dilemma here is that components need to interact with each other, but may not depend on each other. The solution is a communication medium, called interfaces [4]. Ideally, the entire external behaviour of a component is defined in terms of interfaces [10].

Several definitions for interfaces are used. In general, an interface is said to be a contract that defines a set of services, identifies the parties involved in these services, and defines terms of use, rights and duties to these parties. When we apply this to software [10], we call an interface the boundary across two independent entities that meet or communicate with each other, which represents specific points of potential interactions between those entities. A point of interaction may be, for example, a method that can be called externally.

5.4.1 Types of interfaces

The interface that we defined so far encompasses in fact the definition of three types of interfaces [4]. To separate concerns, one could distinguish, in the first place, so-called *provides* and *requires* interfaces. The *provides* interface should only define which methods it makes available, whilst the *requires* interface should define which methods it needs. It is good to make this distinction, because this way the binding knowledge is taken out of the components, such that the binding can be made at product level [24]. A product is a *configuration* of a set of components. All *requires* interfaces of a component must be bound to precisely one *provides* interface; each *provides* interface can be bound to zero or more *requires* interfaces.

To separate concerns even further, and facilitate reuse, components should not contain configuration specific information [24]. But if all configuration specific code would be removed from the component, it would be not very usable. Therefore components should be parameterised over all configuration-specific information. Only, when a lot of parameters are needed, the component might get difficult to understand. Therefore, the *configuration* interface is defined. A configuration interface contains a set of

parameters that can be used to fine-tune the component in a configuration (for a particular context and set of requirements). With each point of choice for a different behaviour of the component, another configuration interface is associated.

5.4.2 Interface specifications

An interface is more than a list of its available services. There may be preconditions required, or error handling provided, or other behaviour that is not described in the syntactic interface. Therefore, documentation is necessary, which is described in the so-called interface specifications. Ideally, an interface specification should provide enough information to avoid unexpected interactions that can occur because of assumptions an entity makes about either the environment in which it is to be placed or the entities with which it interacts [10]. In fact, it should contain all information the user of an interface needs to know to interact with it, and nothing more. This information should include restrictions and side effects (how to use the element in combination with other elements). The architects should choose what information is permissible, appropriate for people to assume about the element, and unlikely to change [10].

Clements et. all. [10] describes a "standard organisation" for interface specifications, which has the following elements:

- *Identity*. This can be the name of the interface, but may include also a version number, or other distinguishing marks.
- *Resources provided*. Most common are methods. One should describe the syntax, semantics (such as value assignments, messages sent and any other humanly observable results), and usage restrictions (such as calling order, or access restrictions) of these resources.
- *Locally defined data types*. If data types different than the ones provided by the underlying programming languages are used. In that case, one should make clear how that data type is to be used (e.g. declaring variables of that data type, converting values of that data type to another, and so on).
- *Error handling*. Any error conditions that can be raised by the resources of the interface.
- *Variability provided*. If different configurations are possible, the configuration parameters should be given, and also a description of how they affect the semantics.
- *Quality attribute characteristics*. The architect needs to document what quality attribute characteristics (such as performance, or reliability) the interface makes known to the element's users. This information may be in the form of constraints on implementations of elements that will realize the interface.
- *Resources required*. Anything the element requires, such as class libraries, or other interfaces. One should describe syntax, semantics and usage restrictions of these resources.
- *Rationale and design issues*. Motivations behind the design.
- *Usage guide*. Extra semantic information (e.g. how individual interactions interrelate, or models/scenario's showing how to interact with the element).

5.4.3 Representation of interfaces

Many notations for interfaces are used. We show an example of interfaces modelled in UML, the unified modelling language [12], to demonstrate a way to depict the existence of interfaces graphically. UML clearly distinguishes the interface from the realised interface (the implementation). UML can be used to model the ordering of interactions, or scenario's and other kinds of behaviour too. In Figure 12 one can see the most compact way to represent an interface in a UML compact diagram. The "lollipop" that is attached to a box that represents a class, is the interface. The class is the implementation of the interface. The interface can be used by other classes.

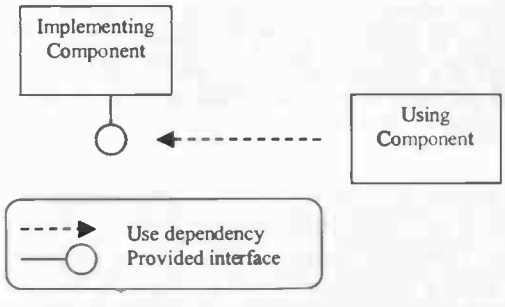


Figure 12: Lollipop (compact) notation for interfaces

Another notation, shown in Figure 13, represents interfaces as packages (denoted as boxes). In these packages, the resources provided are given. Arrows indicate the use dependencies between the (sub) interfaces. In this example no generalisation (a parent interface generalising children interfaces) is provided; in that case other arrows would be used to indicate the generalisation dependencies. The example given here, is a part of the Patientadminstration subsystem.

In short, the functionality the interface provides is as follows. In the attributes of the DataObject image-related data are stored. The DataSource represents physical data sources such as networks and pipes from which data can be "pulled", whereas the DataSink represents physical data sources such as networks and pipes from which data can be "pushed". Then, the DataSourceConnection represents the access (connection) for a data user to physical data sources, and the DataSinkConnection represents the access (connection) for a data producer to physical data sources. Finally, the DataDictionary contains descriptive information about the object types (which attributes they have), and attribute tags (definitions).

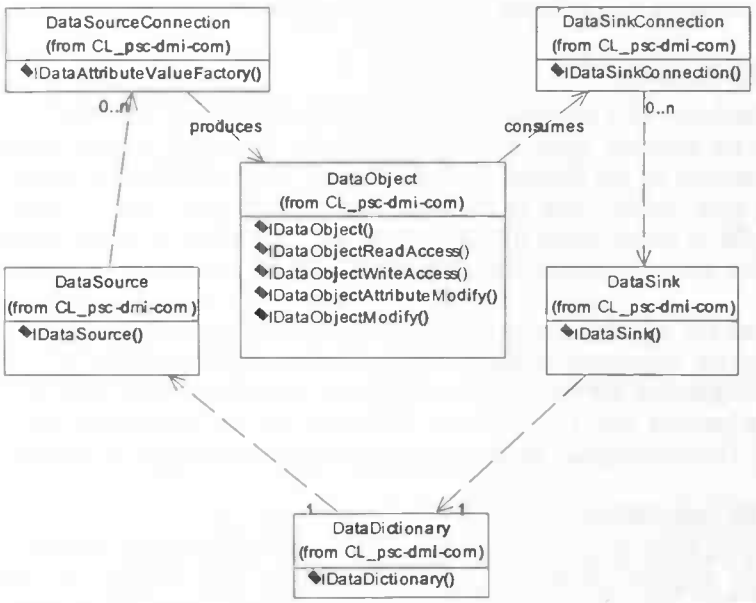


Figure 13: Producer/Consumer interface, represented in UML package diagrams.

For defining interfaces (syntactically), e.g. IDL: *interface definition language*, is used. IDL by itself is, similar to UML, inadequate for fully documenting interfaces, because it offers no language constructs for discussing the semantics of an interface. Therefore we still need natural languages.

5.5 Management of interface changes

5.5.1 Interface management

The management of interfaces is about controlling changes to interfaces. Changes may concern e.g. additions of new functionality or changes to component level access (such as the include scoping mechanism used at Philips). The major concern is the stability of the interfaces, because if an interface is not stable, the users of that interface have to adapt to the referenced interface each time it has changed again, leading to even more changes and consequently more effort and costs.

Measures to enable stable interfaces are:

- Have small components, thus small interfaces, with a limited set of, preferably generic, functionality [14].
- Require authorisation for interface changes. In a development plan should be specified at which particular point the documentation is brought up-to-date. Avoid revising documentation to reflect decisions that will not persist [10].
- Recognise component evolution [17], as to specify the intent and alternatives during design, so the decision to choose one of the alternatives can be made even at run-time (general or multi purpose programming).

Making allowances for component evolution is possible in two ways: by variation (describing variability points where the component can be configured), or by composition (diversity is created by adding/replacing components). Both ways are described further in the next paragraphs, 5.5.2 and 5.5.4.

5.5.2 Interface size

Small interfaces facilitate reuse, are flexible in composition and are easier to manage change. This fine factoring of interfaces such that they are small but usable, is in itself an interesting challenge. One way to do this, is by role-based modelling [14]. This means that the component has an interface for each role it exposes [14]. The concept of roles is based on the notion that components may have several different uses in a system. These different uses of a component in a system are dependent on the different roles a component can play in the various component collaborations in the system. When role-based interfaces are used, the users of a component can communicate with the component through these smaller role interfaces instead of through the full interface. Thus the communication between components becomes more specific. One of the advantages of role-based modelling is component changes are more localised, thus affecting only the users of the specific interfaces concerned.

5.5.3 Diversity by variation

To increase the flexibility of a component, the component should have so-called variability points, to enable the choice for different applications. These choices are in fact delayed design decisions [17]. The delay is determined by the binding time: the time at which the choice is made. This can be e.g. during compiling time, linking time or run-time. The most extreme form of variability is when all design decisions can be taken during run-time. Then, the architecture is totally dynamical. This offers enormous flexibility, but also requires a lot of expertise and time to implement the (possible) decisions!

Kinds of variations are e.g. the choice for the presence or absence of components (options) or the selection of a concrete component to implement an abstract one (instance out of several alternatives). Various ways to implement the variability points exist, depending on the kind of variation, and the binding time. Mechanisms that are used very commonly are e.g. inheritance and configuration (by using parameters). Unfortunately no unified notation to describe variability is currently available.

5.5.4 Diversity by composition

Composition is in fact the reverse of variation, which can be illustrated as follows [26]. Whereas variation is having product specific plug-ins for a reusable component framework, composition is having reusable components composed in a product specific way. Thus, the framework represents the products' commonalities, whereas the components represent the products' diversity.

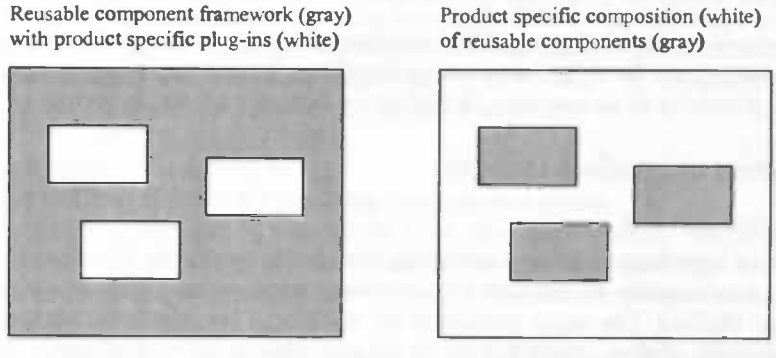


Figure 14: Variation (left) and composition (right) are complementary approaches.

The idea is that the components don't change - thus its interfaces are stable -, are not dependent on each other ("the first can be used without the second, and vice versa"), and only need to be arranged in some specific way. The diversity is in the choice of the components, which can even be done at runtime. Note that diversity by parameterisation (thus, variation) is not possible at runtime, but only earlier, e.g. during compile-time.

Of course both approaches to implement diversity can be combined. One should find the right balance in the combination. For large software organisations [26] recommends a bottom up approach (from software components), with a lightweight (top-down) architecture to prevent architectural mismatches. The usage of composition is particular useful for companies that cover a wide product scope (in which

the components can be reused). The usage of variability within the components then adds an even more flexible usage of those components; such that they may behave different in various environments.

5.5.5 Compatibility of products

The ultimate independent deployment of components would be realised if the components were upwards and downwards compatible, and reusable and portable. Van Ommering [23] illustrates these concepts as follows (see Figure 15). The names "Client" and "Server" stand for different pieces of software that can call each other's functions; the client being user of the functions provided by the server.

- A client is upward compatible with a server if it can be combined with newer versions of the server, and downward compatible if it can be combined with older versions of the server;
- A server is reusable if clients that were not taken into account when the server was created can use it.
- A client is portable if it can use servers that were not taken into account when the client was created.

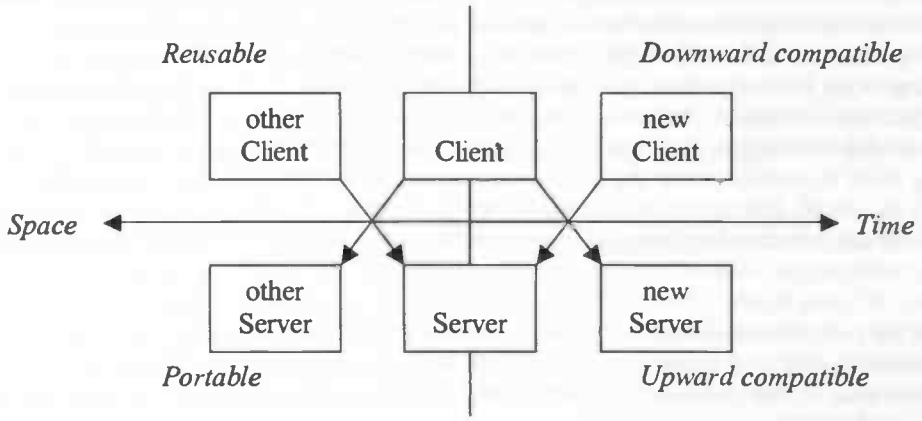


Figure 15: Ways of independent deployment

Components that have all these characteristics are extremely stable, thus facilitating all software components to be developed separately both in time and in space, having their own evolution paths, and at the end they being combined to one working system. In the next paragraph we have studied the concerns that can be handled by the right organisation of interfaces. Note that there exist other solutions too, but we have not discussed them here.

Compatibility in time (upwards)

All users (clients) of a component will be upwards compatible with it, if that component is so-called *backwards* compatible, i.e. all clients that can run at the current component should be supported at the new component as well. Thus, a backward compatible extension is achieved if a new interface still supports all functionality of the previous interface. To make components backwards compatible, one can use versioned interfaces or immutable interfaces.

The concept of versioned interfaces is to allow interfaces adding functions, but not removing any. One should guard however, that the extensions are semantically compatible. To manage changes to the interface, version number should be attached to the interface. These can be used by clients to specify which version of the interface they require (at least). This may lead to complex compatibility relations among various versions of interfaces.

The principle of immutable interfaces is they don't ever change after they are published. This only is possible when the interfaces are kept small! Therefore, a component should be allowed to provide multiple interfaces. If changes to an interface are needed, then a new interface will be added, while the old interface still is supported. Microsoft COM uses this approach.

Compatibility in space (reusability)

For a component to be reusable a solution is to use configuration interfaces or diversity interfaces. Both are ways to parameterise the component, such that context specific decisions can be postponed. A configuration interface enables this by providing *set* and *get* functions to retrieve certain properties of the software, which influence the behaviour of the software. Disadvantage is the system has to be

initialised during run-time. A diversity interface retrieves the values of parameters from its environment by using "#define" statements, or a registry that contains key-value pairs with settings for a variety of software. Thus the software can be fine-tuned for a specific configuration, before it gets compiled.

5.6 Summary and evaluation of techniques

After the investigation of the literature related to interfaces, this section summarises the trends identified, and tells how the developments have contributed to this thesis. In the last subsection, the situation at MR regarding those developments is described.

5.6.1 Evolution of software engineering discipline – towards component organisation

With the expected evolution of software engineering towards a discipline that is based on components that can be exchanged, companies should prepare their organisation in such a way that they can adapt their software development to the changing environment, so they remain competitive in their branch. This means their development should be organised such that they can use components, and preferably also develop them (as described in [37]). Furthermore they should weight their competence at various areas against what is available as third party software. When the component software market starts growing, this may accelerate. Therefore companies should think of a good balance between make-all, which is currently the biggest part, and buy-all. It is important to know one's competitive edge, because it's getting more important (when the basic software is not an issue anymore, because the competitor has exactly the same). But notice, even if the company develops all software by itself, then components may be useful too within a (large) organisation, because they are easier to develop independently and to reuse.

To support the evolution of software engineering, it is required to:
- Organise subsystems as components: black box, with clear unambiguous interfaces
- Be conscious of one's competitive edge: focus will be stronger as soon as basic software gets really standardised

The trend of component engineering gives a motivation for the importance we see in the definition of clear unambiguous interfaces. It supports our effort to investigate how these interfaces should be organised.

5.6.2 Independently developing subsystems

A base for independent development is of course the usage of components, with corresponding interfaces, because they clearly define everything one needs to know to develop a component without needing information from the other components. But then, the stability of the component interfaces is important, because changes to an interface might lead to side effects. In the worst case a component that was using the subject interface, won't be able to use it anymore.

To have stable interfaces, it is required to:
- Take account of future changes, by providing configuration points.
- Choose the right partitioning ('composition') for the functionalities among components, and implicitly have the right 'size' of functionality for the corresponding interfaces. Also, the interfaces may be organised further into sub-interfaces, and a component may have multiple interfaces.
- Make a very clear distinction between specifications and implementation (while the specification is the same, the implementation may be varying).

With respect to this research, we have taken account of all these issues, when we studied the possibilities of interface change management.

5.6.3 Developing at a higher level of abstraction

Higher level of abstraction development is a big step from now. It may indicate a completely different development organisation in future.

To develop at higher level of abstraction, it is required to:
- Shift software design from solution domain towards problem domain
- Use frameworks, to generate (and verify) 3GL code from models automatically

The modelling techniques themselves are outside the scope of this paper. However, the advantage of modelling is they make the *"what"* really clear, and hide the *"how"*. That's exactly what we want to achieve with interfaces.

5.6.4 MR situation regarding the trends discussed

At MR, the subsystems (or building blocks within the subsystems) can't be called components – as defined by Szyperski [37] – yet. The development is white box: the code of the building blocks used is visible to the users. The building blocks are not independent of each other.

Because development is done concurrently in different development streams, and cooperation with foreign countries grows, the need for independent subsystem parts rises. To get there, interface management could play an important role.

Currently, interfaces only have a supporting, not a prominent role, and are therefore not fully described. At MR the Interface Specifications are sometimes defined as part of the Requirement Specifications. The differences between the Interface Specifications and Requirement Specifications are not made very clear. But interfaces that are used externally (i.e. by other building blocks than the implementing one) have got a prominent location in the code archive: at subsystem level, in a special directory. Also, the use relations of the interfaces are checked. This means a building block may only use from a defined range of subsystems.

Future changes are estimated roughly during the beginning of a project: software architects decide which building blocks are accessible for changes and which are not. Because development is done in different projects, the code is reflected in different development streams, which must be merged into each other. Therefore communication about what is changed in which stream is very important, because merging is much more difficult when an object has been changed in both streams. Though, there is no formal way of monitoring these changes and the consequences (impact) to the merges.

MR Software is developed in 3rd level languages. The development is made more generic by using the .NET environment, which supports different programming languages. Models (in UML) are used in some parts of the documentation, to describe what is implemented, and how the implementation should be used. However, these models are only descriptive, and not a base for programming. The UML models used to describe MR interfaces were intended to be mapped 1-1 to the implementation, as a control mechanism on the consistency between the UML models described and the code. But they were not supposed to generate code from. Thus, no visible intention exists to use 4th level languages (techniques) in near future.

6. Case studies at Philips MR

Here we discuss which interfaces we studied, which stakeholders were concerned, and how we prepared the interviews. Finally the cases are described. The results are given in the chapter 7.

6.1 Identifying goals, objects and stakeholders concerned

6.1.1 Research deliverables

According to the assignment at Philips MR we had to define the interface specifications and the way in which their description and usage could be facilitated.

As the current interface specifications at Philips MR were described in documents, which were based on an interface specification template, we aimed at first at improving that interface specification template such that it would prescribe those items that we identified as useful. With the knowledge we got from literature, we pre-defined the following requirements to this interface specification template:

- According to chapter 5.6, the documentation of 'what' and 'how' should be separated, the interface specifications are supposed to cover only 'what' is implemented (and the design specifications the 'why' and 'how' of the implementation).
- If certain variability is provided, this should be described in the Interface Specification.
- If measures have been taken that guarantee a certain quality level, these should be described in the Interface Specification.
- If applicable, possibilities with respect to automatic code generation will be considered.

To describe how interface specifications could be better embedded in the development process, we aimed at delivering some guidelines. The goal of the guidelines should be to realize better up-to-date interface specifications.

6.1.2 Selection of convenient interfaces

As a validation for the deliverables, we studied two cases at Philips MR. The cases would involve a study of an MR system's interface and documentation, and interviews with users of that interface. To determine which interfaces were convenient, we used the following requirements:

- interfaces are used by other building blocks (so we could interview the users)
- responsible people are available

The last requirement was the first that mattered, just because the research would be done during the summer, when many people have a holiday. The owner of the Platform subsystem was available. We decided then to study the configuration part of platform, because the interfaces of configuration are used by building blocks of almost all other subsystems, thus there would be many developers that had used the configuration interface.

Later, when it appeared that the configuration interface was not very complex (e.g. no concurrency or quality issues involved), we decided to study another case, which would be more complex. The owner of another subsystem suggested, for this reason, to study the "Command Dispatcher" interface. This happened to be located in the Platform subsystem too. The number of users turned out to be rather small, but as they were all available, it would not be a big deal, and we would even cover the whole group of command dispatcher users.

6.2 Interviews

6.2.1 Identification of stakeholders

With respect to interfaces, the following stakeholders can be identified at MR – and their use of interfaces, and concerns:

- Designers and architects: the responsible persons for negotiating and making trade-offs among competing requirements. They should be guarding the systems' integrity and manage the interface dependencies. They want to know all key decisions that have been made (why is which interface located where).
- Developers: the persons to implement and make use of the interfaces. They e.g. need to know which functionalities need to be implemented.

- Testers and integrators: the persons to check the quality of the software. They e.g. need to guard the performance and reliability of the system.
- Software Configuration Managers: the persons to support the development and build of the system. They need to know how all components of the system are related, and especially they need an overview of all variability provided.
- Project Managers: planning purposes: forming development teams, allocation of project resources, tracking progress, costs, etc.

6.2.2 Selection of stakeholders

We aimed to cover as much different stakeholders as possible, to identify their possibly different needs. Regarding the contents of the case interfaces, developers may be expected to have the most detailed knowledge. Therefore we first interviewed the using developers and designers of both interfaces.

As we expected management issues to be less domain-specific, we decided also to set up interviews in a wider context – concerning the whole software system. Therefore we asked designers and architects to answer our questions, as they have the best overview of the whole system. The project managers were represented in this group (as there were designers that were also project manager). Unfortunately there was no more time to interview the configuration managers and integrators, but we did draw a tester to our interviews.

6.2.3 Definition of checklist and questionnaires

The questions for the case interviews were mainly aimed at exploring the (interface) users' needs with respect to the contents of the interface specifications. Therefore we used Clements' organisation of interface specification model [10] as a model for a checklist for interface specifications. We wanted this checklist to provide as much contents as possible for the interfaces, with the purpose to select the proper ones needed for the resulting template for interface specifications.

The items concerning quality attributes have been selected in discussion with two senior designers, who identified relevant quality attributes for Philips MR (so we could explain them to the users). This reduced our initial list of about twelve quality items[5] to three: performance, scalability and security. The realisation of two other quality attributes was mentioned at another place in the list (configurability is described at the item Variability provided, and information about correctness can be found in the item Error handling). For other quality attributes, such as maintainability, usability and availability, we could not find concrete measures to present to the users. The complete checklist is given in Appendix B. We also added some questions about improving the documentation process. Those questions were open, because we didn't know beforehand what the issues would be. The complete questionnaire is given in Appendix A.

The questionnaire for the management interviews contained open questions that were primarily guidelines to interviews: they concerned the organisation and management of interfaces. We offered the interviewees different imaginary situations to which they could respond, and add their own ideas. Again the complete questionnaire is given in Appendix A.

6.2.4 Interviewing the stakeholders

The organisation of the interviews is described already in paragraph 3. The interviews for case 1 have been done in August 2003. The interviews for case 2 have been done in September 2003. The results can be found in the next chapter.

Whereas in the previous interviews we merely interviewed people in the role of developer (although they might be designers too), we expected to get more ideas about the organisation and management of the interfaces from designers and architects. The results are presented as a survey, in the next chapter. The interviews with the senior designers and software architects have been done in October 2003.

[5] This list of quality attributes is not trivial; different authors use different definitions. Van Gurp [14] defines the following quality attributes: complexity, configuration, correctness, flexibility, maintainability, modifiability, performance, reliability, reusability, safety, scalability and usability.

6.3 Interviews and cases at Philips MR

Here, we have described the functionality and context of the interfaces of the cases that we studied at Philips MR. We started with a description of the current interface specifications. After each description, we have given an evaluation of the subject described.

6.3.1 Current Interface Specifications

Directions to describe Interface Specifications are currently given in the "Standard: Interface Description" [54] and "Standard: OO Interface Specification" [57]. The directions given in these standards are reflected in templates to create an Interface Specification document. Thus, in the templates is pointed where what information should be described. The generic template for Interface Specification is given in Appendix C. The other template is left outside consideration, because it is only applicable for interfaces of code that has been developed with object-oriented methodology.

Both templates and standards have not been updated lately. The Interface Specification template provides not much information. Beside some general information such as the scope of the document and the intended readers, it contains the following items (without much explanation, as you can see in the appendix):

- Description
- User interface
- Declaration (software interface)
- Input (software interface)
- Output (software interface)
- Message layout (message interface)
- Message elements (message interface)
- Threading issues

The introduction chapter of the "Standard: Interface Description" tells that the Interface Description (similar to what we call Interface Specifications) is "in principle part of the Functional Specification", and therefore "in order not to describe common sections twice, this document refers to the standard for the Functional Specifications in many cases". The Standard even states that "the Interface Description is not such that everything a user needs to know to use the described functionality is described in the Interface Description", because it would be confusing then to know what information should be in a Functional Specification, and what should be described in an Interface Description.

So far, the required context of the Interface Description is not made explicit. In the remaining chapters of the Standard Interface Description, the minimal requirements to the Interface Description are described as follows:

- For each function with a user interface:
  - Short description
  - User interface
- For each function with a software interface:
  - Short description
  - Inputs
  - Outputs
- For each function with a message interface (i.e. between processes)
  - Short description
  - Message layout
  - Message element
- For a function with another type of interface
  - Type of interface
  - Description

The Object Oriented standard is quite analogous to the general standard; the main difference is that classes instead of functions are described.

Evaluation of interface issues described

It is strange that the standards for Interface Specifications don't prescribe having all information that an interface user needs, described in the Interface Specification. If the standards clearly define what information is needed in which document, there would be no confusion. Therefore, we suggest having

all requirement issues described in the Requirement Specification (RS), and all information one needs to use the interface in the Interface Specification (IS). This way the IS is the only document that is to be delivered (thus visible to external users of the interface), and the RS can be used only internally. Corresponding to the new definition of the Interface Specification standards, the interface specification template should contain all issues that need to be described in an interface specification.

6.3.2 Case 1: Configuration Interface

The configuration part of Platform consists of three layers of software (see *Figure 16*). At the first layer is DAI, the Data Access Interface, which is supplied by an external supplier. It is a COM interface that provides all basic methods to access any repository. At the second layer is AWAPCO, standing for Application Software - Application Configuration. AWAPCO is made by the Platform Component and Services (PCS) Group and is mainly used internally in this group. It is meant to hide DAI, and solve COM related problems. This third layer, AWASW, is a C interface, which is used by almost everybody within MR software who wants to retrieve values of attributes from the MR configuration repository.

The configuration repository contains configuration attributes, e.g. hospital names, colour schemes and coil types. DAI and AWAPCO access these attributes by basic methods, e.g. get_root(), get_child() and get_attribute(). AWASW contains more compound methods to retrieve attributes from the configuration repository (using the basic functionality provided in lower layers), e.g. get_hospital_name().

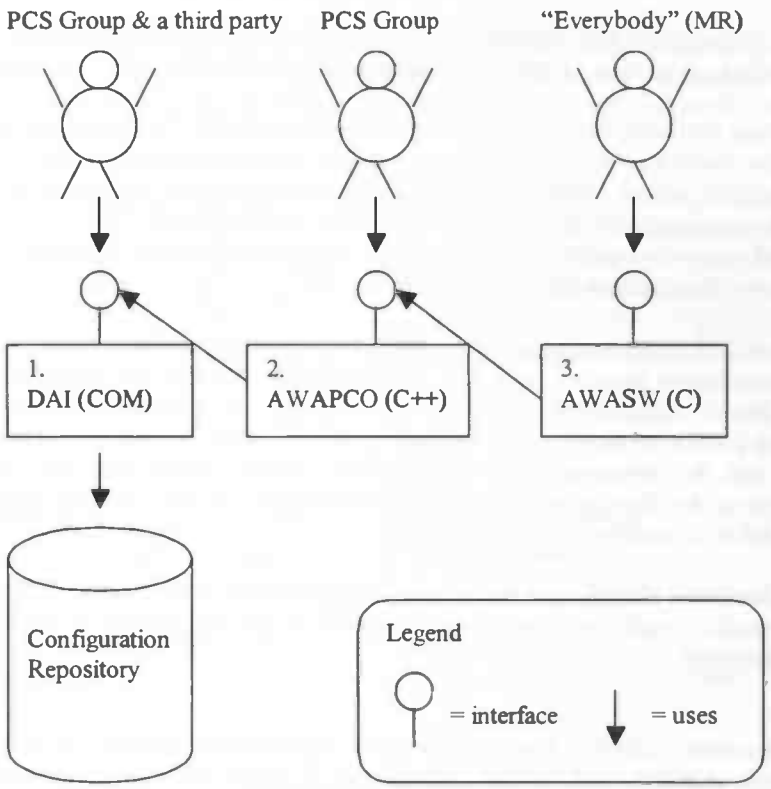


Figure 16: Three layers of configuration

Current Configuration Interface Specifications

Because the AWASW layer provides the interface to the 'external' users (that says: other users than the PCS group, but still within the MR department), we have focused our study on this layer. We have identified first what has been described in its interface and evaluated what we found. Those findings can be seen as an introduction towards the interviews, in which we want the stakeholders to evaluate the interface specifications of AWASW. The final results can be found in the next chapter.

The AWASW Interface Specification is embedded in the document that describes the MR Configuration Model [53]. Because little interface information can be found here, we have also investigated which interface information is described in other sources, i.e. code, header, other software

development documents. Below, we have first objectively described what we found; in the following sections we have evaluated the results, and given conclusions that will be used to define the interface specification template and the interface specification procedure.

The code (awasw_cfg_wnt.c) contains,
- Local data definitions (initialisations of structs)

And for each function:
- Function name
- Arguments names and types
- Return type
- Package and scope name
- Semantic description
- Precondition if needed
- Postcondition if needed
- Calling sequence if needed
- Error handling if needed
- Notes if needed

The header (awasglo.h) contains:
- Data types: structs and enumerations
- Function names, with their argument names and types, and return type

As said before, the interface specification is merely an addition to the MR Configuration Model. This document describes properties of all configuration data, needed for the MR scanner software. It includes:
- Class diagrams that show the relations between the objects in the Configuration repository
- Configuration Validation rules (indicating e.g. which combinations are allowed)
- Short description of all attributes that belong to Configuration objects (e.g. tag number that indicates the location in the database, possible values, and meanings)
- Summary of methods: function name, argument types, tag number of attribute (if applicable) and sometimes possible implementation directives

Evaluation of interface issues described

There are inconsistencies between code and documentation, like obsolete methods. As the "IS: MR Configuration Model" [53] merely describes the structure and contents of the MR Configuration Model, and only gives a summary of the methods provided by the AWASW interface, it would make more sense to call the document "MR Configuration Model" rather than "*IS:* MR Configuration Model". It might not be clear to the users of the configuration interface, that the document we discuss is indeed intended to be an Interface Specification.

Most interface issues are described in the AWASW Requirement Specification (RS) and (comments in the) code. No quality requirements were made explicit in the RS, so also no realisations of quality attributes are described.

Conclusions:
- The configuration's Interface Specification gives very little information. To provide a user of the configuration interface with all the information he needs to use this interface, the Interface Specification should be described in more detail.
- As there might be confusion about the location of the document, it would be good to have the Interface Specification described in a separate document (and the MR Configuration Model in another).

6.3.3 Case 2: the Command Dispatcher

The Command Dispatcher is a component used to issue commands to unknown command performers. A command is a global function, which may be accompanied with a number of arguments. The function may be implemented by one or more command performers. The coupling between command names and command performers is kept in the Registry (containing e.g. command name, name of implementing performer and internal method name).

The Command Dispatcher will be incorporated into a dynamic link library (dll), so each issuer of a command has its own Command Dispatcher; each client when it accesses the functionality of the command dispatcher loads the dll into its process space.

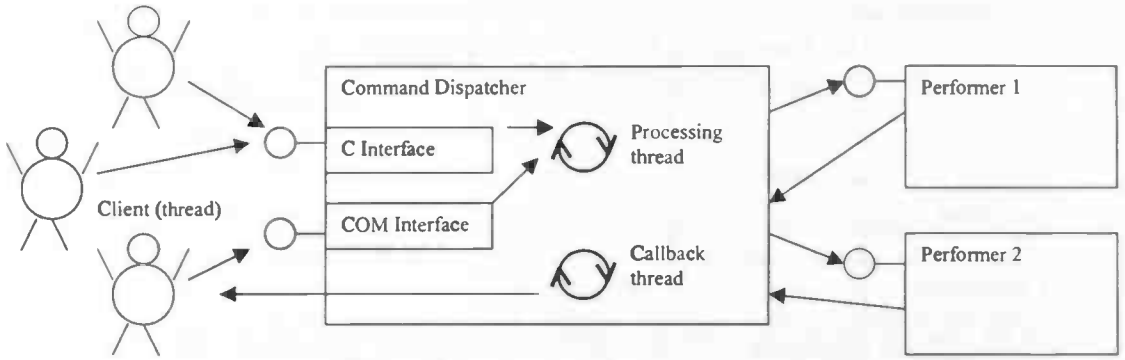


Figure 17: Overview of the Command Dispatcher (usage)

To communicate with the clients, the Command Dispatcher provides two interfaces: a COM interface and a 'C' style interface. To hide COM for the clients, the Command Dispatcher creates a thread internally that processes the commands. This processing thread is controlled via a message loop, that enables synchronisation between the user thread and the processing thread.

To communicate with the command performers the Command Dispatcher requires the performers to implement their own COM interface for each command they perform. The callback thread of the Command Dispatcher receives notifications from the command performers and sends them to the client applications.

The command performer may decide within the implementation of the command if it must be queued, or performed immediately. The client may choose to issue its commands synchronously or asynchronously. In the first case, the invocation of the command will return when the command has been processed. In the other case, the invocation of the command will return directly (which means the client remains responsive). The result of the invocation will be returned later. Figure 18 and Figure 19 show how these choices for clients and performers appear in different use cases.

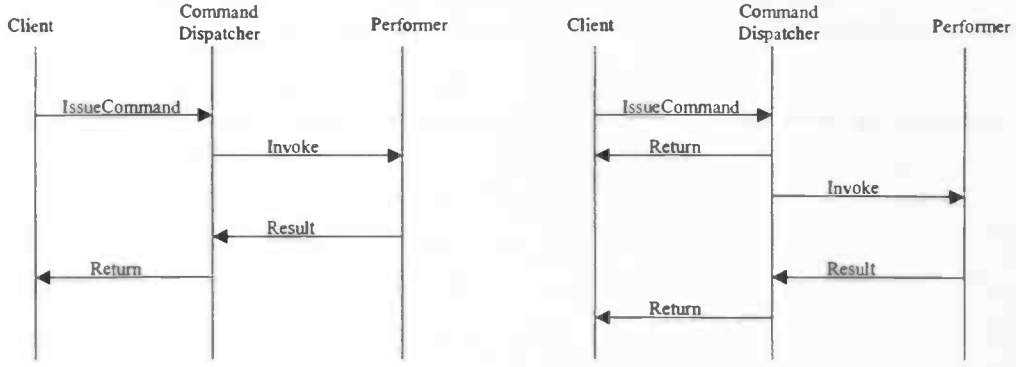


Figure 18: Use case for issuing a synchronous command, not queued (left) and use case for issuing an asynchronous command, not queued (right).

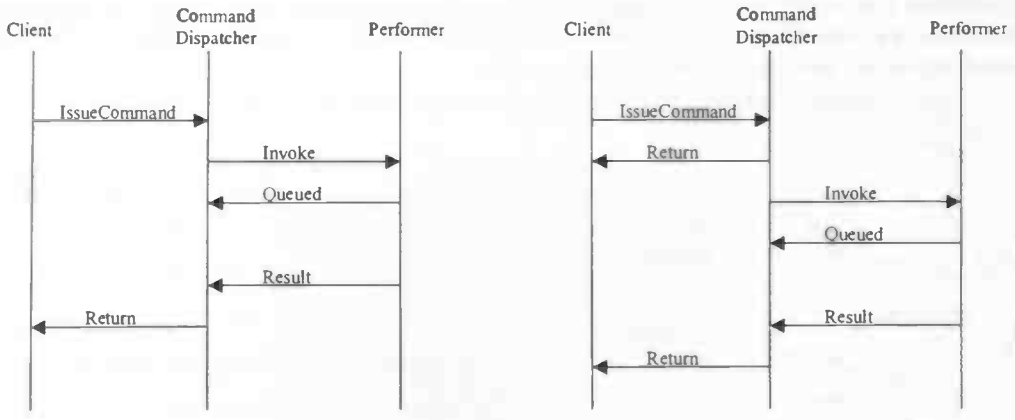


Figure 19: Use case for issuing a queable command, synchronous (left) and use case for issuing a queable command, asynchronous (right).

Current Command Dispatcher Interfaces

The Command Dispatcher's Interface Specification is a chapter of the Command Dispatcher's Requirement Specification document [48]. This chapter contains:

- Locally defined data types;
- Description of all functions for both C style and COM interface; for each function: argument names and types, return type and semantic description;
- Guidelines to the command performer interfaces (based on IDispatch interface), and example of a command performer interface.

Evaluation of interface issues described

The interface specification contains most of the suggested items to be described. Data types and methods for the clients are given. Threading seems not very complete; the interface provides a lot of combining possibilities, but not all exceptions are clear. COM knowledge is expected (for providers), so no COM issues are described. Quality issues are not described. Again, the interviews with the interface users will show if this information is really 'missing'.

7. Results of interviews with Philips employees

Here are results of the interviews done at Philips MR. First we describe the results of the interviews with users of the interfaces that we studied: the configuration interface and the command dispatcher interface. Then we describe the results of the interviews with the designers and architects. In paragraph 7.3 we present an analysis of all the results. Finally, in paragraph 7.4 we present the deliverables and recommendations for Philips MR.

7.1 Interviews with interface users

We have interviewed six users of the configuration interface, which were developers and designers that made use of the configuration interface. They had used at least two methods of the configuration interface: the init() to use the package and another one. The group of interviewees represented in large extent the developers that use the configuration interface, because of its constitution: the interviewees covered all the using subsystems, and the designers represented – as supervisors of the developers – an even bigger group of developers. However, statistically this group is not large enough to draw exact conclusions from the results we got. Nevertheless a lot of valuable information is gathered from the interviews, which is summarised in this section.

Similar to the interviews with the Configuration Interface users, we have interviewed the developers that used the Command Dispatcher Interface, in fact, they were *all* the users of the command dispatcher interface. Four of the five interviewees had used the Command Dispatcher both as a client and as a performer. As a provider they were expected to know COM (Microsoft's Component Object Model). The other interviewee was only a client, and used the C-interface.

7.1.1 Contents of the Interface Specifications

In *Figure 20* and *Figure 21*, the answers given to questions A3-A5 are shown for the configuration interface, respectively the command dispatcher interface. In the figures we have shown a horizontal bar for each of the questions:

A3: Do you think this item is sufficiently described?
The bar indicates how many interviewees thought that for this item enough information was provided (in the documentation) to use the interface.

A4: Do you think this item is necessary for the use of the configuration interface?
This bar indicates how many interviewees thought the item was needed to understand and use the configuration interface.

A5: Do you think this item is not needed at all?
This bar indicates how many interviewees thought the item would never be necessary to be described (for any interface).

Configuration Interface: contents of the Interface Specifications

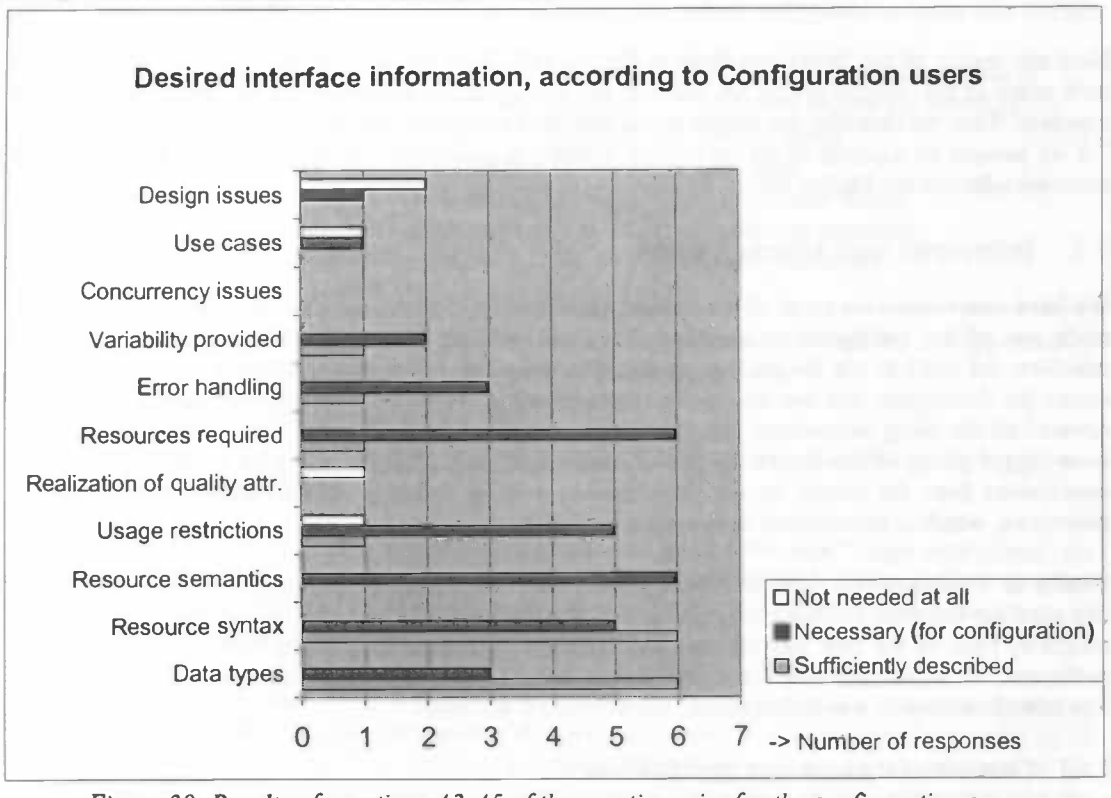


Figure 20: Results of questions A3-A5 of the questionnaire for the configuration case.

In Table 2 the previous results are presented in such a way that we can easily see which items are described sufficiently, but not needed, and vice versa. We have partitioned this table quite rigorous, thereby assuming that the opinion of the majority of the interviewees yields. We have marked the area of items that are not described sufficiently, but do need to be described (in grey), because that's the area that indicates which information the developers are missing.

	Needed for configuration	Not needed for configuration
Described sufficiently	Data types Resource Syntax	
Not described (sufficiently)	Resource semantics Resources required Usage restrictions Error handling	Design issues Concurrency issues Quality attributes Variability provided Use cases

Table 2: Areas of need and fulfilment (for items described of the Configuration interface)

As one can see, according to the interviewees there are no items described that are not needed to use the configuration interface, which is good. But four items, the resource semantics, resources required, usage restrictions and error handing, are needed to use the configuration interface, but are not described sufficiently yet.

Command Dispatcher Interface: contents of the Interface Specifications

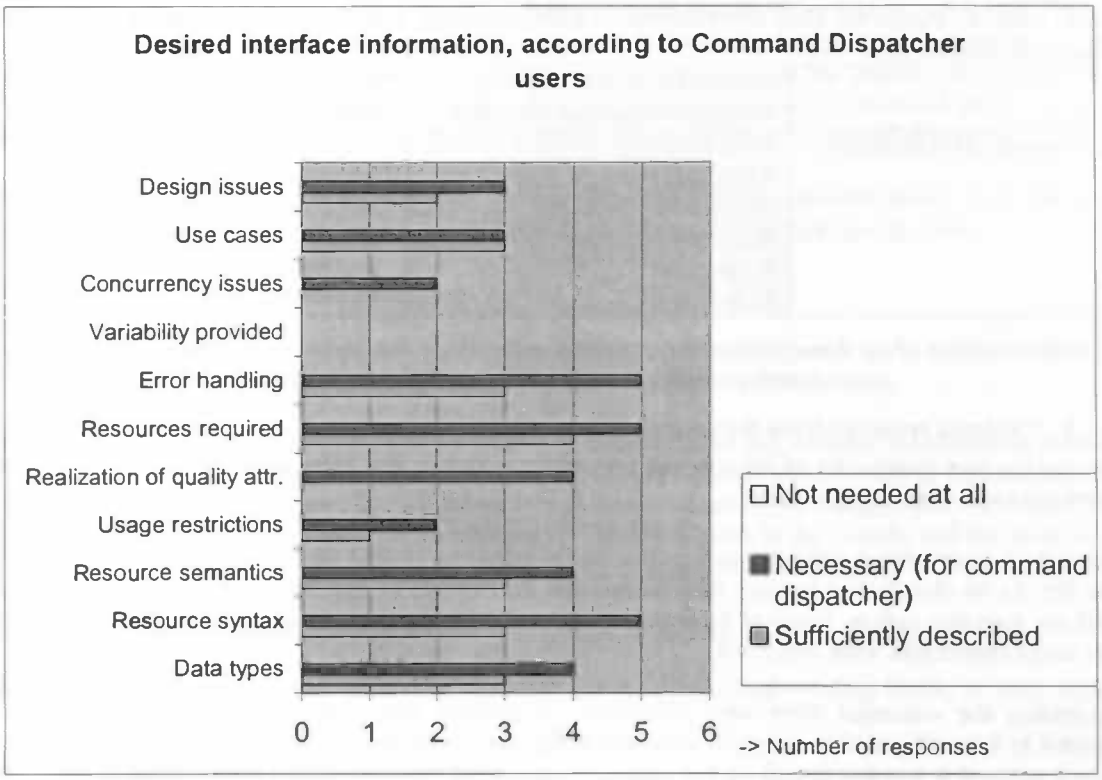


Figure 21: Results of questions A3-A5 of the questionnaire for the command dispatcher case.

Again, we also created a table to show the relation between described and needed items, for the command dispatcher interface.

	Needed for configuration	Not needed for configuration
Described sufficiently	Error handling Resources required Resource semantics Resource syntax Data types Use cases	
Not described (sufficiently)	Quality attributes Design issues	Usage restrictions Concurrency issues Variability provided

Table 3: Areas of need and fulfilment (for items described of the Configuration interface)

From this table, we can conclude that most items that the interviewees need are described presently. Only some quality issues and design issues are not clear enough.

Finally, we have merged the results of both the cases, by drawing the limit at a response of minimal 6 of the total of 11 persons. Thus we retrieved the following table, which only contains items that are not described sufficiently:

	Needed for configuration or command dispatcher interface	Not needed for configuration or command dispatcher interface
Not described (sufficiently)	Error handling Resources required Usage restrictions Resource semantics Resource syntax Data types	Design issues Use cases Concurrency issues Variability provided Quality attributes

Table 4: Areas of need and fulfilment (according to the using developers of both the configuration interface and the command dispatcher interface).

7.1.2 Contents remarks from the interviewees

During the interviews a lot of information popped up, which is also relevant to discuss in this report. Here are some interesting remarks.

Almost all interviewees were somewhat (positively) surprised to find the item "usage restrictions" on the list. As we already had learned, they thought this item should be described, but it appeared they had had not seen this before. Some of these interviewees remarked that it would safe them a lot of time if the usage restrictions were described.

Regarding the semantics there were differences of opinion. Most interviewees answered that they wanted to have all possible semantic information that was mentioned in the checklist [Appendix B, of which some of it was not described, but only a few remarked that they missed information. Users of the command dispatcher, for example, missed information about the effects of the parameters (in the variant record provided).

The users of the configuration interface agreed with each other on the redundancy to describe quality attributes; they thought the configuration interface was too simple to have any quality issues addressed. Regarding security, only the restrictions of who were allowed to use the set-functions were relevant. Most of the interviewees thought security should be handled at a higher level. The users of the command dispatcher missed an indication of the performance provided, and an indication of the usage purpose for which that performance was meant to yield.

The users of the command dispatcher thought there were too many possible use cases. This had as a consequence that not all of them were described. The same yielded for the (with use cases) related concurrency issues. As COM was presumed to be known by the users, no COM issues were described. Some users thought a user manual with cautions for COM dangers should be given. The command dispatcher users that wanted to know the design issues, were the command dispatcher users that were also providing methods.

7.1.3 Documentation process suggestions from the interviewees

The users of the configuration interface thought the documentation was not easy to find. The specification documents were indeed scattered over the archive. As a solution, they all came up with the idea that the documentation of the code should be closer together.

A first step of getting the documentation and code closer together was made a few years ago by combining the code archive and documentation archive together in a single structure. Together with this migration the building block structure was introduced, which resulted in a hierarchical partitioning of the code and the documentation. Some "old" documents are still not upgraded towards a higher level document, and are therefore still hard to find in the archive.

With regard to consistency of the documentation with the code, some interviewees (as mentioned before) suggested to use documentation generation from code. One interviewee warned not to use trendy names in interfaces, because they would get outdated and confusing. Another interviewee would like to know which methods in an interface were "leading" when there were multiple versions of the same method in the code. Finally, some interviewees remarked to have the contents of the interface specifications checked with the code on a predefined moment.

Almost all interviewees thought that interface specifications should be evaluated after some time, because they said they would change during development. They also advocated more iterative design and evaluation of the requirements with the design, and the design with the interface specifications. Furthermore, three of them thought that the interface specifications should be reviewed wider, i.e. not only by the building block owner, but at least by a senior designer or even a software architect.

One interviewee suggested having a list of good practices of interface specifications, such that the developers had examples to refer to when specifying or reviewing an interface specification.

7.2 Project Management interviews

We have interviewed (senior) designers, a software architect, a test coordinator and a system architect, six persons in total, in order to retrieve Philips MR related management information.

We did these interviews to identify how the interviewees thought the MR software development should be organised, such that changes to the software system could be done easily (without affecting much other software parts). Therefore, we identified two different situations of change, that we wanted to discuss with the interviewees: one was the addition of a building block to the system, and the other was a change of a building block that would affect users of the building block's interface.

Situation 1: A building block (or even subsystem) is added to the MR software system.

We first asked the interviewees what they wanted to know of the added building block, so they could manage their software projects properly.

7.2.1 Information needed

The designers and architects wanted to know the Requirement Specifications and Interface Specifications (they especially mentioned concurrency information, and quality aspects), a description of the new building block, a motivation for adding the building block to the system and an overview of the external dependencies (which interfaces are required?). If they would be involved in the development of the new building block, things they wanted to check on were:

- Can the interface be separated in a generic and specific part? Let the generic part be as big as possible. Describe the generic and specific part separately in the Interface Specifications;
- Does the interface not contain too much functionality?
- The Interface Specifications should not contain any implementation details;
- The names of the methods provided should be unambiguous, not trendy, and be consistent with each other (i.e. symmetric, intuitive, consequent).

7.2.2 What is required in an interface specification

Both designers and software architect thought that interface information in general should be described in the Interface Specifications. Only quality aspects, such as reliability, security and performance, were expected to be described in the Requirement Specifications (according to two interviewees)

Regarding configuration possibilities, one liked to have an overview of what configuration combinations are allowed. All configuration information is known at a central place now. A suggestion was to have a generic interface with sub-interfaces for different configuration options. For the description of quality aspects, several designers suggest to provide acceptance tests. Finally, to overview all interfaces' functionalities, the interviewees promoted a uniform representation of the interfaces.

Situation 2: A building block is changed, not only internally, but also externally (interfaces)

7.2.3 Managing changes to interfaces

The designers and software architect said that changes to interfaces could be managed only if the system was first made manageable, i.e. the interfaces should contain the right functionality (not too much header files), and the functionality of the interfaces should be described clearly. Regarding the right 'partitioning' of functionality to interfaces, two designers suggested to further expand the building

block structure that had not been completed totally. Another designer added that interfaces should be as small as possible, and defined such that they wouldn't need other interfaces.

Two designers and the test coordinator suggested having a mechanism that automatically generates information to monitor e.g. the interface dependencies, the number of interface changes, and the documents that have been updated recently - which is registered in the so-called "SMAL" (global project overview) at Philips MR.

Two designers and the architect mentioned that the first phase of the development process, when a new project was started, should be used to analyse the interface dependencies, and investigate the impact (or consequences) of the expected changes on the interfaces. Then the senior designers and architects should make agreements, especially at subsystem level, about which interfaces may be changed, syntactically or in behaviour. This can be seen as an extension to the policy to identify at the start of a project which building blocks may be changed.

Two designers thought "the polluter should pay", meaning that one that requests changes to an interface, should also be responsible for announcing the changes to the clients of the interface, or even negotiate the suggested changes with them. Though, one person should always be responsible for the consequences of the changes. Most interviewees thought this was the building block owner; one thought this was the segment leader.

One designer thought that authorisation for interface changes was necessary, but it wasn't enough to guarantee interface stability. He believed the only thing that could make interfaces stable was a good, generic, design. The senior designers should control the design decisions to make functionality generic or not (as over-generalisation is not desirable either).

In general, the designers thought interface changes should be controlled more (e.g. also authorisation by an architect), and the policy should be not to change interfaces, unless really necessary.

The system architect emphasised that the system's "components" should be black box, "oriented at external usage", such that input-output control would be facilitated. The system architect and one of the designers suggested that interfaces should also be described at architectural level, i.e. less detailed than the Interface Specifications, but with more focus on dependencies at a higher level. Thus, an overview of the interfaces could be represented more easily.

7.2.4 Suggestions to improve the development process

This merely is an abstract of what is mentioned in the previous paragraph, with some additional remarks. Regarding the development process, the following remarks were made:

- The policy should be not to change interfaces (without having analysed the consequences), especially not late in the development process.
- Volatile information should be generated automatically (by tools), e.g.
  - The number of changes per interface
  - The number of methods per interface
  - The registration of the clients of an interface
- To prevent changes that are due to the reparation of inconsistencies in code and design, the system architect recommended reflecting a global design in the code before implementing.
- Also to prevent future changes, the test coordinator suggested doing an impact analysis for each problem request, to adjust testing programs to it. Thus, problems could be detected earlier. Also, he thought the Requirement Specification and the Test Specification should be described by different persons, such that they could interact which each other.

7.3 Analysis of the results

In the next subsections we have categorised the main results of the cases and interviews. We have started with the required contents of the Interface Specifications. Then we discussed the interface contents, and finally we discussed the embedding of interface changes to the development process.

7.3.1 Interface Specifications should be more complete and clearly located

Interface Specifications are crucial for the efficient development of the MR product family, because they facilitate the communication about the software's functionalities. This is especially important for

the development at different geographical sites. Also they support the independent development of software parts, because each developer knows at what existing software functionalities he can rely. And finally, new developers need shorter introduction time, if they can find all useful information in the software system.

At the moment, the Interface Specification document does not have an important role at Philips MR. The documentation is not read often, because developers can't find it or think it is out of date, or some are working there for years and thus know the code (which they can view) good enough. Therefore, it is difficult to control the updating process of documentation. This is a problem, because the documentation can express stable information that cannot be hold in the volatile code.

To solve this problem,

- The documentation should have a prominent place in the development process. As the interfaces that are used externally are provided at subsystem level nowadays, it would make sense to provide the Interface Specifications of those interfaces also at subsystem level.
- The contents of the interface specifications need to be clear. Developers need to know what to describe. From the cases we learned the following interface information should be described (see also Appendix D and the generic conclusions, paragraph 8.3):
  - The resources (i.e. other interfaces or libraries) needed to use the interface
  - The semantics of the resources (e.g. methods) the interface provides
  - Usage restrictions, such as calling sequences or access restrictions
  - Error handling (which errors can be expected, in what situation, and what are the consequences)
  - The syntax and data types of the resources the interface provides

From the cases we learned that different interfaces indeed require different information to be described. For the configuration interface, almost all the items we just mentioned should be described better. The interface of the command dispatcher was judged with more satisfaction. But this interface was also more complex than the configuration interface, because there were also concurrency and quality issues. Especially the description of those issues could be improved.

7.3.2 Interfaces contents should be optimised

A good modularisation of the interfaces prevents them from (frequent) changes. Too much functionality should be avoided, as this raises the risk that some of it will be changed; though of course too little functionality makes the interfaces not very usable. To get a right division of functionality over the interfaces, insight in the interfaces' dependencies and the interface's functionalities is needed. This information helps to estimate the impact of changes (e.g. how many other building blocks are affected, do the changes cause users to change their behaviour?), and facilitates seeing what alternatives are possible. Thus negotiations upon the required contents and dependencies of an interface can be done. The goal is to have only little, preferably no, other interfaces included. Thus the domino effect of changes is reduced. Some designers point that to have the interfaces organised this way, the building block structure should be expanded further on.

Furthermore, the functionality provided by an interface should preferably be as generic as possible, such that the interface doesn't need to be adapted to specific demands for each individual user.

Also, to prepare software for configuration changes, the variability points should be made explicit. When they can be traced in the interfaces, it is much easier to adapt the software in case configuration possibilities need to be changed or added. Currently the Philips MR software system provides many configuration options. However, these are implicit, as they are only implemented in the code, and not described at a higher level, thus not easy visible. This is an area of improvement! The configuration options can be described in sub-interfaces (for each configuration option), or be provided as parameters to an interface. Thus, for different configurations, different implementations of a method or interface can be used. This also provides flexibility regarding the addition of configuration options: in that case, only a new sub-interface or parameter needs to be added.

7.3.3 Interface management starts with analysis and negotiation at project's start

To control the interface changes, it is important to determine which interfaces are to be touched – according project's plan that prescribes which features that have to be implemented – in the earliest phase of development. Then, the next step is to change the concerned interfaces, and only then the

implementations may be done. This is to insure that changes to interfaces are well considered and are not just the *consequence* of some implementation change.

For the negotiation on the functionality partitioning, only an abstract interface description would be sufficient. This abstract interface description only needs to reveal the interface's dependencies and functionality, and doesn't need to describe all information needed to use the interface. Also for new interfaces an abstract description can be created. This description may be part of the detailed Interface Specifications (such as we have defined in 7.3.1), or may a separate document.

7.3.4 Other conclusions

The differences between the Requirement Specifications (RS) and Interface Specifications (IS) are not very clear in the Philips MR standards, which makes the developers confused. Often they think some items should be described in the RS document, when they in fact belong to the IS document. Therefore, the standards and templates of the Requirement and Interface Specifications should make the distinction between both very clear. The IS should contain all information a user needs to use the interface, and the RS should only contain the agreements made between clients (may be service engineers or manufacturers) and developers, which define what the requirements to the implementation are. For clarity we have shown the main differences in a table:

RS	IS
What functionality should be implemented, and why?	What functionality is provided? What are the semantics (e.g. what values are returned, which messages are sent)
What are the constraints or dependencies on software/hardware?	What are the usage restrictions? (e.g. calling order of methods)
What errors must be handled? What may be the effects?	What errors can occur? What is the state of the resource when the error has occurred?
What quality requirements must be met?	What quality is guaranteed? (only if it is necessary the user knows this)
What configuration options must be provided?	How to choose which configuration option?
What functionality must be tested?	What other issues are relevant for the user? (e.g. concurrency issues, use cases, which resources are required to use the provided functionality)

In the Philips MR archive, various representations of interface specifications exist. This has originated for historical reasons. For better comprehensibility and consistency, the interface specifications should be represented in (much) the same way. The template of interface specifications we defined [Appendix D] is a good base for this. In addition, agreements on naming conventions can be made.

7.4 Deliverables and recommendations

We have mentioned the concrete deliverables for Philips here. General solutions are described in the next chapter.

7.4.1 Template Interface Specifications

We have rewritten the template for Interface Specifications, in such a way that the contents cover all a user needs to know to use the interface described. We have listed all items that were regarded useful to know according to the interface users of both cases. We also showed items that were regarded useful to know according to only a few users, but then we described in which situation this was useful. The whole interface template is given in Appendix D.

7.4.2 Configuration Interface Specification

In cooperation with the owner of the configuration building block, we have improved the configuration's interface specifications. Therefore we created a new document, such that the interface specifications were better visible. We have added information for all the methods, according to the template for interface specifications we defined. The document is now part of the archive. It only needs to be moved to higher location in the building block structure.

7.4.3 Guidelines for embedding in development process

The interface specifications are not available at higher levels in the building block hierarchy, because before the migration to this hierarchic structure they were part of a flat documentation structure. Thus most of the documentation is still located at lower levels in the building block structure. They should be upgraded to subsystem level, such that external users can find the interface they want to use and its specifications in the same subsystem. Also, the Interface Specification should not be described within the Requirement Specification document anymore.

Also because of the migration from a flat documentation structure to the building block hierarchy, the dependencies between the interfaces cross all different levels of the new structure. As the interface dependencies have been made explicit now (by the include-scoping mechanism), the large amount of cross dependencies is visible. The number of cross dependencies needs to be reduced, by moving interfaces to more appropriate locations in the building block structure. Also interfaces that contain too much functionality should be subdivided into smaller interfaces.

Further improvement to get stable interfaces stable can be achieved by providing as many generic functionality as possible, and providing variability points, which allow the building developer to change the implementation without needing to change the interface.

7.4.4 Further recommendations

Independent development of software part is the easiest for basic software that is used by (many) external parties, and doesn't need to be changed too much. At Philips MR, the Platform subsystem provides the most basic software to other subsystems. Therefore the Platform subsystem is a good starting point to develop as a stand-alone component, which can be developed independently from the other subsystems.

8. Conclusions

In this chapter we describe our generic solutions to manage complex software systems. In the following subsections, we stress the importance to embed the specification of interfaces in an organisation's development process, and we define how interfaces should be managed in that development process, and what information should be described in the interface specifications. Finally, we give a validation of the conclusions we presented, and suggestions for future work.

8.1 Embed interfaces in the development process

Interfaces should be *first citizen* in the organisation's complexity management, as they are the medium to negotiate upon different modularisations and dependencies and changes. To facilitate independent development of software parts, interfaces should be organised such that they are changeable (without affecting the interface users) and preferably independent of other software parts. Thus the domino effect of one change affecting the whole software system can be reduced. Also a reduction of use-dependencies through the software system will make the system more clear and comprehensible.

8.1.1 Define the location and contents of the interface specifications

To make this management upon interfaces possible, it is important that their definition is embedded in the organisation's development process. The location and contents of the interfaces should be described clearly. Currently the specification of interfaces is not explicitly embedded in, e.g., the Rational Unified Process (RUP) [28]. In the RUP the guidelines to manage changes only describe the process of managing changes by version management and posing so-called Change Requests, such that change decisions and impacts are communicated. In our opinion, the management of interface changes should also be described in models such as RUP. In the next subsections we have described issues with respect to interfaces that should be addressed in the software development process.

8.1.2 Define changes to interfaces at project start

The definition of interfaces (changes) should be done early in the development life cycle, directly after or in parallel with the definition of the requirement specifications. Thus, the framework for implementation changes is set, such that only the details need to be filled in. This top-down approach prevents interfaces from changes that are due to ad-hoc implementation changes.

For the negotiation on the (new) functionality to be described in which interface, knowledge about the interfaces' functionality, dependencies and configuration options is needed. This information should not be too detailed, because it is intended to be part of an overview of several interfaces – such that they their (functional) interrelations become clear, and the optimal composition of interfaces is visible. Thus, it is important to have an *abstract* interface description for the management of interfaces. For new interfaces in the system, this abstract interface description may be the base for further development. Later on the detailed Interface Specification can be developed, and the interface be implemented.

8.1.3 Separate internal and external concerns

We already stated that the ultimate way to get independent interfaces is to develop the system parts as *stand alone* components. This might, however, be a step too far for system parts that are at the edge of development, because they have to react quickly to frequently changing requirements, which makes the effort and time to stabilise their functionality quite useless. Though, also in that case, it will be useful to organise those system parts as entities with clearly defined interfaces and specifications, such that negotiations on changes to interfaces are facilitated during project planning, and such that the internals and externals (the deliverables) of those system parts can be separated.
Advantage of this separation is that the distinction between the product's specifications (reflected in the deliverables) and the product's implementation is made clear. Thus it is easier to check the stability of the specifications, while the implementation may change. Also, it is possible to deploy the product as a black box (see Figure 22).

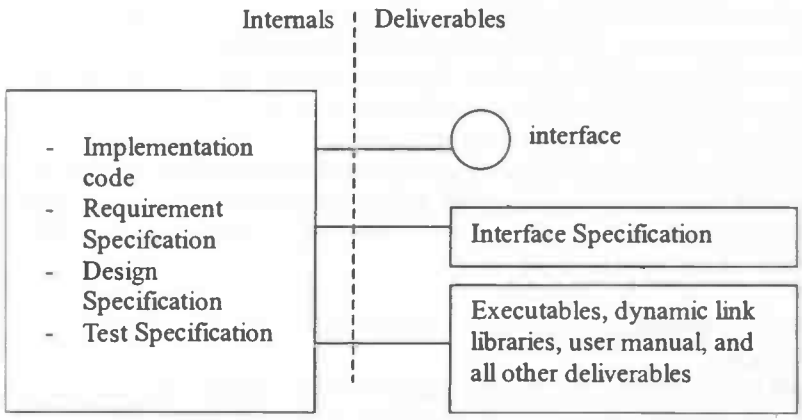


Figure 22: Internal versus external concerns

8.2 Develop flexible interfaces

The goal of interface management is inherent to that of complexity management: to keep the software comprehensible and usable, the number of dependencies between interfaces, and the number of changes done to interfaces should be minimised. Thus the relations between different functionalities remain clear, and developers don't need to search the whole archive to find out what consequences a change to their piece of code will have. But how should this perfect organisation of interfaces be achieved? It always is a trade-off between providing enough functionality such that they don't need to depend on other interfaces, and providing not too much functionality, such that the chance of changes is low.

The following measures help to reduce the number of changes:

- Provide generic functionality, such that the interface doesn't need to be changed when another specific usage of the functionality is requested;
- Use configuration points (diversity interfaces); this is especially useful in product families, where the different products may be the results from different system configurations;
- Use small (immutable/versioned) interfaces, or an interface for each kind of usage (role) or aspect, such that changes regarding a specific functionality issue will only affect the users of that specific functionality.

8.3 Interface Specifications

All the information a user needs (and nothing more) should be described in the interface specifications. The template we have developed for Philips is an example [Appendix D]. The results of the cases showed there is general information each user needs to know, and information that is only needed in specific conditions.

One of the first things the users needed to know was the syntax of the resources provided (including, e.g. the resource names, and names and logical data types of the resource's arguments and results). As this information is given in the interface header file that is per definition available (otherwise it could not even be used), we didn't copy it in the interface specifications. Neither did we describe design issues in the template, as these issues don't concern the users of the interfaces, thus they don't belong to an interface specification.

The items that should be described in all Interface Specifications are the following.

- *Data types*
  These concern the semantics of the data types provided: what is the data type used for, what is the meaning of its elements?
- *Semantic description of the resources*
  This concerns all information related to the meaning and behaviour of the resources, such as a resource description, preconditions to the usage of a resource and postconditions (e.g. a certain value is returned, a message is sent, and so on).
- *Usage conditions*

These concern all conditions that should be satisfied to use the resources, such that their provided functionality is guaranteed. They may include e.g. calling order restrictions, access restrictions (only to be called by users of subsystem X), and *resources required* in order to use the interface (such as a dynamic link library that needs to be linked before).

- *Error handling*
  This concerns the description of the way errors are communicated to the user, and what the consequences of a particular error are. The user needs to know what options he has to react to an error when it has occurred (e.g. can he retry the function, can he use partial results he has retrieved so far, are there alternative resources?).

Items that should be described optionally are the following.

- *Concurrency issues*
  These should be described if the interface can be used by multiple threads. The user then needs to know how the threads behave in all imaginary situations, especially when multiple threads simultaneously enter one of the interface's resources.
- *Quality realisations*
  These should be described if there are relevant quality attribute characteristics the user should be aware of, e.g. a guaranteed minimal level of performance, a warranted level of security, and so on.
- *Variability provided*
  This should be described if the interface allows methods to be configured in some way. It should tell how the configuration parameters can be used, and how they will affect the semantics of the interface.

8.4 Research Validation

8.4.1 What should be described in the Interface Specifications?

We have proposed a template for Interface Specifications [Appendix D]. This template is based on interviews with the users of the interfaces (of both cases). They confirmed that certain information they needed was not in the interface specifications yet. Certain items were only useful in specific environments (e.g. where performance is important, where threads are used, and so on). To ensure that these items will be described if they are necessary, they are part of the template.

8.4.2 What is a good representation of Interface Specifications?

This question has not been answered. Important is that all the items as described in the template can be represented. Most of the items are expressed well by natural languages. A new trend is to express the same information by models, but the modelling techniques are not very mature yet, and are not capable yet to capture all the concerns that should be expressed in the Interface Specifications.

8.4.3 How can Interface Specifications be embedded in development process?

We concluded that the description of interfaces should be defined in the development process, such that its contents and locations are clear. For Philips this means an adaptation to the IS standard and template, and a relocation of the interfaces over the building block structure.
These conclusions are validated by the interviews with the project management: the designers and architects confirmed that interfaces now didn't have the right place in the system, for the historical reasons mentioned.

8.4.4 How can changes of interfaces be managed, and minimised?

First agree on interface changes, then implement. Also define interfaces such that they are flexible for change, e.g. have generic functionality, have configuration points defined.
Here also the designers and architects validated our conclusions. They confirmed that changes to interfaces should be defined early in the development process, so they could manage them better.

8.5 Future work

8.5.1 Modelling interfaces

With respect to development of modelling techniques, it should be said that at the Philips X-Ray department already models are used to generate (abstract) header files from. Therefore they have created a 1-1 mapping of the interface (UML) models to the interface header files in IDL. If it would be possible to describe the whole Interface Specifications in terms of models, then even more details

(implementation code) could be generated from those models. Only therefore more the modelling techniques and generation tooling should be developed much further.

8.5.2 Managing interface changes

Several techniques of decomposition exist. But it is difficult to estimate the best choices with respect to the organisation of interfaces. Further work should be done to identify for which situations which modularisations offer the best insight and most flexibility. Also the representation of configuration points is an interesting area of research.

9. Appendices

Appendix A. Questionnaires

Questionnaire 1 – User interviews

Name interviewee:
Function:

A. "WHAT"
1. (open) What do you need to know to use these methods?
2. (closed) What information is described sufficiently? – for each item from checklist: yes/no
3. (closed) What information do you think is necessary (including what you need in your function) – for each item from checklist: yes/no
4. (closed) What information do you think is not needed at all? – for each item from checklist: yes/no
5. (open) What information do you think is missing?

B. "WHERE"
1. (closed) Where is this item usually described?
2. (closed) Is this the right place? (if not, see sub-questions) – yes/no
3. (open) Where should it be? (and why)

C. "HOW"
1. (closed) When should these items be updated – for each list item: (design phase/implementation phase /test phase/otherwise
2. (open) By who should this information be described/updated? (developer/building block owner/otherwise)
3. (open) Suggestions on interface specifications (what/when/etc)?

Questionnaire 2 – Management interviews

This questionnaire is a guideline for the interviews. Not all questions need to be answered in detail.

Name interviewee:
Function:

Situation 1: Consider the addition of a new subsystem (or a building block to one of the subsystems).

1. What do you want to know of it? – and why?
   (Assume you can't view the internals (code))

2. How to describe…
   (e.g. in what documentation?)

- Relations to other building blocks (resources required, resources provided)
- Functionality (semantic information, e.g. value assignment, error handling, etc.)
- Quality aspects (e.g. performance, availability)
- Configurability/variability (what options, and what consequences)
- Other things you want to know…

Situation 2: Consider the new building block has been changed (not only internally, but also externally visible)

3. How to manage this change?

Sub-questions
- How to control the impact to other building blocks (users)?

(who is responsible for consequences)
- How to minimise the effect of changes?
  (priorities? study evolution of changes?)
- How to notify the users?
  (how, by who)

4. What about the organisation of the development process in general?

- When, what information (could IS be described at the end?)

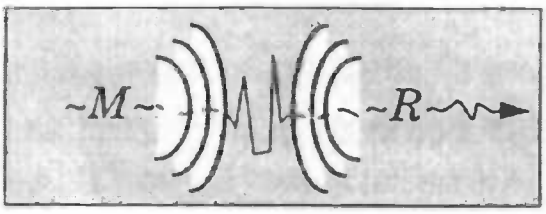
- What conditions must be satisfied?
  (e.g. max. size of methods that are provided to external users)

Appendix B. Interface specifications checklist

1. Locally defined data types (if different from ones provided by programming language)
2. Resources syntax (resource name, names and logical data types of arguments, names and logical data type of results)
3. Resource semantics (all visible effects/results)
   - 3.1. Resource description
   - 3.2. Context/environment description
   - 3.3. Value assignment
   - 3.4. Changes in an element's state
   - 3.5. Events being signalled or messages sent
   - 3.6. Effect to other resources
4. Usage restrictions (resulting from preconditions)
   - 4.1. Calling order
   - 4.2. Access restrictions (e.g. only to be called from a particular thread)
5. Realisation of quality attributes [14]
   - 5.1. Performance: usage of CPU/memory, storage to what medium etc.
   - 5.2. Security: what security is provided
   - 5.3. Scalability: what measures are taken to satisfy the required scalability
6. Resources required (syntax, semantics, usage restrictions)
7. Error handling
   - 7.1. Identification: status indicator / exception raised
   - 7.2. What is the state when the error has occurred: just aborted, or already recovered? What can be done: retrying possible, computing of partial results, or even correcting the error, or using alternative resources?
8. Variability provided: what configuration/parameters for what situations
9. Use cases
10. Design issues
    - 10.1. Rationale
    - 10.2. Constraints
    - 10.3. Patterns used
    - 10.4. Alternatives

Appendix C. Template Interface Specification [old]

IS: <Doc Title>



MR - Software / <Doc Project>

AUTHOR:	<Doc Author>
DOC. ID:	XJSxxx-xxxx
REV. LEVEL:	0.1
DATE:	2003-02-11
STATUS:	Draft

Authorization by:	segment leader
Reviewers:	buildingblock owner, senior designer, customer, author
Commentators:	software architect, ...
Copy:	Archive, ...
Doc.file:	<Document Name>

Abstract

This is the abstract page of the document. The abstract page can be generated by means of the paragraph format "Abstract". Just give the title text the style 'Abstract' and the section will begin at a new page. The text following will be body text (or any formatting you apply afterwards).

Introduction

Read the documentation standard, this template only contains very short hints.

Purpose

Must be filled in (for which users, for which readers).

Scope

Must be filled in (what do you describe, what do you not).

History

Date	Level	Editor	Description
2001-10-17	0.1	<Editor>	First Draft. Hint for the user: This date is the date you first created this document. The date you see in the header is the modification date. Both dates must be changed manually.

Status of document

Definition of terms and abbreviations

Term	Description
<Term>	<Term description>

References

Ref.	Doc. number	Author	Title
<doc.number>		<Author>	ReferenceItem
XJR155-7312		D. Havenith	DR: Multithreaded Software

Overview
Remove if not useful.

Interface Description Function X

Description
Must be filled in.

User Interface
For a function with a user interface.

Declaration
For a function with a SW interface.

Input
For a function with a SW interface.

Output
For a function with a SW interface.

Message Layout
For a function with a message interface.

Message Elements
For a function with a message interface.

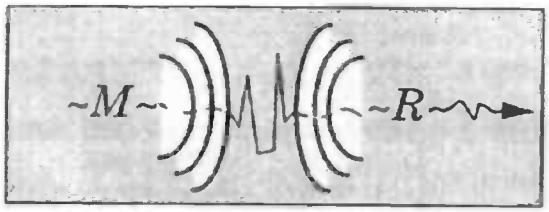
Threading Issues
For a function with a SW interface. If applicable, describe the following items. See also 0.

- Does the function create threads that are relevant outside the component (not *internal* threads)?
- What are the requirements to the threads that use the function, if any? For example: must a thread have called CoInitialize before using the function? If so: which apartment should the thread be in? Or: should this thread have a certain priority?
- What is the behaviour when multiple threads simultaneously enter this function, specifically:
- Will threads block, waiting for other threads to finish? If so: on what kind of functions is the thread waiting (fast changes in global state, long calculations, file operations, user input).
- Is the functions thread-safe under any circumstance? Is re-entrancy by recursion allowed?
- For any callbacks that the function accepts and that may be called by a thread created in the building block, the following issues must be addressed:
- When will the callbacks be performed?
- What may the callback functions do and what not, in specific:
- May callbacks call functions of the originating building block?
- Which other functions may the callback function call?
- May callbacks disable or enable other callbacks?
- May callbacks trigger other callbacks to be started?
- May the callback lock a resource, or wait for a resource to become available? Which specific resources may the callback not lock?

Description of interface
For a function with another type of interface.

Appendix D. Template Interface Specification [new]

IS: <Doc Title>



MR - Software / <Doc Project>

AUTHOR:	<Doc Author>
DOC. ID:	XJSxxx-xxxx
REV. LEVEL:	0.1
DATE:	2003-02-11
STATUS:	Draft

Authorization by:	segment leader
Reviewers:	buildingblock owner, senior designer, customer, author
Commentators:	software architect, ...
Copy:	Archive, ...

Doc.file: <Document name>

Abstract

This is the abstract page of the document. The abstract page can be generated by means of the paragraph format "Abstract". Just give the title text the style 'Abstract' and the section will begin at a new page. The text following will be body text (or any formatting you apply afterwards).

Introduction

Read the documentation standard, this template only contains very short hints.

Purpose

Must be filled in (for which users, for which readers).

Scope

Must be filled in (what do you describe, what do you not).

History

Date	*Level*	*Editor*	*Description*
2001-10-17	0.1	<Editor>	First Draft. Hint for the user: This date is the date you first created t document. The date you see in the header is the modification date. E dates must be changed manually.

Status of document

Definition of terms and abbreviations

Term	*Description*
<Term>	<Term description>

References

Ref.	*doc. number*	*Author*	*Title*
1	<doc.number>	<Author>	ReferenceItem
2	XJR155-7312	D. Havenith	DR: Multithreaded Software

Overview
Remove if not useful.

Interface Description

Description what functionality this interface provides (not method specific, but in general), and for what purpose.

Context Description
Describe how the elements (methods, objects) of the interface relate to their environment. Class diagrams might be useful.

Usage Conditions
Describe everything the user has to do before he can use this interface. Think of:
- Resources required (such as the header file that should be included, libraries and so on)
- General preconditions (e.g. to use the methods of this package, first data has to be initialized)
- Access restrictions (e.g. this interface may only be used by the X group)
- Other requirements (e.g. memory allocation)

Error handling / logging
Mention which error messages / logging is given in what situation, and also mention what the consequences are to the user! E.g. is retrying an option, can partial results be computed, should the error be corrected, can alternative resources be used?

Semantics general
Describe the semantics that the methods provided have in common. Think of:
- Value assignment (may be setting the value of a return argument, but may even be updating a central database)
- Events signalled or messages sent
- Effects to other resources (e.g. opening access to repository enables other methods to retrieve something from that repository; deleting objects may cause other methods producing errors when they try to access those objects)

Variability
If the interface allows methods to be configured in some way, then describe the configuration parameters, and how they affect the semantics of the interactions in the interface.
For each configuration parameter, name and provide a range of values, and specify the time when its actual value is bound (e.g. at runtime).

Threading Issues
For a function with a SW interface. If applicable, describe the following items. See also 0.
- Does the function create threads that are relevant outside the component (not *internal* threads)?

- What are the requirements to the threads that use the function, if any? For example: must a thread have called CoInitialize before using the function? If so: which apartment should the thread be in? Or: should this thread have a certain priority?
- What is the behaviour when multiple threads simultaneously enter this function, specifically:
- Will threads block, waiting for other threads to finish? If so: on what kind of functions is the thread waiting (fast changes in global state, long calculations, file operations, user input).
- Is the functions thread-safe under any circumstance? Is re-entrancy by recursion allowed?

For any callbacks that the function accepts and that may be called by a thread created in the building block, the following issues must be addressed:

- When will the callbacks be performed?
- What may the callback functions do and what not, in specific:
- May callbacks call functions of the originating building block?
- Which other functions may the callback function call?
- May callbacks disable or enable other callbacks?
- May callbacks trigger other callbacks to be started?
- May the callback lock a resource, or wait for a resource to become available? Which specific resources may the callback not lock?

Quality realizations
Optional. Describe the realized quality attribute characteristics that the user needs to know about.
E.g. minimal performance provided, limits to scalability (e.g. if input does not exceed size x, then the given performance is guaranteed), warranted level of security, and so on.

Future extensions, ideas
Optional. Anything that is likely to be changed in this interface in the future.

Data types

Data type name
Short description what this data type is used for, plus the syntax of this data type

Data type name
Short description what this data type is used for, plus the syntax of this data type

Methods

Method name
Description what the method does.

```
RETURNTYPE                      methodname(
(   argumenttype  argumentname
,     argumenttype2     argumentname2
);
```

Preconditions: Description of preconditions.
Postconditions: Description of postconditions.
Result (in case of enumeration, fill the following fields, otherwise delete them).
E.g. In case value x is available / the method has succeeded.

Method name
Description what the method does.

```
 RETURNTYPE                      methodname(
(   argumenttype  argumentname
,     argumenttype2     argumentname2
);
```

Preconditions:	Description of preconditions.
Postconditions:	Description of postconditions.

ation, fill the following fields, otherwise delete them).
ailable / the method has succeeded.

First Appendix.

Add your first appendix here.

AppSection

Use Appendix 2 style for Sections

Second Appendix

Add your second appendix here.

<<< END OF DOCUMENT >>>

Appendix E. Configuration Interface Specification [old]

Beneath is part of the old Configuration Interface Specification, which was described in the MR Configuration Model [53]. No data types were described there, and the methods were given as follows (only three methods are shown, just to indicate how the document was organised).

General overview of the implementation in WNT

The following table gives an overview of how the implementation in WNT will be; a short note of the implementation in VMS is also listed in the last column of the table. [11] gives an overview of all attributes from the MR Configuration mode and the corresponding items from Figman in section 1.8.

All the functions started with AWASW_get_ and AWASW_connect(), AWASW_close() in the fifth column will be declared in awaswloc.h and defined in awaswloc_wnt.c.

- The "get" functions will take the address of a variable as input parameter to store the output value and return boolean to indicate the value is get successfully or not. The translation from other data types to required type (normally to enumeration) is done here.
- The AWASW_init() function will use "MRScanner" as systemID to connect to the repository. Afterwards, all the "get" function can be executed. The "connect" function takes *void* parameter and returns *boolean* to indicate whether the function call is succeed or not.
- The AWASW_exit() function will close the connection. The "close" function takes *void* function and return *boolean* to indicate whether the function call is succeed or not.

Notation used inside table:

+	include only. Basically, no change is necessary for such functions.
-	obsolete. Those functions will be removed from the archive.
3	yes. Function interface will stay the same, necessary changes will be made.
?	to be decided
new	not in aswwnt archive, but already added to or are going to be added to aswnt712 archive.

Configu-ration items	Will be imple-mented	Functions	Tag	Attributes name in configuration repository and possible implementation directives	Imple-mentation in VMS
3	3	AWASW_init (void)		AWASW_connect() which uses the logical name GYRO_SYSTEM_ID to identify the configuration data model	GYROFIX .IDX DGFCG_o pen_read() AWASW KEYS_init
3	+	AWASW_set_resp()			
3	3	AWASW_gr_switchbox _type()	3001, 9032	GradientSwtich.GradientSwitch Type AWASW_get_gradient_switch_ type	

Appendix F. Command Dispatcher Interface Specification

Beneath is the whole chapter of the Command Dispatcher Interface Specification (from the RS Command Dispatcher [48]).

The Command Dispatcher will implement two types of interfaces. The first interface is intended for users without knowledge of COM and the second interface for user with knowledge of COM.

Data types

In order to communicate to and from the Command Dispatcher some data-types are used. The description of these data-types can be found in the coming paragraphs.

string

In the 'C' style interface the strings will be passed as MR Unicode compliant OST_char. The COM interface and the interface between the performers and the Command Dispatcher are both based on COM. In COM the BSTR data-type, which is also Unicode compliant, is used to pass string arguments.

result

In order to pass back a result from the command performer to the client, via the Command Dispatcher the result data-type is used. This result will be an integer data-type. The result integer can have the following values:
0 = unspecified
1 = success
2 = failed
3 = queued
4 = unknown_command
5 = incorrect_nr_arguments

invocation_id

The invocation_id will be used to uniquely identify an invoked command. The invocation_id will be send from the Command Dispatcher to the command performer. If the command is queued by the performer, and the result is returned later, the invocation_id should be specified by the performer. The invocation_id should have a unique value. In order to guarantee this, the integer value of the invocation_id is constructed using a rotation number.

P_COMD_ARG_VAL_STRUCT

The P_COMD_ARG_VAL_STRUCT is used to specify function arguments. Currently only integer, floating point and string arguments are allowed. Because the data-type of these arguments can vary, the structure contains an element, which specifies the data type, and elements, which contain the actual data.

```
structure
{
  TYPE_ENUM type;              /* empty, word, string, int or real  */
  int       int_value;         /* Always filled, non-numeric = 0  */
  double    double_value;      /* Always filled, non-numeric = 0.0 */
  int       string_length;     /* Length of 'string_value' */
  OST_char  string_value_arr[MAX_STRING]; /* Always filled  */
} P_COMD_ARG_VAL_STRUCT;
```

Callback functions

The completion callback function is used to notify the client that a performer has completed an asynchronous command. Two callbacks function can be specified, one for the 'C' style interface and one for the COM interface.
The P_COMD_COMPLETION_CB_FUNC for the 'C' style interface will have the following layout:

P_COMD_COMPLETION_CB_FUNC (void* client_data_ptr,
P_COMD_RESULT_ENUM result)

The callback function for the COM interface will have the following layout. Note that this must be an IDispatch based interface and it must be exposed through the type library of the client.

Completion ([in] VARIANT* return_arguments, [in] int result)

An alternative completion callback which also gives you back the client data originally provided for the command can also be used. For this, you need to implement a specific IP_COMD_CommandDispatcherNotify interface on the client. Note that implementing this interface overrides the prior one, which will then never be called. This interface has a different Completion function with the following signature:

Completion ([in] int result,
[in] VARIANT* client_data_ptr,
[in] VARIANT result_arguments)

Note that this interface is also IDispatch based, but is accessed as a custom interface by Command Dispatcher, making it more efficient.

Command Dispatcher interface description

The clients will need an interface to the Command Dispatcher to issue commands. In the current situation (Orion) the clients have an understanding of the command to be performed. This means that the layout of the command is known at design time (number of arguments, argument types and argument layout). So for the current interface retrieving this kind of information is not needed. However in the future this might become an issue, see Chapter 8.

'C' style interface Command Dispatcher

The 'C' style interface of the Command Dispatcher will have the following functions:

Interface function: IssueSynchronousCommand
Arguments: OST_char command_name_arr[]
int nr_arguments
P_COMD_ARG_VAL_STRUCT arguments_arr[]
Return type: P_COMD_RESULT_ENUM result
Description: This function issues a synchronous command to the Command Dispatcher. The IssueSynchronousCommand function will return after the performer has performed the command. The issuer of the command is blocked for any input. The Command Dispatcher will use the command_name_arr to determine the performer and issue the command to the appropriate performer. The arguments of the command have to be in the same order as defined in the '.idl' file of the performer. The return value will indicate the success or failure of the command.

Interface function: IssueAsynchronousCommand
Arguments: OST_char command_name_arr[]
int nr_arguments
P_COMD_ARG_VAL_STRUCT arguments_arr[]
P_COMD_COMPLETION_CB_FUNC* completion_cb_ptr
void* client_data_ptr
Return type: P_COMD_RESULT_ENUM result
Description: This function issues an asynchronous command to the Command Dispatcher. The IssueAsynchronousCommand function will return immediately and invoke the command in the background. The Command Dispatcher will use the command_name_arr to determine the performer and issue the command to the appropriate performer. The arguments of the command have to be in the same order as defined in the '.idl' file of the performer. The result of the command will be returned using the completion callback function, which will also return the client_data_ptr. The return value will indicate the success or failure of the command.

COM interface Command Dispatcher

The COM interface of the Command Dispatcher will use the standard IDispatch interface. This interface is chosen, because it enables type library information. This may be needed in the future to send command layout information, see the chapter on Future Requirements, Chapter 8. The users of the Command Dispatcher have to specify a interface, which implements the Completion method. This method is used to inform the issuers of the commands of the result of asynchronous issued commands.

The COM interface to the Command Dispatcher will have the following methods:

Interface method: IssueSynchronousCommand

Arguments:	BSTR command_name	// Command Name to be // issued
	VARIANT arguments	// Variant containing the // command arguments
	VARIANT* return_arguments_ptr	// Variant pointer for returning // the result arguments
	int* result_ptr	// integer returning the result
Return type:	HRESULT	
Description:	This function issues a synchronous command to the Command Dispatcher. The IssueSynchronousCommand function will return after the performer has performed the command. The issuer of the command is blocked for any input. The Command Dispatcher will use the command_name to determine the performer and issue the command to the appropriate performer. The arguments of the command have to be in the same order as defined in the '.idl' file of the performer. The return value will indicate the success or failure of the command.	

Interface method: IssueAsynchronousCommand

Arguments:	BSTR command_name	// Command Name to be // issued
	VARIANT arguments	// Variant containing the // command arguments
	IUnknown* callback_ptr	// Interface pointer of the // Callback interface
	VARIANT* client_data_ptr	// Pointer to the client data
	int* result_ptr	// integer returning the result
Return type:	HRESULT	
Description:	This function issues an asynchronous command to the Command Dispatcher. The IssueAsynchronousCommand function will return immediately and invoke the command in the background. The Command Dispatcher will use the command_name to determine the performer and issue the command to the appropriate performer. The arguments of the command have to be in the same order as defined in the '.idl' file of the performer. The result of the command will be returned using the completion callback function, which will also return the client_dat. The return value will indicate the success or failure of the command.	

The interface to the Command Dispatcher will look in terms of an IDL description similar to the interface below:

```
interface IP_COMD_CommandDispatcher : IDispatch
{
   [id(1), helpstring("method IssueSynchronousCommand ")] HRESULT
IssueSynchronousCommand(
                [in] BSTR command_name,
                [in] VARIANT arguments,
                [out] VARIANT* return_arguments,
                [out, retval] int* result);
   [id(2), helpstring("method IssueAsynchronousCommand ")] HRESULT
```

```
IssueAsynchronousCommand (
                [in] BSTR command_name,
                [in] VARIANT arguments,
                [in] IUnknown* callback_ptr,
                [in] VARIANT* client_data,
                [out, retval] int* result);
};
```

Command performer interface guidelines

The interface between the Command Dispatcher and the command performers will be based on COM. In order to refrain the Command Dispatcher of knowing compile time all command performer interfaces, the IDispatch type interface is used for the performers. This interface facilitates the COM late binding protocol.
Because all performers implement different commands, it is impossible to specify a general interface for them. The interfaces of the various performers should however adhere to some guidelines.

- Each performer interface should be implemented using the standard IDispatch interface.
- The method names will be used by the Command Dispatcher to issue the commands, so care should be taken that these method names are stored in the Registry.
- The 'C' style issuers of commands can only issue commands with integer, floating point and string arguments. So performers should currently only use these types of arguments.
- Each method should return a unique invocation_id and the result of the command.
- If the command performer has queued a command and return the result value queued to the invoker of the command, it should always send a notification back, stating the result of the queued command.

Example command performer interface

The interface to the command performer will look in terms of an IDL description similar to the interface below:

```
interface IScanControl : IDispatch
{
    [id(1), helpstring("method StartScan")] HRESULT StartScan(
                [in] int invocation_id,
                [in] IUnknown* callback_ptr,
                [out] int* result,
                [out] VARIANT* result_arguments);
    [id(2), helpstring("method StopScan")] HRESULT StopScan(
                [in] int invocation_id,
                [in] IUnknown* callback_ptr,
                [out] int* result,
                [out] VARIANT* result_arguments);
    [id(3), helpstring("method RemoveScan")] HRESULT RemoveScan (
                [in] int scan_id,
                [in] int invocation_id,
                [in] IUnknown* callback_ptr,
                [out] int* result,
                [out] VARIANT* result_arguments);
};
```

Command Performer notification interface

In order to send notifications from the command performers to the Command Dispatcher a standard interface is used. This interface will currently only be used for notifications of completed commands. In the future this interface will also be used for progress reporting.

```
ICommandPerformerNotification
{
    [id(1), helpstring("method Completed")] HRESULT Completion(
                [in] int invocation_id,
                [in] int result,
```

```
                [in] VARIANT result_arguments);
};
```

Appendix G. Configuration Interface Specification [new]

Beneath is part of the new Configuration Interface Specification. The IS of the AWCOIL package that is described in the same documented (that existed already) is omitted here. Only one of the data types and three methods are shown, just to indicate how the new IS document is organised.

Interface Specification AWASW package

This package provides C-style interfaces to access the configuration data in the configuration repository. The configuration repository contains several objects, which have a number of attributes. By calling AWASW_init() a copy of the configuration repository contents is cached. Afterwards, the other AWASW methods can retrieve or set values of the attributes from or to the cache. Information about all the attributes can be found in the document that describes the MR Configuration Model [1].
The caller of a function is responsible to meet the preconditions. If the preconditions are not satisfied, there is a severe programming error, in which case the AWASW package will abort.

AWASW characteristics

Context description

The relations of the different objects, and their attributes, in the configuration repository are described in class diagrams in [1].

Usage conditions

To use this package, the awdll@*.dll must have been linked, and the header file awaswglo.h must have been included. To use one of the AWASW methods, first AWASW_init() must have been called, to open the connection to the configuration repository.

Error handling / Logging

There is no logging. If the preconditions are not met when calling one of the methods, the AWASW package will abort.

Semantics general

For all methods yields:
There are no effects to other resources. The configuration repository can not be changed.
There are no messages sent or events signalled.

Variability

There is no variability provided.

Threading

No threading is used; no concurrency issues exist.

Validation

There are no validation rules available.

Future extensions, ideas

This interface could be partitioned and then divided into smaller interfaces (appropriated for different usage). For naming consistency it would also be good to rename each method that gets an attribute value to get_*name*, and each method that sets an attribute value to set_*name*. And finally it would be consequent to have symmetric methods, that says for each get_method there should also exist a set_method.

AWASW data types

AWASW_AVAIL_ENUM

AWASW_AVAIL_ENUM is the return value of AWASW_headset_available, AWASW_sld_option, AWASW_ecg_adapt_filt_cabling, AWASW_tte_option and AWASW_docking_unit_type.

```
typedef enum
{
    AWASW_AV_MIN = FALSE - 1,
    AWASW_AV_NOT_AVAILABLE = FALSE,
    AWASW_AV_AVAILABLE = TRUE,
    AWASW_AV_MAX
} AWASW_AVAIL_ENUM;
```

AWASW methods - general

AWASW_init

This function initialises the ASW-package. The connection to the configuration repository is opened for read access.

```
void        AWASW_init
(  void
);
```

Postconditions:	The connection to the configuration repository is built.
Returns	Nothing.

AWASW_set_resp

This function sets the type of the optional respiratory device.

```
void                          AWASW_set_resp
(  AWASW_RESP_ENUM      set_resp_type
);
```

Preconditions:	AWASW_init() must have been called before calling this function.
Postconditions:	The value of the resp type in the cached configuration repository is set to set_resp_type
Returns	Nothing.

AWASW_gr_switchbox_type

This function returns the gradient switchbox type from the scanner configuration.

```
AWASW_GR_SWB_TYPE_ENUM
typedef enum
{
    /* value software                  text in file                          */
    AWASW_GSBT_MIN = ENUM_MIN,
    AWASW_GSBT_NONE                 /*< @NO_GR_SWITCHBOX                  >*/,
    AWASW_GSBT_DUAL_PAR_XYZ         /*< @DUAL_PARALLEL_XYZ                >*/,
    AWASW_GSBT_MAX
} AWASW_GR_SWB_TYPE_ENUM;

AWASW_GR_SWB_TYPE_ENUM  AWASW_gr_switchbox_type
(  void
);
```

Preconditions:	AWASW_init() must have been called before calling this function.
Postconditions:	Gradient Switch Type is returned.
Returns	AWASW_GR_SWB_TYPE_ENUM, corresponding to GradientSwitch.GradientSwitchType (tag 3001,9032). See [1] for attribute information.
NONE	No gradient switchbox is used.
Gradient Switchbox	"Gradient Switchbox" is used.

Bibliography

1 A. Agrawal, T. Levendovszky, J. Sprinkle, F. Shi, G. Karsai, "Generative Programming via Graph Transformations in the Model-Driven Architecture". OOPSLA conference on Object Oriented Programming, Systems, Languages and Applications, 2002.

2 E. Barszcz, S.K. Weeratunga, E. Pramono, "A model for executing multidisciplinary and multizonal programs". NAS (NASA Advanced Supercomputing) research, 1993.

3 J. Bettin, "Practical Use of Generative Techniques in Software Development Projects: an Approach that Survives in Harsh Environments". OOPSLA 2001 workshop position paper.

4 J. Bosch, "Design & Use of Software Architectures – Adopting and evolving a product-line approach". ACM Press/Addison-Wesley, 2000.

5 J. Bosch, "Object Oriented Frameworks – Problems and Experiences", in "Building Application Frameworks", M.E. Fayad, *et al.*, 1999.

6 P. Bouaziz, L. Seinturier, "From Software Parameterization to Software Profiling". Tutorial and Workshop on Aspect Oriented Programming and Separation of Concerns (AOPWS), 2001.

7 M.G.J. van den Brand, P. Klint, and C. Verhoef, "Re-engineering needs Generic Programming Language Technology". SIGPLAN Notices 32 (2), 1997.

8 R.J. Bril, A. Postma, R.L. Krikhaar, "Embedding architectural support in industry". IEEE International Congres on Software Maintenance, 2003.

9 F.P. Brooks, "The mythical Man-Month". Addison-Wesley, 1995.

10 P. Clements et. all., "Documenting Software Architectures – Views and Beyond", SEI Series in Software Engineering, 2003.

11 K. Czarnecki, U. Eisenecker, "Generative Programming - Methods, Tools and Applications". Addison-Wesley, 2000.

12 M. Fowler, "UML Distilled second edition – a brief guide to the standard Object Modeling Language". Addison-Wesley, 2001.

13 D. Frankel, "Using Model-Driven Architecture to Manage Metadata". IONA whitepaper, 2002.

14 J. van Gurp, "On the design and Preservation of Software Systems", Ph.D. thesis, 2003.

15 B. Ibrahim, "Semiformal Visual Languages, Visual Programming at a Higher Level of Abstraction", Fifth Conference of the ISAS, Orlando, 1999.

16 I. Jacobson, M. Griss, P. Jonsson, "Software Reuse – Architecture, Process and Organisation for Business Success". ACM Press, 1997.

17 M. Jaring, R.L. Krikhaar, and J. Bosch, "Visualizing and Classifying Software Variability in a family of Magnetic Resonance Imaging Scanners". Software – Practice and Experience, 2003.

18 D. Keppel, "Managing Abstraction-Induced Complexity". Technical Report, UWCSE 93-06-02, University of Washington, 2003.

19 P. Kroll, "The Spirit of the RUP". Rational Software, 2001.

20 S. McConnell, "Daily Build and Smoke Test", IEEE Software – Best Practices Vol. 13, July 2003.

21 Naval Center for Cost Analysis (NCCA) software team, "Fourth Generation languages Issue Paper", 1996.

22 S. Nystedt, C. Sandros, "Software Complexity and Project Performance". Master thesis by Nystedt and Bachelor thesis by Sandros, 1999.

23 R. van Ommering, "Configuration Management in Component Based Product Populations". International Conference on Software Configuration Management (ICSM), 2001.

24 R. van Ommering, F. van der Linden, J. Kramer, J. Magee, "The Koala Component Model for Consumer Electronics Software". IEEE Computer, Vol. 33, No. 3; 2000.

25 R. van Ommering, "Techniques for Independent Deployment to Build Product Populations". Working IEEE/IFIP Conference on Software Architecture (WICSA'01), 2001.

26 R. van Ommering, J. Bosch, "Widening the Scope of Software Product Lines – From Variation to Composition". Proceedings of the Second Software Product Line Conference (SPLC2), pp. 328-347, 2002.

27 M.C. Paulk, "Structured approaches to Managing Change". 1999.

28 L. Probasco, "The Ten Essentials of RUP – The Essence of an Effective Development Process". Rational Software White paper, 2000.

29 C. Robson, "Real World Research". Blackwell Publishers, 2nd edition, 2002.

30 T. Rötschke, R.L. Krikhaar, D. Havenith, "Multi-View Architecture Trend Analysis for Medical Imaging". Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), 2001.

31 R.Kobro Runde, K. Stølen, "What is Model Driven Architecture", 2003.

32 P. Schmidt, "Interface management issues in Component-based Architectures". Ground System Architectures Workshop (GSAW), 1999.

33 J. Franklin Skipper, "Assessing the Suitability of UML for Capturing and Communicating Systems Engineering Design Models". Vitech Corporation, 2002.

34 I. Sommerville, "Software Engineering". Peerson Education Limited 6th edition 2001.

35 G. van der Stap, A. Strack van Schijndel, H.Goedvolk, D.B.B. Rijsenbrij, "An architectural model for component based development". First working IFIP Conference on Software Archiecture (WICSA1), 1999.

36 L. Sterling, E. Shapiro, "The art of Prolog – advanced programming techniques". MIT Press series in logic programming, 1994.

37 C. Szyperski, "Component Software – Beyond object-oriented programming". ACM Press/Addison-Wesley, 1998.

38 C. Szyperski, "Component Technology – What, Where and How?". International Workshop of Component Based Software Engineering (ICSE), 2003.

39 J.P. Wadsack, J.H. Jahnke, "Towards Model-Driven Middleware Maintenance". OOPSLA conference on Object Oriented Programming, Systems, Languages and Applications, 2002.S

40 H. Wegener, "Agility in Model-Driven Software Development? Implications for Organisation, Process and Architecture", OOPSLA conference on Object Oriented Programming, Systems, Languages and Applications, 2002.

41 W. Zhao, "A generative and Model-Driven Framework for Automated Software Product Generation". International Workshop of Component Based Software Engeneering (ICSE), 2003.

Internet pages:

42 Free online dictionairy of computing (foldoc), "Fourth generation language". http://wombat.doc.icac.uk/foldoc.

43 MR Intranet, "Design Standards". http://pww.mr.ms.philips.com/development.

44 Object Management Group, "Executive overview of Model Driven Architectures". http://www.omg.org/mda/executive_overview.htm.

45 Philips Medical Systems. http://www.medical.philips.com.

46 Software Reality, "Domain oriented software". http://www.softwarereality.com/design/domain_oriented.jsp.

47 What Is encyclopedia, "Programming language generations". http://www.whatis.com.

Internal pages (only visible to PMS-MR employees):

48 A.P. Brouwer, "RS: Command Dispatcher". MR Software Requirements and Interface Specification, XJS-155-7008, April 2001.

49 W. Doornekamp, "Coding Standards". MR Software Quality Manual, XJV 155-0701, Jan. 2002.

50 M. de Jong, "UM Build Process for Developers". MR Software User Manual, XJS 155-7122, Jan. 2003.

51 R.L. Krikhaar, "Architecture and Design". MR Software Quality Manual, XJV 155-0601, Jan. 2002.

52 R.L. Krikhaar, "MR Building Blocks". MR Software Specification, XJS 154-1386, Sept. 2001.

53 X. Lin, "MR Configuration Model". Including MR Software Interface Specification, XJS 155-6634, May 2003.

54 F. Mannens, A. Zephat, "Standard: Interface Description". MR Software Requirements to Interface Specification, XJS 141-4214, March 2002.

55 F. Mannens, A. Zephat, "Standard: Functional Specification". MR Software Requirements to Functional Specification, XJS 141-4071, March 2002.

56 M. van der Vliet, "Interface changes during the development life cycle", 2003.

57 A. Zephat, "Standard: OO Interface Specification". MR Software Requirements for Object Oriented Interface Specification, XJS 154-1328, November 1999.