MSc Thesis Report in Computer Science

Documenting Software Architectures

Documenting Software Interfaces

Lei Xiao

1160877

L.Xiao@EWI.TUDelft.NL

June 2004

PHILIPS

Documenting Software Architectures

Documenting Software Interfaces

Author: Lei Xiao
Student Number: 1160877
June 2004

Supervisor: Ir. Bas Graaf

Committee Members:
Prof. Dr. Arie van Deursen
Ir. Frans Ververs
Ir. Bernard Sodoyer
Ir. Bas Graaf
Rob Kommeren

Table of Content

TABLE OF CONTENT I

PREFACE III

ABSTRACT V

1. INTRODUCTION 1

1.1 CONTEXT 1
1.2 GOAL 2
1.3 STARTING POINT 2
1.4 APPROACH 3
1.5 CONTENT AND ORGANIZATION 3

2. DEFINITION, IMPORTANCE, STAKEHOLDERS 5

2.1 DEFINITION 5
2.2 IMPORTANCE 5
2.3 CONTENTS IN ONE INTERFACE DOCUMENT 5
2.4 STAKEHOLDERS OF INTERFACE DOCUMENTATION 6

3. TEMPLATE FOR DOCUMENTING SOFTWARE INTERFACES 11

3.1 INTRODUCTION 11
3.2 SERVICE 12
3.3 EXTERNAL DATA TYPES, VARIABLES AND CONSTANTS 15
3.4 ERROR HANDLING 17
3.5 LIMITATIONS AND QUALITIES 18
3.6 COMPONENT REQUIRES 19
3.7 PROTOCOL 19
3.8 RATIONALE AND ISSUES 21

4. APPLICATION OF THE TEMPLATE 23

4.1 SUPER AUDIO CD PLAYER 23
4.2 DVD RECORDER 24
4.3 APPLICATION EVALUATION 26
4.4 EVALUATION CONCLUSION 27

5. EXPERIENCES WITH SOFTWARE INTERFACE DOCUMENTS 29

5.1 ATTRIBUTES OF GOOD INTERFACE DOCUMENTATION 29
5.1.1 ACCURACY 29
5.1.2 CLARITY 29
5.1.3 INTEGRITY 29
5.1.4 MAINTAINABILITY 29
5.2 EXPERIENCES WITH DOCUMENTING SOFTWARE INTERFACES 30
5.2.1 DOCUMENTING FROM THE READER'S POINT OF VIEW 30
5.2.2 FOCUS ON THE PROPERTIES BELONGING TO THE INTERFACE 30
5.2.3 USE A STANDARD TEMPLATE TO CREATE AN INTERFACE DOCUMENT 31
5.2.4 FIND A BALANCE POINT BETWEEN TOO LITTLE INFORMATION AND TOO MUCH 32
5.2.5 SUGGESTIONS FOR DOCUMENTING SERVICE SECTION 33
5.2.6 RECORD RATIONALE 34
5.2.7 UPDATE DOCUMENT IN TIME 35
5.3 EXPERIENCES WITH EVALUATING INTERFACE DOCUMENTS 35
5.3.1 PROPOSED METRICS 36
5.3.2 EXPERIENCES WITH USING THE METRICS FOR EVALUATION 38

6. CONCLUSION 43

GLOSSARY 45

APPENDIX A: TEMPLATE OUTLINE 47

APPENDIX B: EVALUATION RESULT 49

REFERENCE 51

Preface

This document is a thesis report of a Master of Science student major in Software engineering at Delft University of Technology. In TU Delft, the master thesis project is defined as an activity with strong ‘engineering’ aspects such as analysis, design and implementation of algorithms, systems, methods and tools. Because of my personal interest in embedded real time system and software engineering, I joined the MOOSE project. In this project, I choose documenting software interfaces as my thesis project.

In this thesis project, I designed a template for documenting software interfaces. This template is based on the current techniques and my personal experiences. Later I have applied this template to two applications belonging to the Philips Digital Systems Lab (PDSL). During the design and application stage, I also gained some experiences with documenting software interfaces. This thesis describes both the template and my experiences. The goal of this thesis is to introduce some knowledge about document software interfaces in order to help people produce a high quality interface document.

For the first-time readers of this report, my suggestion is to concentrate on chapter 1 in order to get some general information about this thesis project, browse chapter 2 to gain some basic knowledge about documenting software architecture, concentrate on chapter 3 to learn the template for documenting software interfaces, skim through chapter 4, at last, read chapter 5 to refer to my experiences, how I use the template, and how I evaluate the interface documents.

People who just want to achieve some background about documenting software interfaces, should read chapter 2. And people who want to get more knowledge about documenting software interfaces, I suggest reading chapter 3.

Readers who want to evaluate my result in the thesis work should browse chapter 1 and concentrate on the following five chapters. These five chapters provide knowledge about documenting software interfaces, as well as the evaluation result and necessary analysis.

I believe in the importance of software architecture for developing complex software system. But architecture need effective communication, I believe that documentation is the key for the communication. I hope this thesis project can provide useful information that can help people to solve the problem that they have met when they document software interfaces.

At the last paragraph, I would like to provide my acknowledgement to my parents, Mr. Toetenel, Mr. Spierenburg and Mr. Kommeren. At last, especially thank to Mr. Bas Graaf, my supervisor, thank you for your kindly assistance during my thesis work.

Without your support, I cannot finish thesis project so well.

-- Lei Xiao

Abstract

Software architecture is an important aspect in software engineering because it allows people to work cooperatively and productively together to produce and maintain a complex software system. Documenting software architecture is important because the architecture documentation serves as a blueprint for a software system. The documentation is the best artifact for early analysis, and the key for post-development maintenance.

As an important part of architecture, interfaces are also worthy of documenting. This report precisely focuses on documenting software interfaces. It develops, selects and tailors some technologies for documenting software interfaces to satisfy the requirement of the industrial environment.

Two applications are mentioned in this report. Both of them are offered by Philips. Philips provided some interface design documents and requests these documents can be clearer and easier to understand. It is a good opportunity for the author to redocument this documentation in order to satisfy the demand of Philips by tailoring some techniques. In order to redocument the documents for Philips well, the author should answer the question of how a software component's interface should be documented so that others can easily use the documentation to implement, apply, and maintain this software component.

To answer the question above, a template was defined for documenting software interfaces in this report. The template specifies a structure for an interface document and illustrates it with examples. This report also presents the experiences of the author on documenting software interfaces. The experiences are acquired from the redocumenting process of the documents provided by Philips. To evaluate the quality of the software interface documents, an evaluation method is introduced in this report. At last, this report gives the evaluation results of the applications in Philips. The evaluation results clearly illustrate that the mentioned question was answered satisfactorily.

1. Introduction

Software architecture, a foundational concept for successful development of complex software systems, has been paid much attention to in the recent years. So what is software architecture? Various communities use various definitions, here one definition is adopted: The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them [1]. This definition clearly shows that software interfaces, the externally visible properties of the components, are vital parts of a software architecture. Therefore, a critical part of documenting software architecture includes documenting software interfaces. This report tackles the problem of how software interfaces should be documented.

1.1 Context

In the MOOSE[1] project (this project belongs to ITEA[2]), it is suspected that the design documentation at PDSL[3] (Philips Digital Systems Lab) does not serve its full potential use during different phases of software development (e.g. implementation, maintenance). One of the reasons is that this documentation is full of implicit assumptions. This makes it often the case that only the authors or people that work on a specific project for a long time are able to understand it. However, these people should typically not be the intended audience as they are very well informed already about the specific design (wrote it themselves or have many years experience with it). So actually this documentation is not used at all for any purpose other than structuring the author’s thoughts by forcing him to think over his decisions again. This leads to situations where new project members face long learning curves and code is actually used as the most up-to-date documentation.

Therefore, there is an assignment for the author. The author has the responsibility to redocument an existing interface specification (software component specification, SCS) that is a good example of the problems mentioned in the preceding paragraph.

[1] MOOSE is an ITEA project starting in the beginning of year 2002 and ending in the end of June 2004. The main aim of the project is to improve product quality and software development productivity through optimized integration and interfacing of systems and software engineering, requirements engineering, product architecture design and analysis, software development and testing, and product quality and software process improvement methodologies.

[2] ITEA, Information Technology for European Advancement, is an eight-year strategic pan-European programme for advanced pre-competitive research and development in embedded and distributed software.

[3] The Philips Digital Systems Lab - Eindhoven is a very special kind of development laboratory. The people develop new systems, concepts and prototypes for ground-breaking Consumer Electronics products and related professional areas for Business Units within Philips.

This interface specification documents the external visible behavior of the component under consideration. The internal workings of the component are outside the scope of this assignment.

1.2 Goal

In order to document software interfaces well, one question should be answered, that is, how a software component's interface should be documented so that others can easily use the documentation to implement, use, and maintain this software component.

Although currently many books and papers on documenting software interfaces are available, most of them are not exactly what are needed in an industrial situation. The author tried to develop, select and tailor some technologies for documenting software interfaces to satisfy the requirements of the real industrial environment. He defines a template for documenting software interfaces. This template has been applied to two applications of PDSL. After evaluating the redocumented interface documents, the result demonstrates that the quality of interface documents will be improved by using the template combined with the experiences of the author. Through these activities, the question mentioned will be answered satisfactorily.

1.3 Starting Point

The master curriculum of TU Delft requires that before the thesis project, the author should finish a research assignment first. In this case the research assignment serves as a preparatory study or literature study for the Master thesis.

In this research assignment, the techniques in the book, *Program Development in Java. Abstraction, Specification, and Object-Oriented Design* [2] were studied. This book defines a format for specifications, discusses the properties of a good specification, and provides several examples. Application of these techniques will lead to a clean (functional) component interface specification in a pre-/postcondition-like format. Besides this activity, other references which are helpful for this project were studied, for instance, the book, *Documenting Software Architecture. Views and Beyond* [3], it provides knowledge about architecture views tells people how to document software interfaces and behavior, and gives some rules for sound documentation.

As the result of the research assignment, one rudimentary template for documenting software interfaces was designed. This template is the starting point of the thesis project.

1.4 Approach

The template mentioned in section 1.3 needs to be further tailored to the environment (product type, existing PDSL development processes, etc...). Components that are documented according to the templates should be easier to implement, maintain and integrate in products.

It is expected that redocumenting will be very hard for the specific component (Hard Disk Navigator component for DVD-RW products, application 2) under consideration. Therefore components (Loader Services components for SACD1000 player, application 1) of which a more complete interface specification exists will also be redocumented using the tailored techniques. This will give an indication of what information is missing in the other component's interface documentation.

At last, the redocumented document should be evaluated. A conclusion should be drawn after evaluating. The ultimate result of this thesis project should include a template for documenting software interfaces, the author's experiences with documenting software interfaces, an evaluation method, and two high quality interface documents which are redocumented by using the template.

1.5 Content and Organization

This report is intended to cover some technologies of documenting software interfaces. It is divided into six chapters. The first chapter introduces the background of the project and the plan and stage of the project. The second chapter discusses the importance of documenting software interfaces, the stakeholders of interfaces documentation and the contents which should be included in the software interfaces documentation. The third chapter brings up a template for documenting software interfaces, a manual for this template that teaches people how to use this template, and some guidelines for the use of this template. The fourth chapter demonstrates two applications of the template, these two applications have already been mentioned in the approach section. Chapter five shares the author's experience with documenting of software interfaces, gives a method for evaluating interface documentation and also shares the experiences with evaluation of interface documentation. The last part is the conclusion, where the author's opinion of documenting software interfaces and the conclusion for the whole report will be presented.

2. Definition, Importance, Stakeholders

2.1 Definition

What is software interfaces? There are several definitions, in [4] interface is defined as a set of services that specify all or part of the externally visible behavior of a component. One component can have several interfaces, and one interface can be implemented by different components. In [3], an interface is a boundary across which two independent entities meet and interact or communicate with each other.

Documenting software interfaces defines the interface between the caller and the component, it exposes the externally visible properties of the component, specifies the services provided by the component and any information which is related to the services.

2.2 Importance

Interface documentation is a part of architecture documentation, therefore, to understand the importance of interface documentation, as well as the importance of architecture documentation should be done at first.

Architecture documentation records the architects' decisions. It serves as the blueprint for a system and the project that develops this system. Architecture documentation tells developers how to make the sets of parts work together as a successful whole. Architecture documentation communicates the achievement of several engineering goals, such as performance, reliability, security, and modifiability. Architecture documentation can protect the investment, which is made during hours of design, and ultimately protect the intellectual property of a company.

Interface documentation is important because what it documents is exactly what the readers need to know about a component in order to use it in combination with other components. Besides, the interface documentation also provides a statement of other visible properties which can help the developer to build a component, test a component and integrate components.

2.3 Contents in One Interface Document

An interface is documented with an interface design document. There is a lot of information related to the interfaces, listing all possible pieces of information is not practical and almost impossible, so the writer should just expose only what the reader

need to know in order to understand the interface, generally, the contents in one interface documentation should at least include.

- *Introduction*. A brief description of the interface.
- *Services*. Including general and detailed description; it is the core of interface documentation.
- *External data types*. List all external visible data types.
- *Limitations and qualities*. The limitations before using the component and the qualities of the component.
- *Requires*. The requirements of the component.

The next chapter will give a detailed description for each item. Also how to document this information will be described in the next chapter.

2.4 Stakeholders of Interface Documentation

During the process of software development, different documents have different function, for example, in Software Requirement Document, there lists the software/hardware environment requirement of the whole software and each component, and there also lists the demand of customers for the software (including performance, quality, etc). The document of Software Architect Description describes the overall architecture of software, it mainly reflects the idea of the software architect, and it is the most important documents describing the software architecture. And, the document of Detailed Software Description acts on telling the developers how to implement each component. This documents describes the behavior of each component, from the implementers' point of view, this is the most important document.

Interface document describes the external properties of each component, listing the services of each component. It faces several different stakeholders. Figure 2.1 shows a possible combination of stakeholders and the relationship between them. Different companies or organizations will have different combination, such as maintainer, analyst and new member is perhaps all part of developer. The arrows in this figure indicate direction of information flow. For example, developers use interface documentation to provide information to analysts and analysts give the response or questions to developer directly or by using the interface documentation (analysts can list their questions or suggestions in the interface documentation). During the software interface development stage, the developers need to prepare interface documentation to the managers to get their necessarily support. On the other hand, the managers can keep informed of current arrangement, development, resource usage and achievements of the software interface via the interface documentation.

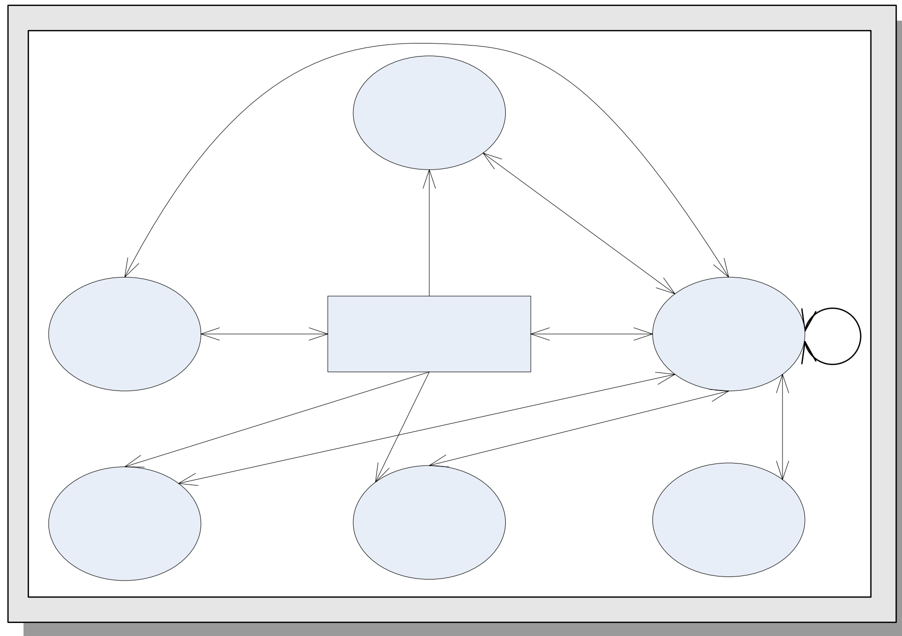


Figure 2.1 the functions of the interface documentation

The figure clearly shows that the analyst, new member, manager, and another project developer do not have the right to modify the interface directly (Of course, in some company or organization, if these stakeholders are included in developer group, then they have the right to modify the interface.). Also it can be seen from this figure that the software developers should supply adequate information for other stakeholders to help them understand the software interfaces.

Different stakeholders of interface documentation have different requirements and expectations. The stakeholders of interface documentation and their requirements of the information are listed below, including the applications of the interface documentation. The readers may notice that some sections of the interface document mentioned here will be described in detail in the next chapter.

1. ***Developer***. Here developers refer to the people who design and implement the software. They are also the writers of the software interface documentation. It is obvious that they are the core and one of the most important stakeholders of the interface documentation. Developers provide all the information of the interface that other stakeholders will depend on. Developers also need the most comprehensive documentation of an interface because during each stage, the documents always serve as a record of the achievements of the previous stage and the foundation for the next stage.

   Possible use scenarios for software interface documentation

   - *Design a component*. The developers need almost all the information of the interface. Later, when they document the interface, it is their duty to ensure that the information is correct because other stakeholders will see and

entirely depend on that information.

- *Use a component*. The developers need to know detailed information about the services provided by the component implementing the interface. This information is mainly focused on the semantic part because the developers need to know the effect of calling the services provided by the component. If the component requires some services from other components, the developers need to read the information in the requires section of the interface documentation.
- *Implement a component*. Implementers need the information in the services, external data types and error handling section of the interface documentation. If the component requires services from the caller, then the scope expands to the requirements section.
- *Integrate software system*. The developers integrate the discrete software components to a complete software system. If the developers can integrate without any errors, then almost no information is requested. However, if they face problems, there are two possibilities for the interface information. First, if the software system builders guarantee that every requirement (the general requirement is listed in the interface document, and the detailed is written in the requirement document) of the components is satisfied, the integrator should focus on the semantic part and try to find the bug. Second, if it is not confirmed that the interface itself is correctly implemented, then both the syntax and semantic part should be focused on.
- *Test a component*. Here the word test means a black-box test, that is, all the behaviors and implementations in the component are not visible, and therefore, the testers should focus on the external visible behavior of the component. The information requested by testers depends on the problems. For example, if the services of the component cannot be called successfully, then the interface document should give the information about the syntax part of the service. The testers need this information to check whether both sides of the interface (the caller and the provider) are syntactically correct or not.

2. ***Analyst***. The analysts' responsibility is to analyze if the functionality or quality of the interface is satisfied. As shown in Figure 2.1, the analysts can give some feedback to the developers if they find out that the functionality or quality cannot be satisfied. The information needed by the analysts depends on the types of the analyses, for example, for quality analysts, the interface documentation should provided enough information in the limitations and qualities section. A special analyst is the customer. Customers do not care about how the developer design or implement the software; they just want their requirements satisfied.

3. ***Maintainer***. The interface document is the starting point for maintenance activities. The information requested by the maintainers depends on what should be updated. For example, if the specification of the service's semantics was changed, obviously, the maintainers need to renew the semantic information in

the interface documentation.

4. ***New member***. The interface documentation serves as an education tool. The educational use consists of introducing people to the software system, including the interfaces. The people may be new members of the developers team, external analysts, or even a new architect.

5. ***Manager***. In [3] is explained that managers are also important stakeholders of interface documents. Managers use the interface documents for planning purposes. They can apply metrics to gauge the complexity and then infer estimates for how long it will take to develop a component that realizes the interface. Depending on the metrics, information might be required about the size of the interface and contained functionality but not on further details. Managers can also spot special expertise that may be required, and this will assist them in assigning the work to qualified personnel.

6. ***Another project developer***. Current software technologies (for example, objects oriented technology, formwork technology) make components reuse possible. To see whether a component can be reused or not, the developer is first interested in the general information (introduction and general service description section) of the component including that what services are provided by the component, and how about the capabilities of the services. If the component requires some services, the developers also need to read the requirements section of the interface document to see if the requirements can be satisfied in the new environment. When the developers continue to qualify the component if it is reusable or not, they will pay more attention to the detailed service description section and try to adapt the component to the new software project.

3. Template for Documenting Software Interfaces

Sections 3.1 to 3.8 describe the organization and content for a software interface document. For each section, it provides a description and an example.

Figure 3.1 shows the organization of a software interface document. Each box names a section of the document and briefly describes its purpose.

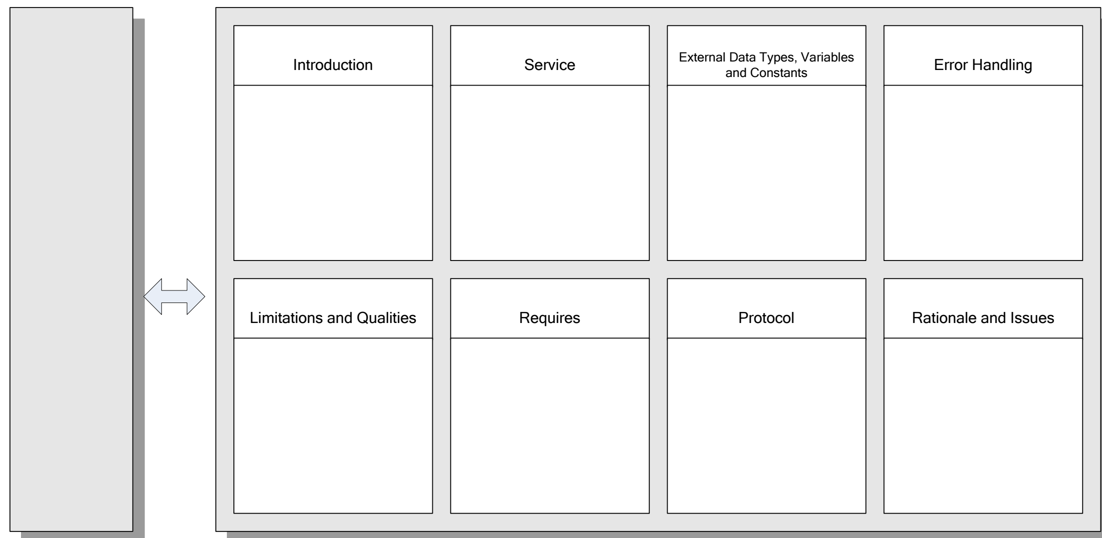


Figure 3.1 Organization of a software interface document

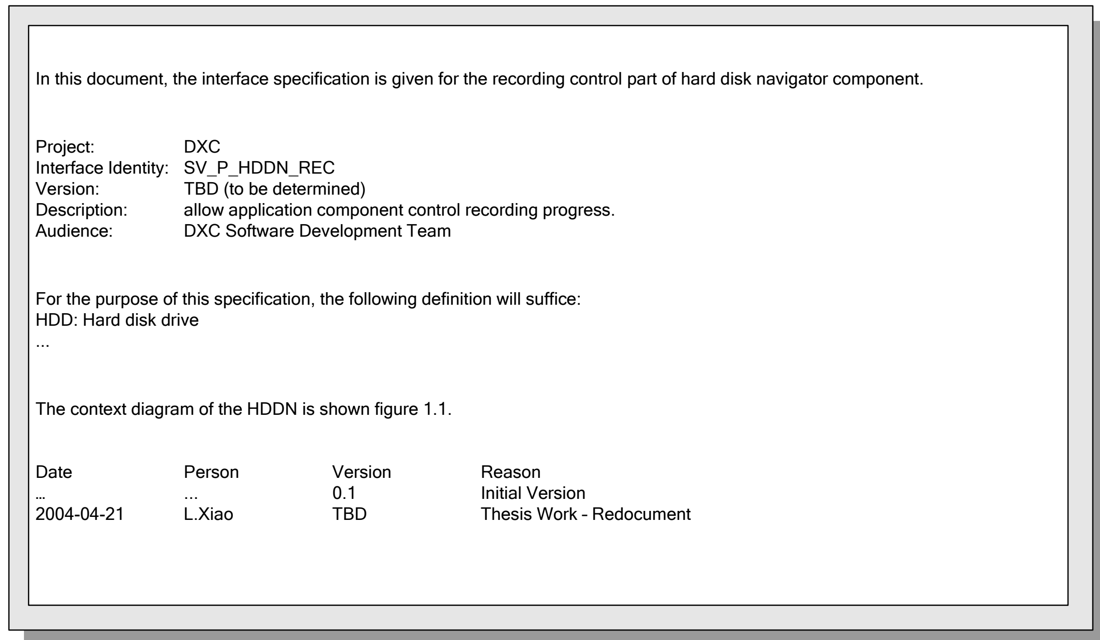
3.1 Introduction

The introduction gives a brief description of the interface, provides an identifier, presents the stakeholders, records the creation and modification dates, and exhibits a context diagram which shows the relations between the component/interface and other components/interfaces. Details to be captured include

- an identifier;
- a version number;
- the names of the design team and authors of the interface document;
- the creation and modification history, including the author's name and the modification reason;
- the intended audience; and
- any related documents.

When there is a single product associated with the interface document, this section can also contain information such as product name, release date and product team.

Figure 3.2 shows an example of the introduction section (excerpt one).



In this document, the interface specification is given for the recording control part of hard disk navigator component.

Project: DXC
Interface Identity: SV_P_HDDN_REC
Version: TBD (to be determined)
Description: allow application component control recording progress.
Audience: DXC Software Development Team

For the purpose of this specification, the following definition will suffice:
HDD: Hard disk drive
...

The context diagram of the HDDN is shown figure 1.1.

Date	Person	Version	Reason
...	...	0.1	Initial Version
2004-04-21	L.Xiao	TBD	Thesis Work - Redocument

Figure 3.2 Example SV_P_HDDN_REC: Introduction (excerpt)

3.2 Service

The heart of an interface document is the section in which the services that the component provides to other components are described. Services can be requested from a component in order to effect some behaviors, and are specified as functions. Each service function is documented by

- *General description*. A description of the service function at a very high level of abstraction.
- *Detailed description*. A detailed specification for the external properties of the service function.

Figure 3.3 shows the organization of the service section. In some cases, related events are necessary to be documented as an independent part in this section. The reason will be explained later.

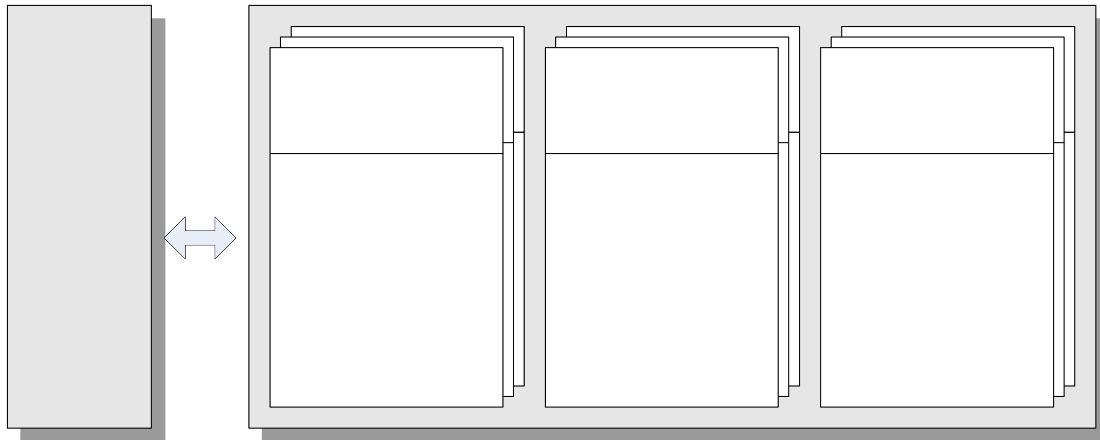


Figure 3.3 Organization of Service Section

Figure 3.4 Example SV_P_HDDN_REC: One of the general interface descriptions (start record)

The general description is aimed at providing an easy to understand description of an interface service function to the reader. The descriptions in this part may lack detail and precision as they use natural language. Figure 3.4 is an example of the general interface description.

The detailed description describes a function in a detailed and precise manner. It includes two parts: a syntax part and a semantics part. The syntax part includes the information that developers of other components need to write syntactically correct code that uses this service function. The syntax part includes

- service function name;
- purpose of the service function;
- parameter names;
- parameter types: I (Input), O (Output) and I-OPT (optional input);
- descriptions for the parameters;
- prototype; and
- function type (used in embedded system): asynchronous and synchronous.

The semantic part focuses on the descriptions of the external behaviors; that is what happens, when service functions are used and under what conditions they can be used. It includes the items below:

- *Requires*. Condition before interaction, if this item is empty, it means that this service function can be called under any condition using syntactically correct input..
- *Modifies*. Identifies any parameters that will be changed by executing this service function.
- *Effects*. Describes the external behavior of this service function and defines the result of invoking this service function. The result could be an assignment of values to variables, a change to the component's state, events generated by this function, and humanly observable results (used in embedded systems).
- *Description*. Lists the explanations in the detailed description.
- *Open Points*. Lists the problems that remain to be solved.

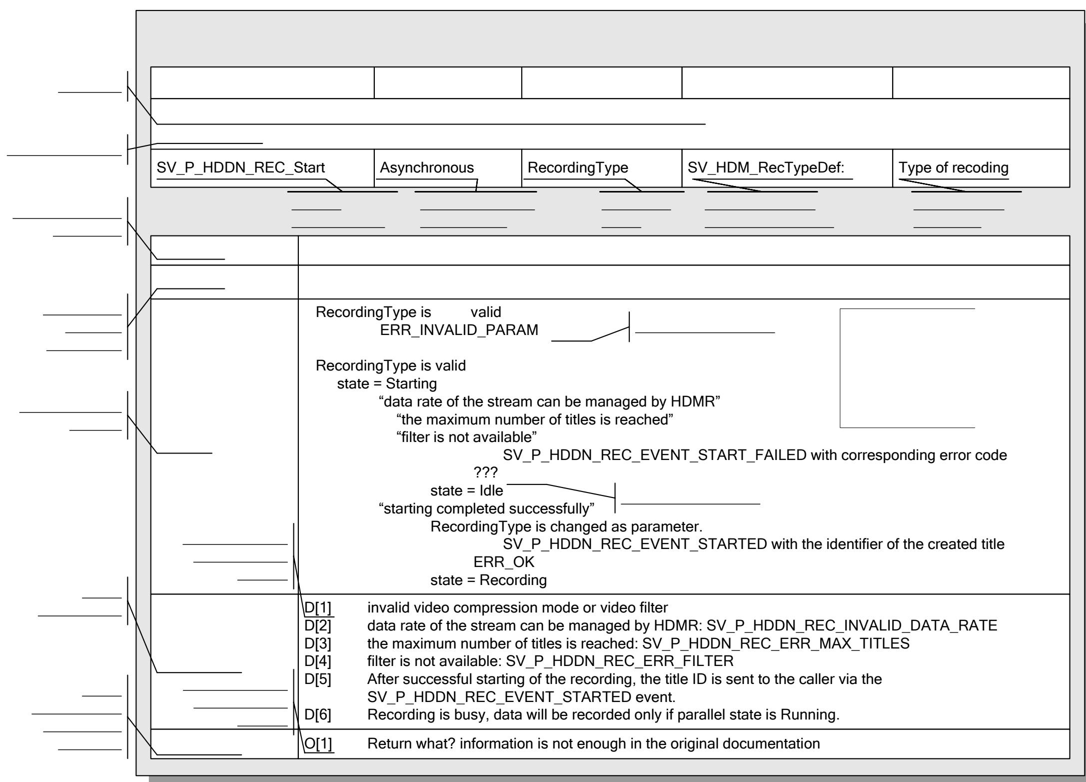


Figure 3.5 Example SV_P_HDDN_REC: One of the detailed interface descriptions (start record)

Figure 3.5 shows an example of the detailed interface description. It defines a template and gives some explanations in this template. Figure 3.6 shows an example of the events. Also it defines a template and explains each item in this template.

Event Name	SV_P_HDDN_EVENT_START_FAILED
Purpose	report the starting the recording is failed
Requires	SV_P_HDDN_REC_Start() failed
Effects	state transits to Idle, the reason of error will be reported to application.
Data	Error code, possible values: - SV_P_HDDN_REC_INVALID_DATA_RATE *(D[1])* - SV_P_HDDN_REC_ERR_MAX_TITLES *(D[2])* - SV_P_HDDN_REC_ERR_FILTER *(D[3])*
Description	D[1] SV_P_HDDN_REC_INVALID_DATA_RATE there is not enough data rate to start the recording D[2] SV_P_HDDN_REC_ERR_MAX_TITLES maximum number of titles reached D[3] SV_P_HDDN_REC_ERR_FILTER error occurred during starting of one of the filters

Condition before sending this event
Identifies the effects after sending or receiving this event
The data carried by the event
List of explanations
Description with corresponding number
Description [Number]

Figure 3.6 Example SV_P_HDDN_REC: One of the Events (start record failed)

Normally, an event is one part of the result of invoking a service function and should be documented in the semantics part, but in this thesis project, application components use callback functions to receive events from the HDDN component; direct documenting events in the semantics part is difficult and complicated. For example, if the event *SV_P_HDDN_EVENT_START_FAILED* is documented directly in the effects clause, the shortcomings include the following:

- The readers must pay attention to the description of the event when they try to understand the effect. Therefore, they cannot wholeheartedly achieve a thorough understanding of the effect clause.
- As the Figure 3.5 and Figure 3.6 show, there are some repetitions; those are the conditions before sending the event in Figure 3.5 and the data of the event in the Figure 3.6. Avoiding those repetitions is necessary because it will be helpful for the reader to concentrate on the description of the service function itself.
- Some functions return the generate the same events. Neither repeating the description of the event, nor just giving the description of the event where it appears first are good. The first is an unnecessary repetition; the second makes it hard for the reader to find the event information quickly.

How to avoid these shortcomings? One solution is to document the events separately as an independent part in the service section. The event part includes the following items:

- *Event Name*. Each event should have a unique name.
- *Purpose*. Describes the purpose of the event.
- *Requires*. Describes the condition before sending this event.
- *Effects*. Identifies the effects after sending or receiving this event.
- *Data*. Identifies the data carried by this event.
- *Descriptions*. Lists the explanations in the event specification.

3.3 External Data Types, Variables and Constants

This section documents the external data types, variables and constants. Figure 3.7

shows the organization of this section.

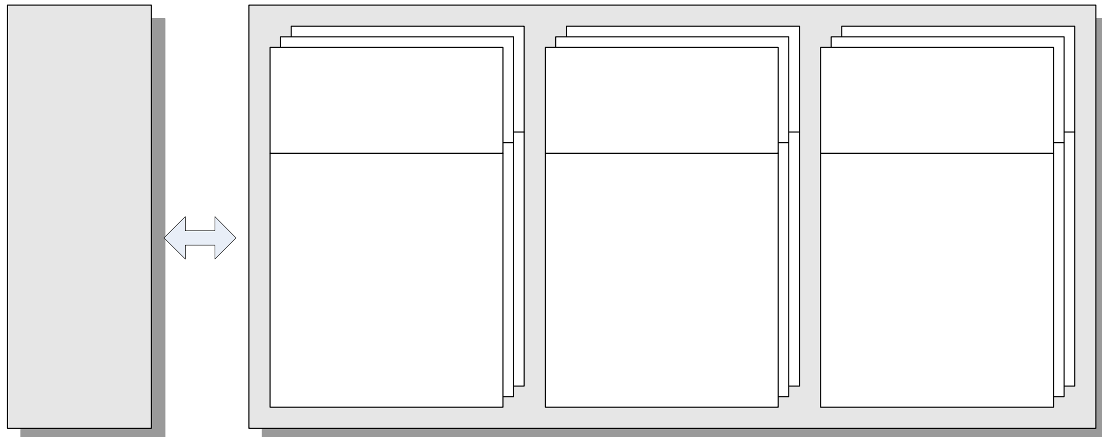


Figure 3.7 Organization of External Data Types, Variables and Constants Section

If any interface services use a data type other than the ones provided by the underlying programming languages, the callers need to know the definition of that data type. An example data type definition is given in Figure 3.8,

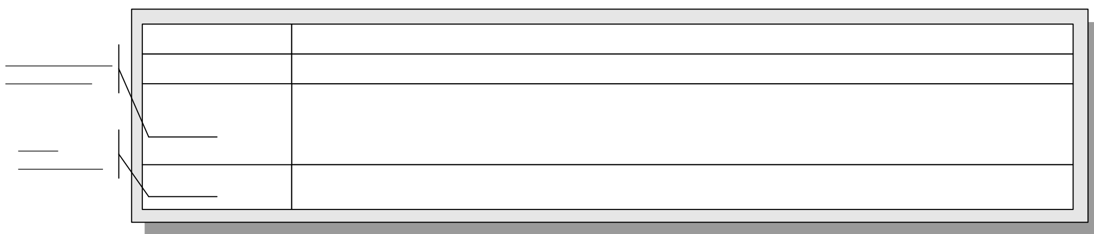


Figure 3.8 Example SV_P_HDDN_REC: One of the external data types (video filter)

The specification of a data type includes,

- *Type Name*. Each external data type should have a unique name.
- *Purpose*. The reason to define this data type.
- *Definition*. The definition of the data type.
- *Description*. Lists the explanations in the data type specification.

Some services use the data type defined by another component, such as the data type `SV_HDM_RecTypeDef` in Figure 3.5, a reference to the definition in that component's (`HDMR`) documentation is sufficient.

This section also documents the external variables and constants. The information includes:

- *Name*. The external variable or constant name.
- *Purpose*. The reason to define this external variable or constant.
- *Description*. List the explanations in the specification.
- *Value (optional)*. The value assigned to the constant.

3.4 Error Handling

Errors and the error conditions that can be raised by the services of the interface are specified in the error handling section. In most practical situations, the same error code might be raised by more than one service function under different error conditions and the error conditions have already been described in the semantic part, for instance, in Figure 3.5, the error code `ERR_INVALID_PARAM` is raised under the condition "`RecordingType is` **`NOT`** `valid`". To avoid unnecessary repetitions, this section simply lists the error conditions, and defines them in a dictionary named error code dictionary. The Figure 3.9 gives an example for the error code dictionary.

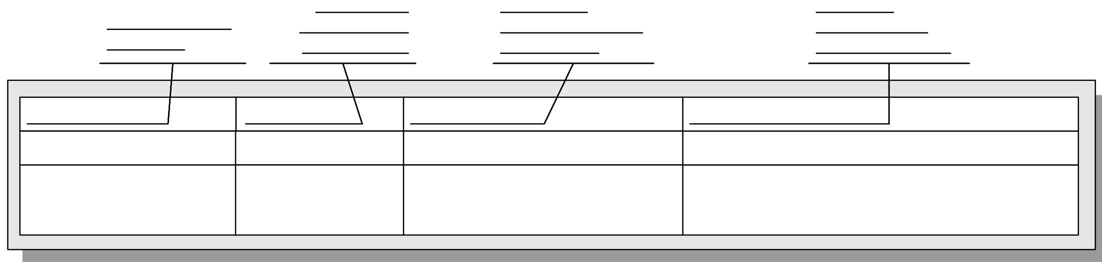


Figure 3.9 Example SV_P_HDDN_REC: the Error Dictionary (excerpt)

The information to be included in the error code dictionary includes:

- *Error code name*. Each error code should have a unique name.
- *Severity Level*. The effect when the error happened.
- *Error Condition*. Under what conditions, the error will be raised. Simply listing the conditions is sufficient.
- *Error Handling Behavior*. Define the common error-handling behavior.

The error code dictionary is the main body of the error handling section, beside it, it is necessary to describe the error handling mechanism. For example, what happens when a service function is called with parameters that make no sense? What happens when a function is called in the wrong state? Figure 3.10 shows an example of the description of the error handling mechanism.

The name of the error code

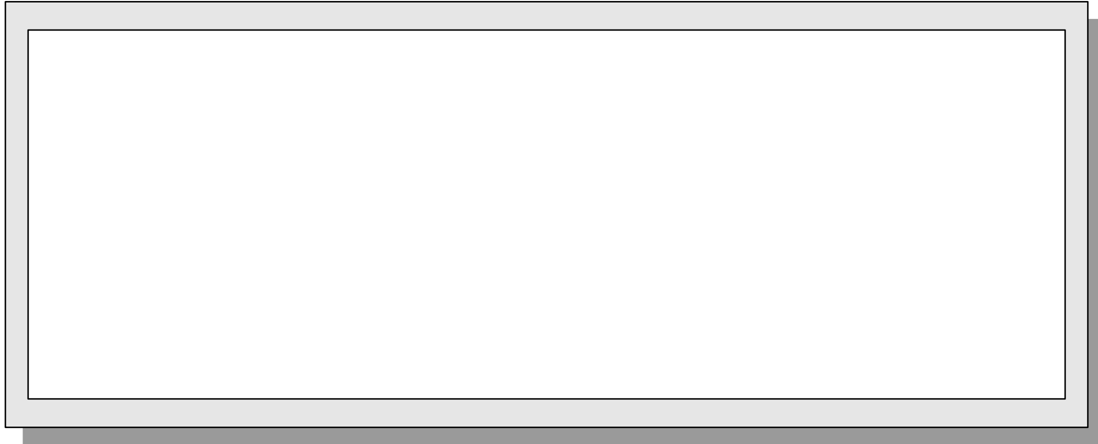


Figure 3.10 Example SV_P_HDDN_REC: the description of the error handling mechanism (excerpt)

As Figure 3.10 shows, in this thesis project, Philips designed two types of functions: synchronous and asynchronous functions. Both of them will return an error code as the result of the function. The difference is that asynchronous functions may send an event depending on the situation.

3.5 Limitations and Qualities

This section documents the limitations and qualities aspect.

- *Limitations*. Limitations are conditions on the interface services (for example, average throughput can not ever be less than 100 transaction/second; the service functions are non re-entrant, etc.).
- *Qualities*. The quality of service (for example, in embedded real time system, performance and throughput are extremely important, so the constraints of these elements should be declared in this section.).

The limitations and qualities mentioned here are summaries from the architecture requirements documentation, the full details of the meaning and measures should be left to the architecture requirements documentation.

Figure 3.11 shows an example,

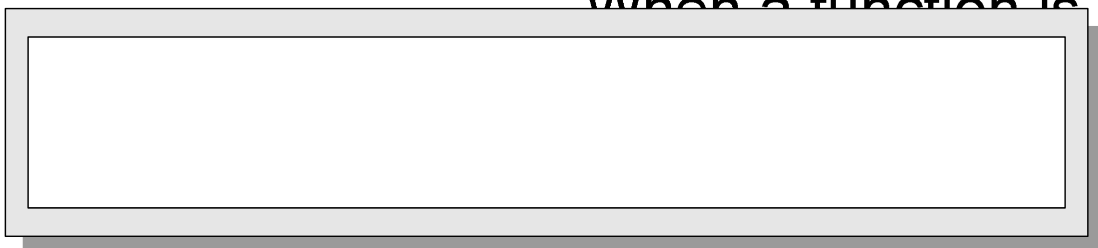


Figure 3.11 Example SV_P_HDDN_REC: the description of the Limitations and Qualities

3.6 Component Requires

If one component requires services, there must be another that provides them. The Component requires section specifies what the component requires. For example, there are three components, REC, APP, and PLAY. The current task is documenting the interface of the REC component; this interface provides some services RAS to the APP component. If REC needs some services PRS from PLAY to support the services RAS, these services PRS should be documented in this section.

The template in this section is similar to the service section. The difference is that this service belongs to other components and maybe the content is a set of assumptions (because maybe the project is not finished, the designer just list the service he wants.). For a developed project, referencing a single definition is enough, but for a project being developed, the information needs to be fully documented (often, this information is documented as a set of assumptions).

3.7 Protocol

A protocol defines all allowable sequences of services and interactions, and thus represents the complete behavior of the interactions between two components. Protocol is documented by,

- *General Description*. A description of the protocol at a very high level of abstraction.
- *Detailed Description*. A detailed specification for the protocol.

The structure of protocol section is shown in Figure 3.12.

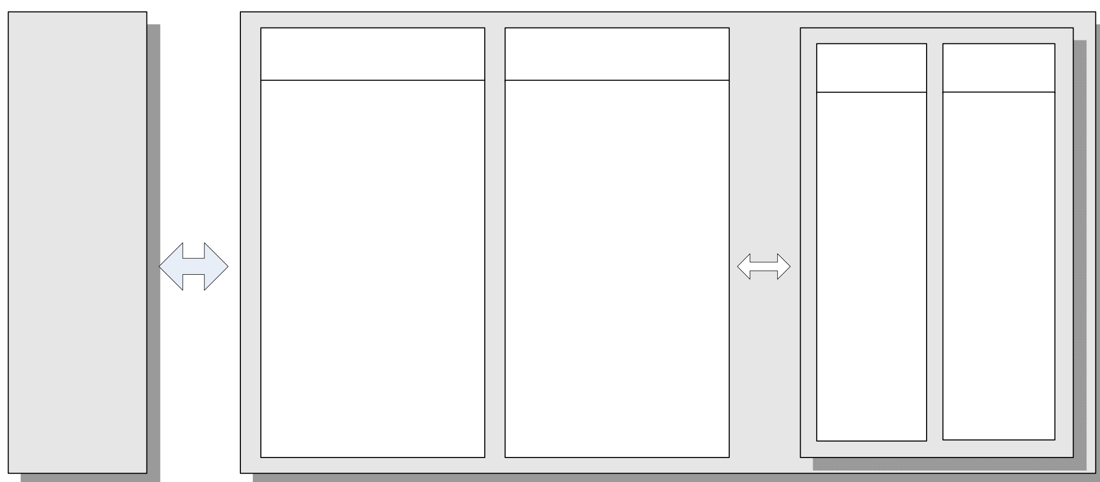


Figure 3.12 Organization of Protocol Section

An example of the general description and the detailed description are shown in Figure 3.13 and Figure 3.14.

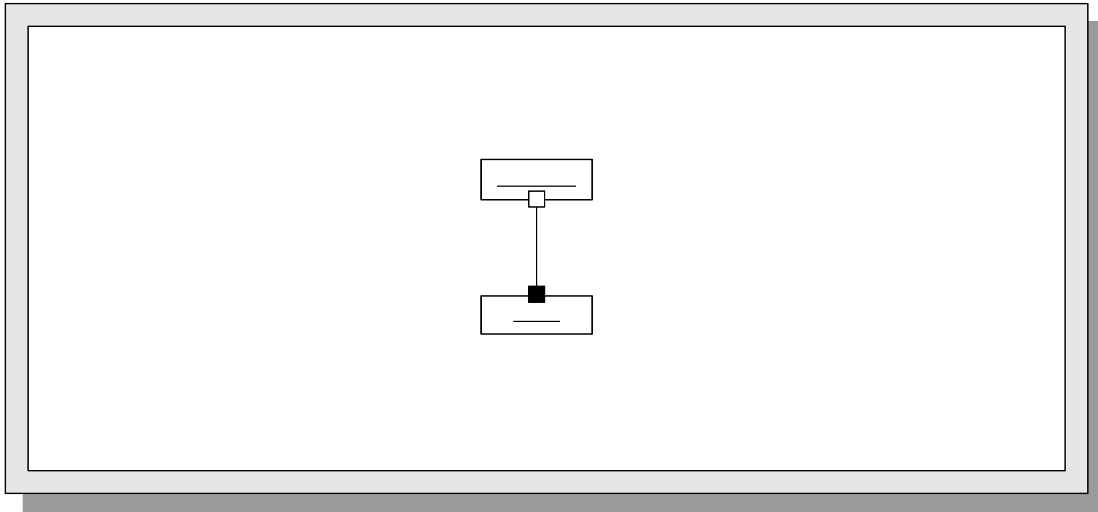
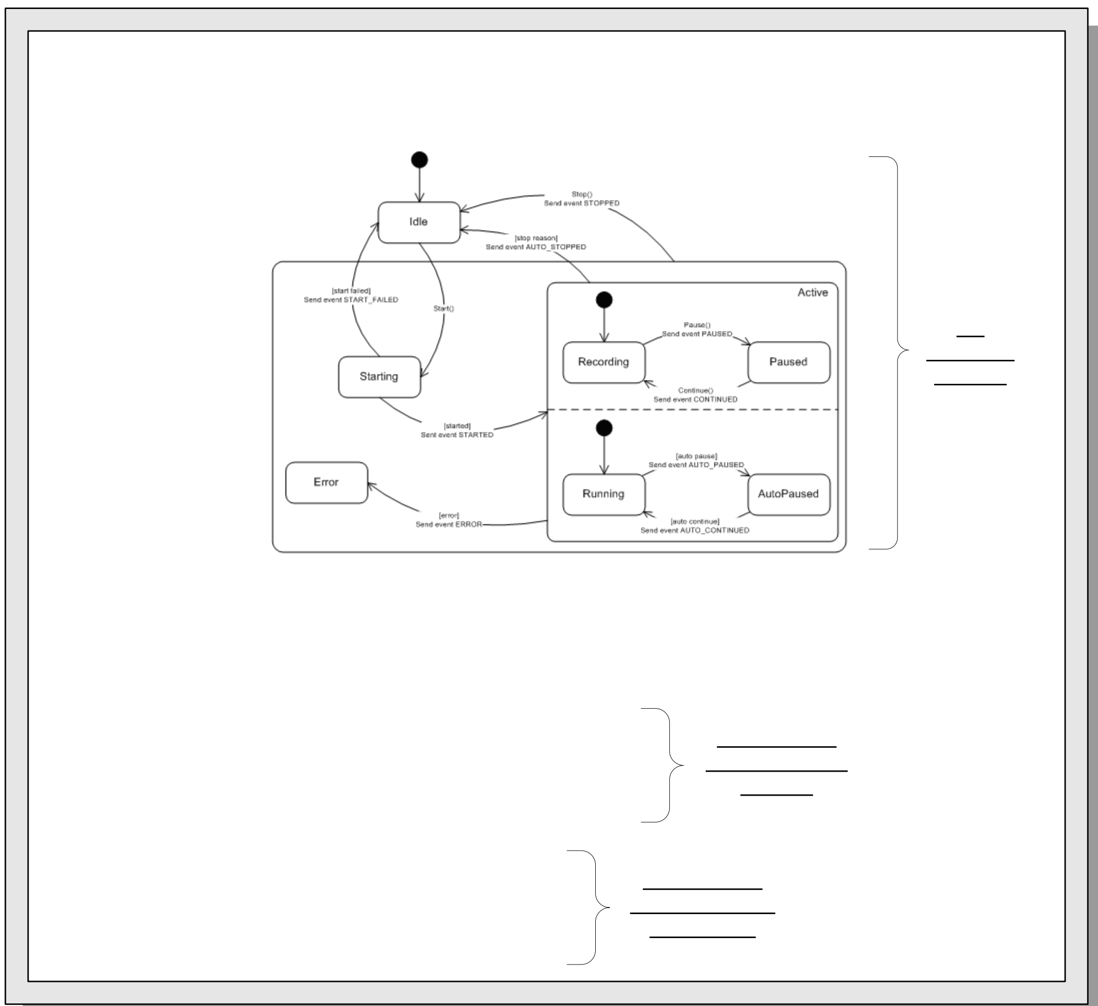


Figure 3.13 Example SV_P_HDDN_REC: the general description of the protocol (excerpt)

As Figure 3.13 shows, the general description of the protocol just gives a general description for the protocol and lists the purpose of the interface protocol.

General Description



s the HD
N_REC c
s applicat

Figure 3.14 Example SV_P_HDDN_REC: the detailed description of the protocol (excerpt)

The detailed description includes,

- *Statechart diagram*. A statechart diagram shows the possible states of the component and transitions that cause a change in state. For a simple protocol, a statechart diagram is enough, but for a complex protocol, it is necessary to document some sub-state statechart diagrams.
- *Description of the diagram*. Describe the purpose, the state and the conditions of the diagram.

3.8 Rationale and Issues

The last section documents the rationale and summaries the unsolved problems.

The rationale includes,

- the reason why the architects choose the current design;
- constraints and compromises;
- the alternatives the architect rejects and states the reason;
- the future development plan about this interface; and
- the insight the architect has about how to change the interface in the future.

The issues part should record the problems that cannot be solved now and includes,

- the open points which were already mentioned in the documentation; and
- the problems which were found during the review of the documentation.

Figure 3.15 contains an example for issues part.

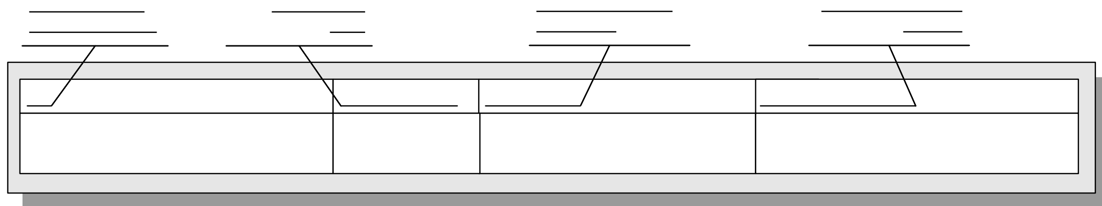


Figure 3.15 Example SV_P_HDDN_REC: the issues

The information to be included in the issues table includes:

- *Open point ID or name*. Each open point should have a unique ID or name.
- *Severity Level*. The severity level of this open point.
- *Description*. Description of this open point.
- *Possible Solution*. Given the possible solution, if none, just leave it blank.

4. Application of the Template

The template for documenting software interfaces has been used to varying degrees on different development projects. The purpose of these applications is to improve the quality of the existing interface description documentation in order to help people understand the software interfaces. In the next section two applications of the software interface template on two projects at Philips PDSL are presented.

4.1 Super Audio CD player

The Philips SACD1000 is a multi-channel Super Audio CD (SACD) player that also plays both PAL and NTSC DVD-Video discs, CDs, Video CDs and even CD-Rs. SACD is a development by Philips as a successor to the Compact Disc (and competitor to DVD-Audio). It provides superior sound reproduction quality, with the ability to render all the nuances of a live performance (for details, see www.superaudiocd.philips.com).

One of the interface design documents was redocumented using the template described in chapter 1. This document describes the software interface specification of the loader services component of the SACD1000 player. Through the interface, the loader service component provides necessary services to the application components of the SACD1000 player. These services include disc and area (area is a part of a disc that contains the data for a single or parts of an application) identification, area switching, tray control, etc. With these services, the loader services component will be used by application components.

After recapturing the relevant interface design information, the redocumenting process started. The information in the original document is accurate (almost no erroneous information, no inconsistent information, and no missing information) and easy to understand. Additionally, the information is well organized, so the original document is not too difficult to navigate. But there are still some problems in the original document:

- Some information in this document is not related to the interface itself. For example, in the required functionality section in the context part (the content is similar to the component requires section in the template which is mentioned in chapter 3), not only the services provided by application components but also those provided by the components which connect to the loader services component are documented. In fact, only the services provided by the application components and required by the loader service component should be documented in this interface document.
- There are some unnecessary repetitions. For example, in the detailed interface specification part (similar to the detailed service section in the

template), a *Return* column listing what the result will be returned is included in each function description. However, the contents in the *Return* column are repeated again in the *Postcondition* column which is next to *Return* column.
- The contents of the *Postcondition* column of the detailed interface descriptions are written in a formal language. It is accurate but it is neither always practical nor necessary, because not all readers understand the formal language (especially for new project members who are attempting to understand the document), and in most cases, informal language is also qualified to describe the interface.

To solve the problems above, the main activities for redocumenting this interface document are,
- adjust part of the organization of the original document, remove the unnecessary part in the old organization;
- adopt a informal language to rewrite the detailed interface description, ;
- remove the unnecessary repetition and; and
- remove the information which is not related to the interface itself.

The original document is 38 pages and the redocumented document is 34 pages. The benefits of the new interface document are,
- size reduction, from 38 pages to 34 pages; and
- the detailed interface descriptions are easier to understand by using a informal language.

4.2 DVD Recorder

The second project is a DVD hard drive recorder. The most natural way to use a DVD recorder is like using a VCR. Most of the DVD recorders record the target programs directly on recordable DVD discs, while the DVD hard drive recorders can also write to hard drive. Obviously, the latter one offers flexibility by saving the contents to a hard disc. With a hard drive in a recorder, people do not have to worry about finding suitable discs to record day-to-day programs. The DVD hard drive recorders work much like the drives used in personal computers. The amount of broadcasted programs that can be stored in the hard drive unit depends on the size of the storage space and the recording mode.

In this application, two interface specifications were redocumented. Both of these interfaces belong to the HDDN (hard drive navigator) component of the DVD hard drive recorder. One interface provides services for the recording control part, including start/stop/pause/continue record, set/get recording mode, and set/get low space warning. The other provides services for the playback control part, including start/stop/repeat playback, set/get playback mode, and change play position. All the services are provided to application components so that the application components can control the HDDN component.

Unlike the first project, there are still some errors and inconsistencies in the interface document, so it is more difficult to redocument. Consequently, the information related with the interfaces should be captured carefully. The main problems in these two interface documents are listed below,

- In the original document, the two interfaces were combined into one document.
- The organization of the original documents is a little bit disordered, for example, the detailed interface description are written in the appendix part, it is better that writing the detailed interface description in the main body of the document.
- Some information is not related to the interfaces, for example, in the original documents, there is a lot of background information.
- There is some wrong information in the original document.
- Some information is insufficiently documented, for example, the rationale, the description of severity level, etc.
- Some information is inconsistent in the original document.
- The general interface descriptions of each service function are too detailed. Some of this content should to be described in the detailed interface description.
- The detailed interface descriptions and external data types are written in program code and the details of the functions and types are missing, for example, there is no *Effect* description. In fact, the original document only provided a C-head file to describe the interface and only the syntax are offered.

For redocumenting the following process was followed:

- recapture the relevant information;
- find and correct the wrong, inconsistent, and missing information;
- split the original document into two parts, one describing the record control interface, the other describing the playback control interface;
- the general interface descriptions are rewritten, the information which is not related to the interface is removed;
- the source code in the appendix is replaced by the detailed interface descriptions, the descriptions of events and the external data types;
- add additional necessary information such as *Effect* to describe the service functions, events and data types;
- add a error code dictionary and the corresponding descriptions;
- remove the irrelevant information in the protocol section; and
- add a new chapter to record rationale and issues. However, the required information for this new chapter is still insufficient.

The original document is 66 pages. After redocumenting, the record interface document is 33 pages, and the playback interface document is 37 pages. The main

improvements of the new documents are,

- the organization of the new documents is easier for the reader to navigate;
- irrelevant information is removed;
- the wrong, inconsistent and missing information is found and is mostly corrected; and
- the detailed interface descriptions and the data type descriptions are easier for the readers to understand because the descriptions are precise and detailed.

4.3 Application Evaluation

To evaluate the new documents, four metrics are considered. They are,

- *Inconsistency amount*. Number of inconsistencies (old / new design description).
- *Error amoun*t. Number of errors (old / new design description).
- *Size*. Number of pages (old / new design description).
- *Missing information*. Number of missing information elements (old / new design description).

The next chapter will give detailed explanations for these four metrics. This chapter just presents the result of the evaluation.

Figure 4.1, Figure 4.2 and Figure 4.3 show the results of the evaluations of the redocumented documentation for the SACD1000, DVD recorder record, and DVD recorder playback interface respectively.

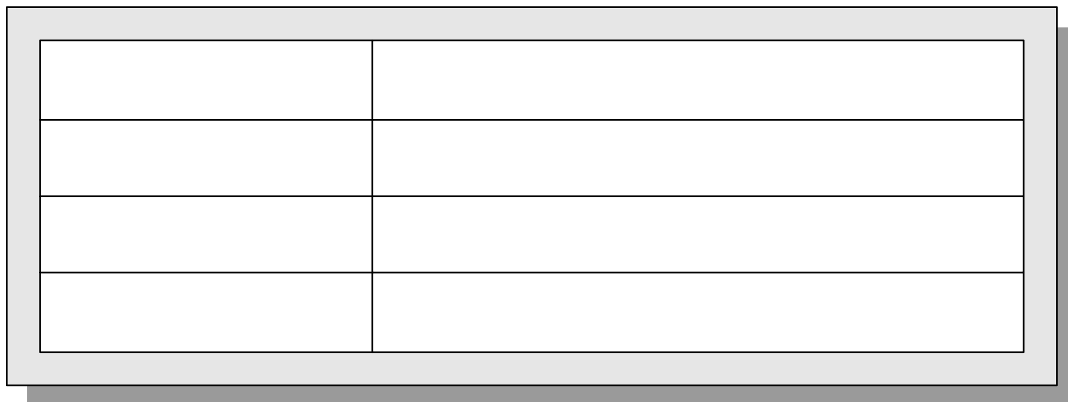


Figure 4.1 the evaluation result of the redocument documentation of SACD1000

The document of the first application SACD1000 is well organized and the information in the original document is accurate and easy to understand, so improvement is not obvious. Only the detailed service descriptions are rewritten and the unnecessary information is deleted, as the result, the size of the document is reduced.

In the second application, some pieces of information in the original document are

shared by both of these two interfaces, because record interface and playback interface were documented as one document. However, these two interfaces should be documented as two independent documents because both of them are so complex that they should not be documented as one document. After the original document is split into two documents (one document describes record interface and one describes playback interface), the total amount of pages is increased to 88 pages. The record interface document is 40 pages and the playback interface document is 44 pages.

Figure 4.2 the evaluation result of the redocument documentation of DVD recorder (record interface)

Figure 4.3 the evaluation result of the redocumented interface documentation of the DVD recorder (playback interface)

Figure 4.2 and Figure 4.3 clearly show that the sizes of the original documents are reduced. Also the amount of inconsistencies, missing information, and errors are reduced.

4.4 Evaluation Conclusion

Size Reduction

The three evaluation results mentioned in this chapter demonstrate that the template designed in this thesis project is a valuable tool for documenting software interfaces. It is a tool that can help the engineer to document the information related to the interface in a structured way, and therefore help the engineer to produce good quality interface documentation (the size of the interface document can be reduced and part

Inconsistency Reducti

of the missing information can be found by using this template). Besides this, the template is also able to describe different software interfaces in these two different projects. However, for finding and reducing error, inconsistency and part of missing information, the template designed in this thesis project cannot provide effective method, that is, finding and reducing error, inconsistency and part of missing information still mainly depends on the experience of the people who document or evaluate the document, not depend on the template.

5. Experiences with Software Interface Documents

This chapter aims to share the experiences of the author with documenting software interfaces. Two aspects of experiences are described, the experiences with documenting and the experiences with evaluating. In order to explain the experiences well, the first thing that should be explained is what a good interface document is.

5.1 Attributes of Good Interface Documentation

Although each good interface documents can have different forms, they share some generic attributes. Four important attributes: accuracy, clarity, integrity and maintainability, are described here.

5.1.1 Accuracy

The content of a software interface document should be accurate. Ambiguous description should be avoided. The descriptions of a certain subject in different parts of the document should be in accordance, i.e. without contradiction.

5.1.2 Clarity

The document writing should be simple and clear. If possible, accompanied with proper tables, charts and figures to improve the document's clarity.

5.1.3 Integrity

Every document should be integrated and independent. For example, in the introduction part, the general introduction of the document should be given; in the main text part, the core content should be given; and when necessary appendix parts should also be included, the appendix part should list the reference (or list the reference in introduction section). Some parts of a certain subject could be the same in different documents. However, this duplication can be necessary. Especially, in a document, quoting another document’s words should be avoided, such as just mention “see chapter *.*.*”. This will offer much inconvenience to the readers.

5.1.4 Maintainability

Different software projects could have many practical differences in their scale and complexity, but most software will be maintained or developed in the future. Documentation should be updated to reflect the progress, it requires the documents to

be easy to review and maintain.

5.2 Experiences with Documenting Software Interfaces

5.2.1 Documenting from the Reader's Point of View

Almost all literature about documenting software architecture (such as [3] and [4]) ranks this rule as the most important one. The reason is obvious: every document will ultimately face to readers. A clear distinction of the readers should be drawn before documenting. As mentioned in chapter 2, different types of stakeholders will have different requirements for the document. In some cases, an interface is documented as different documents in order to satisfy the different requirements of the stakeholders, at this time, the writer should pay more attention at the distinction of the readers, for example, the documents prepared for managers should not use too many software scientific words as can be used in a document faced to developers.

5.2.2 Focus on the Properties Belonging to the Interface

An interface document should focus only on the interface which is being documented. The information belonging to other interfaces should not be introduced. Irrelative or unnecessary information should not be included in an interface document. During the documenting process, one of the most common mistakes during documenting is that the writer might document some contents about how components are implemented in an interface document. Although documenting how to implement is important, it is not the scope of the interface document. Another common mistake is to introduce background of the software systems. However, this content is not necessary to be included in interface documentation, neither. Because in most cases, the readers of the interface document are only interested in properties belong to the interface, and if they want to know more information about the background of the software systems, they can read the document of the software architecture description directly. Besides it, in a complex software system, there will be hundreds of interfaces, documenting the background of the software system in every interface document is not practical.

So what should be documented in the software interface documentation? As [3] defined, an interface is a boundary across which two independent components meet and interact or communicate with each other. This definition obviously tells the author that documenting software interfaces is documenting the "boundary", that is, the external properties of a component. In order to distinguish whether a property is external or internal, the writer should consider it from the caller's point of view and consider the component which provides the interface as a black box (of course, everything inside the black box is invisible). Now what can be seen or obtained by the

caller are the properties belonging to the interface. The properties are listed below,

- *Identifier*. The name of the interface.
- *Services*. The service functions that can be used by the caller.
- *Data types, variables and constants*. The data types, variables and constants are defined by the component and are externally visible by the caller.
- *Components requirements*. What the component requires from the caller.
- *Limitation*. All limitations of the interface.
- *Protocol*: All allowable sequences of services.

Here is a tip: if the information is indispensable for the caller to successfully interact with the component, then this information should be included in the interface document. For example, a caller A request a service S of the component B and this service are provided through the interface IB, at first, caller A would have to know the name of the interface(Identifier), that is, IB, because there is more then one component in the software system, and each component has several interfaces. After obtaining the name of the interface, the caller A should acquire the name of the service (function name) and how to call it (syntax part). The caller A also needs to know the effect of calling this service (semantic part). If there is any data type defined by component B itself (external data types), then the caller A should know the definition of the data type. Other questions are: is there any limitation when using this service (limitations)? Does the service S require any service from the caller A or other component (requires)? Should caller A request any other service before use the service S (protocol)?

5.2.3 Use a Standard Template to Create an Interface Document

A standard template includes completeness rules for the information, for instance, on what aspects will be documented in the service section of the document and how to document the service section. As a bridge connecting the writer and the reader, the template helps the writer to document the information in a useful and for the reader easy to understand way.

Using a standard template to create an interface document has benefits to both the writers and the readers.

- For the writers, using a standard template helps them to plan and organize the contents, realize what should be documented, and reveal what work still remains to be done. In the review or maintenance process, the template helps them to find the erroneous, inconsistent and missing information quickly as well as to modify the specific information easily.
- For the readers, an interface document created by a standard template helps them to navigate the document easily, and access the specific contents quickly.

5.2.4 Find a Balance Point Between Too Little Information and Too Much

Documenting an interface is a matter of striking a balance between including too little information and too much.

The most concise document may not always be the best one because,

- Although too little information enhances the conciseness of the document, it will prevent readers from completely understanding the document;
- Too little information is neither sufficient nor detailed enough, it will barrier the document to serve as a blueprint for construction.

In order to make the document readily comprehensible, appropriate size increasing of a document by adding sufficient information is necessary. However, it is also important to avoid unnecessary redundancies or repetitions because,

- Too much is more likely to contain errors.
- Too much is less likely to be carefully read by the readers.
- Too much is more likely to be misunderstood by the readers.
- Too much will cost the reader much more time. Duplicate information even in slightly different forms will still confuse the readers. It will cost unnecessary time for the readers to understand why a different way was used to express the same thing.
- Too much will also bring inconvenience to the writer in further modification. A long widespread document is difficult to edit.

Therefore, a balance point between too little information and too much should be discovered. Figure 5.1 shows the relation of information and time. It also shows where the balance point is. The time axis represents how much time is needed to understand the information provided by the document, while the information axis represents the information amount provided by the document.

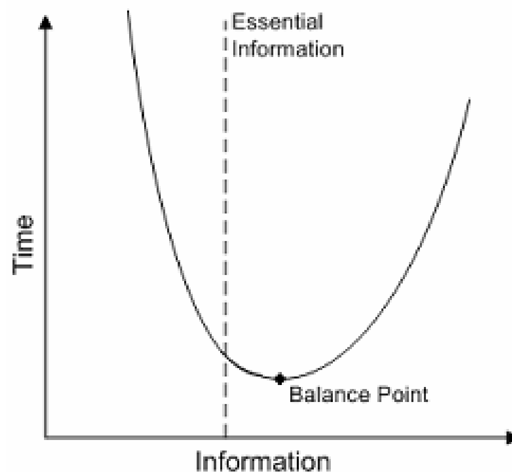


Figure 5.1 Balance Point of Too Little Information and Too Much Information

As shown in Figure 5.1, if the document does not provide the essential information, it will be very difficult to understand (It will cost the reader more time to understand the information, and sometimes, it is impossible to understand the document if the document cannot provide enough information.). After providing the essential information, adding some extra information such as some redundancy may help the reader to comprehend the content, but it is also possible to cost the reader more time. Hence, redundancy and repetition should not be introduced without a good reason and every redundancy and repetition should be on the right place.

Finding a balance point is mainly based on the experiences of the writer. No matter how hard it is, the effort on finding the balance point is aiming to make the document easier to understand and modify.

5.2.5 Suggestions for Documenting Service Section

As the most important part of the interface document, it is worthy to discuss experience with it in a separate section. Some suggestions for writing a good service section are,

- Only write down the properties which are visible by the caller. This rule is so important that it was mentioned before (see 5.2.2).
- Only write down the properties which are needed by the callers. Documenting each aspect of each service is not always practical. The interface document should expose only what the callers need to know in order to request the services of the component efficiently.
- Do not use implementations/source code to describe the effect of the services.
    1. Using implementation will limit the inspiration and the selection of programming tools for the implementers.
    2. The contents will be very difficult for some readers to understand. For same function, different implementer has different method to realize, and maybe the method is complex that only part of the readers can understand. However, the reader should focus on the effect itself, not the implementation.
    3. Modification of the service specifications will be difficult.
    4. The implementation cannot provide enough details in most cases.
- If the writer hesitates whether the readers can understand a formal description language (for example, IDL, Interface Definition Language) or not, he should not use the formal language. Service specifications written in formal language are not always practical. They are useful and precise. However, not all readers understand them.
- Avoid using prose style in describing the service specifications. Prose is written in ordinary language lacking precision. In [3] is mentioned that if it is necessary for the writer to use prose, then be as precise as possible, especially the verbs. For each verb, the writer must ensure that he really

understands what the exact meaning of the verb is and the readers should be able to verify the meaning of the verb.

- Do not only use examples to describe the effect of the service. An example defines the semantics of a service for only the single case illustrated by this example. The reader might make a good guess from this example and some inferences from his guess. However, all of the inferences are from his guessing. Building software systems cannot be based on guesswork. So, examples may lead the readers into a wrong thinking direction.
- If the writer is not sure whether a piece of information is correct or not, or a piece of information should be included in the interface documentation or not, then it should be documented in the *Open Point* column (in most cases, just write TBD - to be determined). Document this information rather than leaving the place in blank. The decisions can be made later, but if presenting a blank space, the reader will wonder whether the information is coming or whether a mistake was made.
- Always supply abundant explanations to the reader. This is better than lacking some explanations the reader needs. The explanations should be documented in the *Description* column. Writing the explanations in the *Effect* column will puzzle the readers: What is the difference? What is the meaning of the difference? When write them in *Description* column, the readers can judge if they need read these explanations.
- Ensure that the general interface descriptions and the detailed interface descriptions are consistent. Ensure that the content in the service section are consistent with the other sections in an interface document.
- For embedded real time system, if the effect can be observed by people, then write it in the *effect* column. For example, when the service "open the CD tray" is called, the tray will be ejected as the observable result. This should be written into the document.

5.2.6 Record Rationale

The rationale is an important ingredient in the interface documentation. Recording rationale is documenting the reasoning behind design decisions, for example, why to choose this option, are there any alternative solutions and why deny those alternatives? If there are any future design plans, the plans can also be documented in the rationale part. Later, when the decision has to be changed for some reasons, it is possible to use the alternatives.

Hundreds or even more decisions were taken during the software architecture design process. To document all of them is obviously infeasible. So which of them should be documented? Although the author does not have any practical experiences about documenting rationale because there is no rationale in the original interface documents, [3] provides some advises on when to choose to capture the rationale.

- The design team spent a significant long time on evaluating options before

making a decision.
- The decision is critical to the achievement of a requirement or goal.
- The decision seems not to make sense at the first blush but becomes clear when more background is considered.
- On several occasions, people have asked, "why did you do that?"
- The content is confusing to new members.
- The decision has a widespread effect that will be difficult to undo.

5.2.7 Update Document in Time

What do the Philips software engineers expect from the design documents and how do they use the design documents? The following is the response from Philips,
- The software designers use the documentation as a basis for their code.
- The documentation is leading in analyzing the consequences for changes.
- If the final code is ready, the design documentation should be leading in the maintenance phase.

These answers demonstrate that a document should be updated timely. Otherwise, the incomplete or out of date documentation could not reflect the truth and therefore it is impossible to finish the whole software development successfully.

But should the documents be kept meticulously up to date? Actually, it is unnecessary. Here are three reasons,
- Design document is a set of high-level abstractions so that any tiny changes will not affect the abstractions.
- Frequently updating the document is strenuous work. It will cost much budget and time of the software project team.
- Some decisions are possibly made and reconsidered with great frequency. Documenting them meticulously will be a repetition.

So when to update the documents?
- A significant decision has been made.
- A decision has been fixed.
- List the update schedule, and update the document following the schedule.

The updated document should be attached a new version number and the update reason should also be documented.

5.3 Experiences with Evaluating Interface Documents

After redocumenting the three documents mentioned in the chapter 4. The author has also evaluated the redocumented documents. The results of the evaluation have already been presented in chapter 4.

5.3.1 Proposed Metrics

As mentioned before, a good document should satisfied four attributes, accuracy, clarity, integrity and maintainability.

However, these attributes are too abstract to be measured directly when evaluating a document. Therefore, several characteristics which have some relations with these four attributes are proposed as metrics. Furthermore, the nine characteristics described in Figure 5.2 are easier to measure.

Figure 5.2 shows the relations.

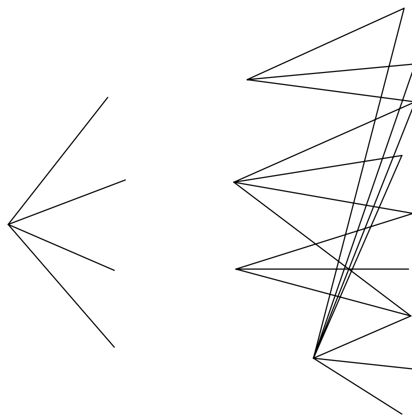


Figure 5.2 the Relation of Quality Attributes and Proposed Metrics

The following is the explanations for each item. The measure methods also will be introduced.

Inconsistency Amount
The contents in one document should be consistent. There are some examples of inconsistency, for instance, in the error handling part of the interface document in DVD recorder (the second application mentioned in chapter 4), the reader was told that asynchronous functions will raise a protocol error event with an error value if called in a wrong state. But in the detailed specification part, this point is not described. The reader will not know which one is correct, then to implement some functions well is impossible.

Measuring inconsistency value is simple; just count how many times an inconsistency occurs in one document. Of course, less is better.

Error Amount
In practical situations, it is difficult to avoid errors. Spelling mistakes, value mistakes, a wrong function description, and an incorrect architecture design, all of these we can

call error, but their difference is also obvious. Based on the severity, a weight will be assigned to measure the error value. For example, the weight of a spelling mistake is 1, and the weight of architecture design errors is 10 (in this thesis project, the weight are ignored and only the number of the errors are counted.).

Size

Document size is a necessary element when evaluating the quality of documentation. People always like a small-sized document. However, too concise can be as bad for quality as too copious. Too concise documents will lose essential information and too copious will cost the reader a lot of time.

Because size has is related to *Missing Information* and *Details,* here only the pages of one document will be counted. This value should be given a small weight because information is more important than size (in this thesis project, the weight are ignored and only the number of pages are counted.).

Missing Information

Documentation should provide enough information to serve as a basis for analysis. A document, in which information is missing, is certainly not a good document. To measure the amount of missing information, just count how much information is missing in one document. Normally, if people cannot understand or feel confused about documentation, the reason is some information is missing or not enough details are provided.

Easy Understanding

Documents should be easy for the reader to understand. To judge whether a document is easy for understanding, we can sum up the time when trying to understand specific information or all the information in one document. To measure easy understanding, a questionnaire should be provided to several readers. This questionnaire lists the question, that is, "how much time do you need to understand this document?" Based on the replies of the readers, the people can determine whether a document is easy for understanding.

Quick Access

A good document helps the reader to navigate the document and find specific information quickly. Quick access is also an important element when judging if documents can be easily maintained, because errors may occur during the design, testing, or even implementing process and maybe the reason is because of the document. To judge whether a document is easy for navigating, we can sum up the time required to find specific content in a document.

Details

Documents should be sufficiently detailed to serve as a blueprint for construction. But too many details are also harmful for the quality of the document. In ideal situation, a

balance point should be found (as mentioned in 5.2.4) in order to measure the degree of details. The value is calculated as 0 when the document lies in the balance point, the value will be increased when the document has too much or too less details. In practical cases, measuring details is the most difficult one in these nine metrics. It is mainly based on the personal experiences of the readers. The method is similar with the method used in easy understanding, and the question is changed as “Do you think if this document provides enough details?”

Impact Analysis
During the maintenance and development process, documentation needs to be modified. Estimating the impact of modifying the documentation is important, it is helpful for decreasing the maintenance time and reducing the probability of errors. As a simple example: it is possible that a data type is modified several times during the software development; and this data type must have some relations with other data types and the service functions. The documentation should clearly show the effect after modifying the data type. That means: the documentation should tell the writer what should be adjusted, e.g. the details specification of the functions?

Revision
During the design and maintenance process, engineers need to revise documents often. Documentation with good structure can be revised without difficulties. The measure method is also similar with the method used in easy understanding, and this time the question is changed as “how much do you need to revise the document?”

Through evaluating these nine items, we can compare two documents and determine which one has the better quality.

Concerning the real situation (There is only one people to evaluate the documents, and therefore the questionnaire method is not feasible. Besides it, the people who evaluates the documents is also the writer who documents the documents, it will be very subjective to measure some items.), only four of them are used to evaluate the quality of the documents of the two applications mentioned in chapter 4, they are, inconsistency amount, error amount, size and missing information.

5.3.2 Experiences with Using the Metrics for Evaluation

It is simple to compare the size of the documents. The comparison can be done by just counting the number of pages.

In order to measure the amount of errors, inconsistencies and missing information, the information of the interface documents should be understood carefully at first. This is the prerequisite for evaluating the documentation correctly. At least, the information should include,

- *Background knowledge*. It is very hard to imagine that the readers can

understand the interface document without any background knowledge. For example, in the second application, the readers should at least know what a DVD recorder is, what functions it provides and how to operate it, etc.

- *Structure of the software system*. How much components are there and what are their functions (a rudimentary grasp is sufficient), etc.
- *Services*. What services are provided by the component, in other words, what services are requested by the caller, and how to request this services, are there any limitation, etc?

After the information is recaptured fully, the evaluation process can be started. The following describes the characteristics of missing information, inconsistencies, and errors respectively.

- *Missing information*. When the readers feel uncertain or puzzled for a certain piece of information. However, this information is not inconsistent with others, and the reader cannot find anything wrong in this information. A general explanation to this situation is the case missing information occurs. Another expression of missing information is the readers themselves feel it is necessary to add more information. Missing information can be supplied in the future.
- *Inconsistency*. If there appear to be different descriptions for a same subject, and the readers could not judge which description is correct, then these different descriptions are inconsistent. One thing should be mentioned here, if the reader can judge which description is correct, this should be classified as an error.
- *Error*. Some errors can be found easily, such as spelling mistakes. However, some are not that easy to find, such as errors in the design. Generally, when the readers feel uncertain about some information, and this information is not inconsistent with others, even though the description is abundant, then it may be that the information itself is an error. Determination of errors is the most difficult one of the four metrics, if necessary, people have to contact the author to seek help. Error information can be corrected in the future.

When the readers feel confused about one piece of information and they cannot recognize which type (missing information, inconsistency and error) of this information is, the author provides a suggested sequence for recognizing missing information, inconsistencies and errors. This sequence is mainly based on the personal experience of the author, the occurrence probabilities and the order of severity of the missing information, inconsistency and error. Figure 5.3 shows the sequence. In the figure, YES means the reader can recognize the type, while NO means cannot recognize. Arrows indicates the direction of the sequence.

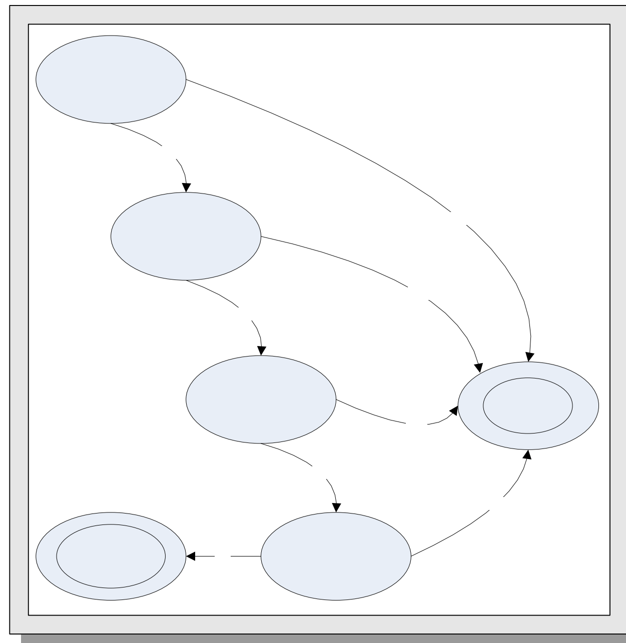


Figure 5.3 Recognizing sequence

At first, if the readers can identify the type of a certain piece of information by themselves, then of course, they can directly go to confirm. If not, the readers should at first check if there is any information missing, then check if any inconsistency occurred, at last, check if this piece of information is wrong or not. If the readers still cannot identify after these steps, they should contact the developers for help.

How about the occurrence frequency of errors, missing information and inconsistencies in each section of the interface document? Figure 5.4 gives a statistical chart. This chart is based on the results of application 2.

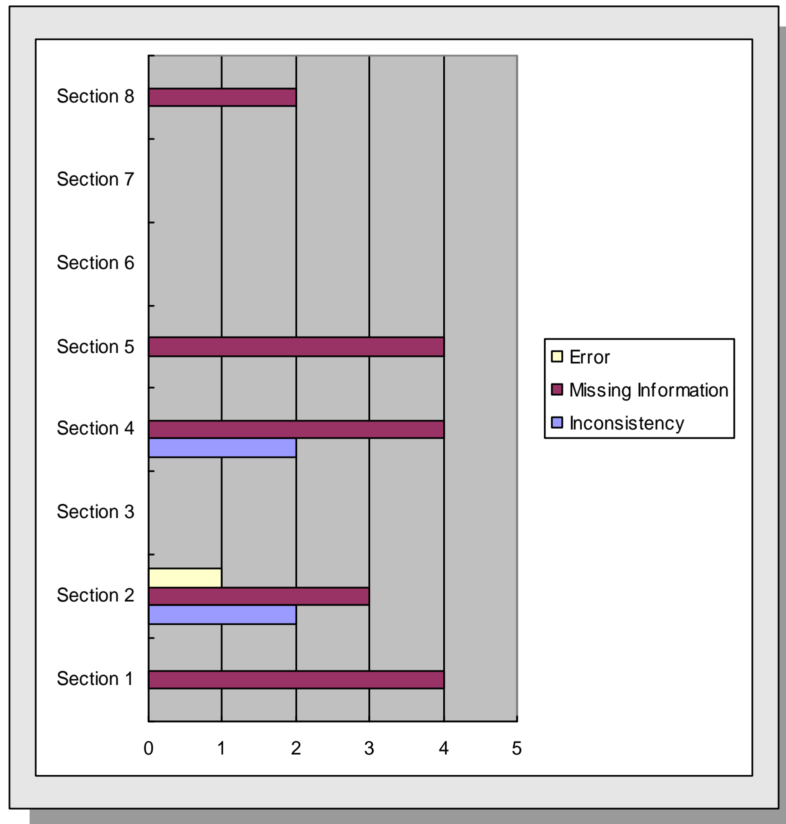


Figure 5.4 Statistics of the occurrence frequency of errors, missing information and inconsistency in each section of the interface document

As shown in Figure 5.4, the error occurs in section 2, Services. The reasons are,

- The chief reason is, the service is the most complicated section in the interface document. Therefore, errors are most likely to occur in this section.
- Unlike other sections, most description languages used in the service section are not nature language. In nature language, some tiny syntax or semantic mistakes may not influence understanding and judging. However, in interface description language (both formal language and informal language), no errors can be ignored.
- Service can reflect the design of an interface, hence, any errors in the design will firstly be found in the service section. Such as, a function was designed without any parameters in the record interface, but there should be parameters in fact.

Other locations where errors occurred,

- *Section 3, External Data Types, Variables and Constants.* This section is as complex as the service section so that it is possible to make errors by carelessness.
- *Section 7, Protocol.* Same reason as above. It is possible that there are hundreds of lines in one figure, so it will be difficult to identify each of lines and letters in this figure correctly.

Missing Information may occur by，

- *Section 1 Introduction.* Lack of explanation of abbreviation and notation.
- *Section 2 Service.* Not enough conditions for error code return. Not error code returned by the function. For example, there is such a saying in Philips document, "Each synchronous function of *HDDN returns an error code, indicating the result of the function.*" But in Figure 3.5, O[1] does not indicate any error code return. Perhaps this is just because the author forgot to write down the return value.
- *Section 4 Error Handling.* No error handling dictionary, and its explanation.
- *Section 5 Limitations and Qualities.* Insufficient description of limitation and qualities.
- *Section 8 Rationale and Issues.* No rationale and issues.

Where inconsistency is likely to occur in:

- General interface description and detailed interface description. For example, it is an inconsistency if the functions mentioned in these two descriptions are different.
- Detailed interface description and the protocol.
- Detailed interface description and the error handling. For example，Philips' document mentioned that, "Any function if called in the wrong or error state, will return the error code, beside it, asynchronous functions will return an event." However, not all service functions follow these rules, that is, in part of the detailed service function descriptions, no indication of this return can be found.

6. Conclusion

In order to answer the thesis project question, "how a software component's interface should be documented so that others can easily use the documentation to implement, use, and maintain this software component", this report proposed a template for documenting software interfaces. Additionally it gives some guidelines about how and where to document the various elements of software interfaces.

To evaluate whether the template designed in this project is a valuable tool for documenting software interfaces or not, this template was applied in two different applications within Philips. After evaluating the interface documents, which were redocumented by using the template, the results show that the quality of the redocumented documents is improved (The template can help the engineer to document the information related to the interface in a structured way. The size of the interface document can be reduced and part of the missing information can be found by using this template). Thus it can be seen that this template is a valuable tool to describe the different interfaces in these two applications. Simultaneously, this result also illustrates that the question mentioned above was answered satisfactorily. However, for finding and reducing error, inconsistency and part of missing information, the template designed in this thesis project cannot provide effective method, that is, finding and reducing error, inconsistency and part of missing information still mainly depends on the experience of the people who document or evaluate the document, not depend on the template.

On the other hand, since the template has been used in two applications within Philips, the author also gained some experiences with documenting software interfaces and reading literatures. Sharing these experiences is necessary in order to help software engineering. These experiences include documenting and evaluating, two aspects. Documenting experiences focus on how to produce a good quality interface document, and it gives some rules and suggestions. While evaluating experiences focus on how to evaluate an interface document, and analyze the result after evaluation.

I hope this template and the experiences can provide useful information that can help people to communicate, to generate good software interface documentation and to establish a standard for documenting software interfaces.

Glossary

architecture	See software architecture
caller	A component which requests the service
component	A unit of responsibility and functionality on a specific abstraction level.
context diagram	A diagram that shows a system or component and the components it interacts with.
interaction	An exchange of information between two components.
interface	A set of services that specify all or part of the externally visible behavior of a component
interface documentation template	A document specifying the structure and content of an interface documentation
IDL	Interface description language
provider	A component which provides the service
protocol	Protocol defines all allowable sequences of services and interactions, represents the complete behavior of the interactions between two components.
property	Property is additional information about elements and relations.
service	A behavior of a component for the benefit of and requested by a caller of that component.
sequence diagram	A sequence diagram is an interaction diagram that details how operations are carried out, that means what events are sent and when events are sent.
statechart diagram	A statechart diagram shows the possible states of the component and transitions that cause a change in state.
software architecture	Software architecture of a program or computing system is the structure or structures of the system, which comprise

software components, the externally visible properties of those components, and the relationships among them.

system — A composition of components organized to accomplish a set of specific functions.

Appendix A: Template Outline

The following gives an overview of the template for documenting software interfaces

Section 1 Introduction

- **Identifier**
- **Version**
- **Design team and author**
- **Creation and modification history**
- **Audience**
- **Related documents**

Section 2 Service

- **General description**
- **Detailed description**
  - **Syntax**
    - Service function name
    - Purpose of the service function
    - Parameter name
    - Parameter type: I (Input), O (Output) and I-OPT (optional input)
    - Description for the parameter
    - Prototype
    - Function type (used in embedded real time system): asynchronous and synchronous
  - **Semantic**
    - Requires
    - Modifies.
    - Effects
    - Description
    - Open Point
  - **Event**
    - Name
    - Purpose
    - Requires
    - Effects
    - Data
    - Descriptions

Section 3 External Data Types, Variables and Constants

- **Types**
  - Name
  - Purpose
  - Definition
  - Description
- **Variables and Constants**
  - Name
  - Purpose
  - Description
  - Value (optional)

Section 4 Error Handling

- **Error code dictionary**
  - Error code name
  - Severity Level
  - Error Condition
  - Error Handling Behavior
- **Error handling mechanism**

Section 5 Limitations and Qualities

- **Limitations**
- **Qualities**

Section 6 Component Requires

Section 7 Protocol

- **General description**
- **Detailed description**
  - Statechart diagram

Section 8 Rationale and Issues

- **Rationale**
- **Issues**
  - Open point
  - Severity level
  - Description
  - Possible solution.

Appendix B: Evaluation Result

1. The evaluation result of the redocument documentation of SACD1000

2. The evaluation result of the redocument documentation of DVD recorder (record interface)

Size Reduction

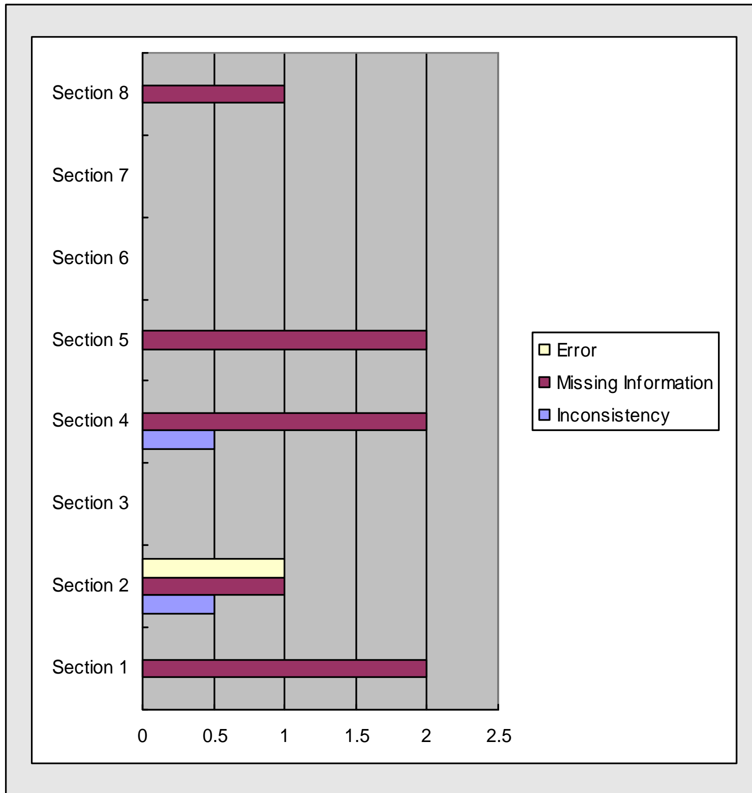
3. The evaluation result of the redocument documentation of DVD recorder (playback interface)

Reducti

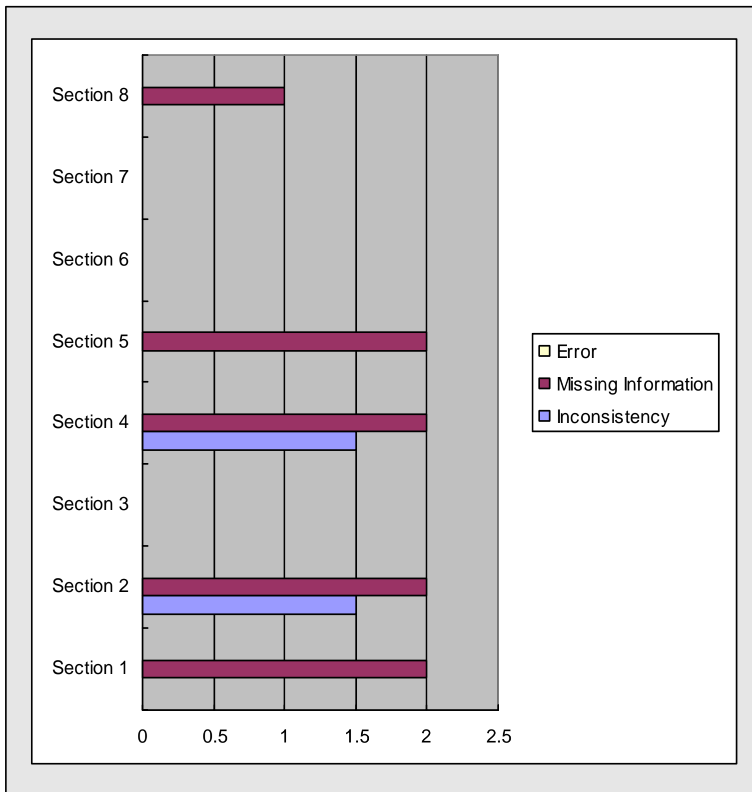
ation R

4. Statistics of the occurrence frequency of each error, missing information and inconsistency in each section of the interface document

Error Reduction



Record Interface Part



Playback Interface Part

Reference

[1]. Len Bass, Paul Clements, Rick Kazman, Ken Bass. 1997. *Software Architecture in Practice*. Addison-Wesley.
[2]. Design. Barbara Liskov, John Guttag. 2001. *Program Development in Java. Abstraction, Specification, and Object-Oriented.* Addison-Wesley.
[3]. Paul Clements, Felix Bachmann et al. 2003. *Software architectures. Views and Beyond.* Addison-Wesley.
[4]. Michael A. Ogush, Derek Coleman, Dorothea Beringer. 2000. *A Template for Documenting Software and Firmware Architectures.* Hewlett-Packard.
[5]. Ronald A. Grace, Derek Coleman, Michael A. Ogush, Steve Rhodes. 2000. *Experience with Documentation of Software Architectures.* Hewlett-Packard.
[6]. Timothy C. Lethbridge, Janice Singer, Andrew Forward. 2003. *How Software Engineers Use Documentation: The State of the Practice*. IEEE.
[7]. Lei Xiao. 2004. *Research Assignment Report.* TU Delft.