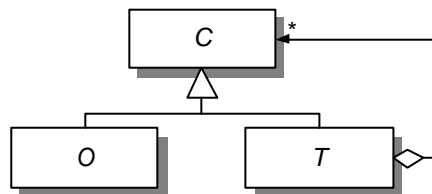


*Documentation of Object-Oriented
Systems and Frameworks
COT/2-42-V2.4*



Centre for Object Technology

*Centre for
Object Technology*

Revision history:	V0.1	990905	First draft outline
	V1.1	991208	Minor changes in Chapter 2
	V1.2	991212	Changes after doc meeting 10.12.
	V1.3	991213	Changes in Chapter 10
	V1.4	991213	Rewrite of 6.2.3, removed 6.2.4,6.2.5
	V1.5	991213	Changes to tools chapter
	V1.6	991214	Minor updates
	V1.7	000330	Minor updates after internal review
	V1.8	000413	Language review and correction
	V2.0	000810	Rewrite based on review feedback
	V2.1	000816	Rewrite based on 0008015 meeting
	V2.2	001011	Rewrite based on FOH comments
	V2.3	001220	Rewrite based on review feedback
	V2.4	010110	Reformat for publication

Authors: Aarhus University:
Kasper Østerbye, Ole Lehrmann Madsen, Elmer Sandvad
Bang & Olufsen:
Carsten Bjerring
Danfoss Instruments:
Ole Kammeyer
Danfoss Drives:
Stefan Helleman Skov
Danish Technological Institute:
Finn Overgaard Hansen, Flemming Hansen

Status: Final

Publication: Public publication

Summary:

This report discusses documentation of the design and the implementation aspect of object-oriented systems and frameworks. The targets for the produced documentation and for this report are developers of object-oriented software. The report divides documentation into three main categories, tutorial, rationale, and reference documentation. Each kind of documentation is discussed and recommendations are presented. Tool support for the development and maintenance of documentation are addressed as well.

© Copyright 2000

The Centre for Object Technology (COT) is a 3-year project concerned with research, application and implementation of object technology in Danish companies. The project is financially supported by The Danish National Centre for IT-Research (CIT) and the Danish Ministry of Industry.

The participants are:
Maersk Line, Maersk Training Center, Bang & Olufsen, WM-data, Rambøll, Danfoss, Systematic Software Engineering, Odense Steel Shipyard, A.P. Møller, Aarhus University, Odense University, University of Copenhagen, Danish Technological Institute and Danish Maritime Institute.

1. INTRODUCTION.....	3
1.1 PURPOSE AND SCOPE.....	3
1.2 TARGET GROUP	3
1.3 O-O SOFTWARE ELEMENTS.....	3
1.4 HOW IS DOCUMENTATION OF OO SYSTEMS DIFFERENT	5
1.5 READING GUIDE	5
2. GENERAL NOTES ON DOCUMENTATION	6
2.1 REQUIREMENTS FOR DOCUMENTATION.....	6
2.2 GUIDELINES FOR ARCHITECTURE DOCUMENTATION	7
2.3 TRACEABILITY	7
3. TYPES OF DOCUMENTATION.....	8
4. TUTORIAL DOCUMENTATION.....	10
5. RATIONALE DOCUMENTATION	11
6. REFERENCE DOCUMENTATION.....	12
6.1 ENTITY-BASED REFERENCE DOCUMENTATION	12
6.2 ARCHITECTURE DOCUMENTATION	14
6.2.1 <i>Documenting Configurations.....</i>	<i>15</i>
6.2.2 <i>Structuring the Architecture Documentation.....</i>	<i>16</i>
6.2.3 <i>Hierarchical Design Descriptions.....</i>	<i>18</i>
7. DOCUMENTATION IN THE DEVELOPMENT PROCESS.....	20
7.1 WATERFALL DEVELOPMENT	20
7.2 ITERATIVE DEVELOPMENT	20
7.2.1 <i>Documentation Process.....</i>	<i>20</i>
7.2.2 <i>“2+1” View Model – as Process Driver.....</i>	<i>21</i>
8. TOOL SUPPORT FOR DOCUMENTATION.....	22
8.1 AUTOMATIC GENERATION OF THE ENTITY-BASED REFERENCE DOCUMENTATION.....	24
8.2 USE OF DEVELOPMENT ARTEFACTS IN DOCUMENTATION	25
8.3 CONSISTENCY	26
8.3.1 <i>Consistency between Development Artefacts and Documentation</i>	<i>26</i>
8.3.2 <i>Consistency between Documentation Types.....</i>	<i>26</i>
8.3.3 <i>Consistency between Development Artefacts.....</i>	<i>27</i>
8.4 LINKS.....	27
9. USE OF VIDEOS AS DOCUMENTATION MEDIA.....	28
10. CONCLUSIONS AND RECOMMENDATIONS	29
11. REFERENCES	31
A. RATIONALE FOR ENTITY DOCUMENTATION.....	33
B. COMPARISON OF “2+1” WITH KRUCHTEN	34

1. Introduction

1.1 Purpose and Scope

This report will describe aspects of documentation of object-oriented systems and frameworks. The scope of this document is to describe the documentation of software intended for developers and maintainers of the source code, and how to document reusable components to facilitate their use by other software developers. While requirements, domain analysis, test etc. are important aspects and need documentation in their own right; we have limited ourselves to documentation of the design and implementation aspects.

We have made no assumption regarding the size of systems we document. Our recommendations include alternatives in several places that take the size and type of the system into account.

This report is the result of activities in the Centre for Object Technology (COT), where a working group in the Case 2 project was set up to examine documentation of object-oriented systems and frameworks. The report is an attempt to provide a bridge between a scientific technical report, and a set of recommendations for industry to follow. This implies that the report will have many discussions as is usual for science, and it will give recommendations that are not completely argued in the report, but have been simply agreed upon by the group of authors.

1.2 Target Group

The target group for this report consists of developers and maintainers of OO systems and frameworks. Documentation tools are also discussed, and hence people responsible for development environments at a department or development team might find the report of interest.

1.3 O-O Software Elements

In this section, we briefly characterise OO software elements in order to define the subjects to be documented. It may be useful to consider the distinction between frameworks and other software elements. In OO software development, a software module is usually a set of classes with operations and data-items (variables). In UML terms, a software module is a **package**.

The purpose of most documentation is to document packages consisting of classes and the relations between packages and classes. The volume and quality of the documentation depend on the importance of the package. A framework with many users in different organisations obviously requires substantially more documentation than a simple application-specific library package. Similarly, the complexity and importance of a package will determine the volume of required documentation.

Consider the following attempt to characterise various sorts of software packages:

- *Application package*
An application package may be defined as a set of classes where the application is an instantiation of one or more of the classes. In addition, there is no client software using this package.
- *Library package*
A library package is a set of classes providing useful objects and functionality, and a typical example would be a container package. A library package is used by one or more other packages often referred to as *client packages*. A library package may be characterized as being placed in a spectre between:
 - *A general library package*
The library package is used by many other (independent) packages in different applications, etc.
 - *An application-specific library package*
This package is written to support a specific application.
- *Framework package*
A characteristic feature of a framework is that it provides an architecture and basic functionality for a set of applications. In a simple case, the framework may provide a skeleton of an empty application where the user has to add the application functionality in order to make it useful. In other cases, the framework may provide default functionality and actually define a complete application. The framework user may then redefine and/or extend the functionality.
Another framework dimension is the distinction between black- and white box frameworks:
 - *Black box framework*
A framework in which components are reused, mostly by composing object instances. No knowledge is required of the internals of the framework to use the framework.
 - *White box framework*
A framework in which components are reused mostly by inheritance from classes in the framework. The framework user may add new subclasses and re-define/extend the behaviour. Some knowledge about the internal implementation of the framework is required to use it.

There is no sharp distinction between general library package, application-specific library package, black box framework, white box framework, or application package. The purpose of the above terminology illustrates the spectre of packages discussed in the report, but mostly the report does not discuss the specific documentation needs for a given type of package.

A well-designed system often consists of a small application package, a few application-specific library packages, and the majority of the code consists of general library and framework packages. A software package often starts as an application package or an application-specific library package.

During the project (or re-use phase if it exists) a package may develop into a general library package or a framework. A good strategy for software development is to develop packages that implement a well-defined functionality. Each class in a package should serve a clear purpose, and the interface of the package/class should be small and definite, thus facilitating the upgrading of a package to a reusable asset. In fact, one should develop reusable packages. However, one must strike a balance between premature generalisations of the functionality of a package.

1.4 How is Documentation of OO Systems Different

Documentation of object-oriented systems is different from documentation of traditional non-OO systems for two reasons; the use of an object-oriented model and the fact this model pervades all development activities.

The object-oriented model brings classes, inheritance, association, aggregation, object etc. in addition to the existing notions of modules and composition. These concepts are used as fundamental building elements in the design process, where these concepts are used to express and document design ideas. The model can be described diagrammatically with a standard notation for these OO concepts, e.g. the UML standard, resulting in more uniform system documentation in different projects and between different companies in the industry.

The use of some of these new concepts, e.g. the use of design patterns and some advanced use of polymorphism, requires an explanatory documentation as these new concepts are more powerful but also more complicated to understand than traditional functional concepts. However, design patterns are today so well known that they can be used to reduce the amount of documentation, as one can simply state that a certain part of the code realises e.g. the visitor pattern.

The other difference emanates from the fact that OO concepts can be used in all activities starting with analysis activities, in design and finally in an actual implementation in a given programming language. The concept of a class embraces all these different activities and it can be expressed directly in an OO programming language.

A software system often includes distribution of executable between different computers as well as multitasking where several processes or tasks are executed on the same computer in parallel or as simulated threads. Such aspects must also be documented. The UML standard includes notation useful for describing both distribution, and for specifying mapping to executables, components etc.

1.5 Reading Guide

Chapter 2 presents the overall requirements for documentation of object-oriented systems.

Chapter 3 presents an overview of the different document types suggested in this report.

Chapters 4, 5 and 6 describe in details the nature and contents together with the guidelines for the different document types.

In Chapter 7, the documentation is put into perspective with development processes, both the traditional waterfall approach and the more modern iterative development approach. Chapter 7.2.2 presents guidelines for the process of writing documents.

Chapter 8 presents the roles and importance of tools in relation to the documentation process.

Chapter 9 presents the concept of using video recordings as an alternative and supplementary approach to the normal documentation media.

Chapter 10 presents conclusions and recommendations.

2. General Notes on Documentation

2.1 Requirements for Documentation

The following requirements serve as a background for most of our discussions in the report, and we will return to them in the conclusion where we presents our recommendations on how to produce documentation of object-oriented software.

1. The volume of system documentation should be **minimal**, but sufficient in order for the developer to locate the information required in order to modify and enhance the system.
2. Since software development tends to evolve into a more and more (geographically) distributed process, the **accessibility** of documentation should be considered.
3. When a developer joins an ongoing project or explores a completed system in order to enhance it, he or she should be able to understand the static and dynamic **structure** of the system.
4. Writing the documentation should be a **natural part** of each iteration of the development process.
5. Another important issue is to **minimise redundancies**, as they are possible sources of inconsistencies.
6. Another topic is **traceability**, in the sense that it should be possible to identify how and where each requirement has influenced the design model and its implementation, and ideally it should be possible from each place in the implementation to trace back to the requirements that let to it.
7. The **main abstractions** and general design decisions in the system should be easily recognisable.
8. The documentation should be separated in sections describing different **aspects** of the system. Important aspects are the static architecture, the dynamic behaviour, and the reference documentation for all system entities.
9. The documentation should include a basic description of the **domain** it is intended to model. This is important for providing an understanding of the environment in which the system will function and of the concepts introduced in the model.

10. A concise description of the **interface** (input/output) to the environment is required to define precisely what is part of the system and what is not.
11. Finally, a description of how the OO system should act on the **events** from the environment should be included and contain a reference to the more specific documents describing the environment (HW, protocols etc.).

2.2 Guidelines for Architecture Documentation

Besides the above general documentation requirements, it is also worth to notice the general guidelines below, which are especially targeted at documenting the architectural aspects of a system as opposed to the detailed entities. We distinguish three important sides of architectural documentation, the contents, structure and writing process.

Contents:

- **Introduce main concepts.** In general the documentation should explain/define main concepts. A document could for example contain an appendix with used concepts. In many cases a concept is also represented in the system as a class. The document should clearly state which concepts are also represented as classes. The major concepts will often be described in the tutorial, whereas concepts used in subsystems may be introduced in the architecture manual.
- **Describe functionality.** The functionality of the overall system/framework should be described, for example in the form of Use Cases and/or scenarios.
- **Document interrelationships.** The system/framework should be divided into a number of logically related entities. It is important to document these interrelationships.

Documentation structure:

- The architecture should be **hierarchically** structured. The documentation should describe the top-level structure in terms of systems and subsystems (packages/sub packages) and then move towards more and more detailed elements.
- **Use examples.** Most people find it easier to learn from concrete examples than from abstract descriptions. In general, descriptions of framework elements should be accompanied by examples of their use. The documentation should include an example of a small, but complete system. If there are many objects, a hierarchical description should be given. Such examples belong to the tutorial documentation.

Writing process:

- It is a creative process to identify and document important interrelated objects. The architecture documentation is usually not a report that can be generated by pushing a button on the CASE tool. However, as we shall see in section 8.2, tools can be used to produce diagrams useful for the explanation of architectural design decisions.

2.3 Traceability

An important property of software documentation is **traceability**, which can be subdivided into forward and reverse traceability. It is a complex subject, in which we do not

have much experience. The following should therefore be taken as a preliminary discussion of the issue.

Forward traceability describes how a given requirement is realised in a given design model, how a specific design element is realised with the actual implementation language, and why (rationale) it was solved as it were.

Reverse traceability describes for a given element which requirements it realises. Each element should thereby realise one or more requirements.

We currently see two forms of **forward traceability**:

1. Traceability of initial requirements.
2. Traceability of new and changed requirements.

The form and structure of the requirement documentation have not been considered as part of this document. For the purpose of this section, we assume the existence of a requirement document containing a list of requirements.

Re. 1)

We suggest that the requirement documentation should contain a list of the requirements. For each requirement, there should be links/references to the documentation of the part of the software implementing the requirement. A link/reference may refer to a section in the rationale discussing design decisions and alternatives to satisfy the requirement. Further links/references may then point to other places in the documentation.

If each element is annotated with links to the requirements it fulfils, such a forward trace-list can normally be automatically generated by a software tool, which extracts the information from a case tool. The tool scans the repository for each requirement and lists the elements, which have references to the actual requirement number.

Re. 2)

This is an intricate issue to handle since it may require integration with a version control system. Dedicated version control systems can normally handle this, but it is also necessary that the version control system and the rest of the development tools be integrated to utilise this functionality. In general, the requirement documentation should be updated to reflect the new requirements. Also, the rationale should/could document changes of design decisions made necessary by new and changed requirements.

Reverse traceability can be obtained with an annotation of a given element with the requirements it fulfils. Each element should thereby have at least one reference to a requirement. If the functional requirements are specified as Use Cases, reference can be made to a given Use Case name or Use Case number.

3. Types of Documentation

Documentation can cover different aspects, ranging from introductory material for novices, to reference documentation for experienced developers. In addition, a rationale for design decisions may be included. The separation between tutorial and reference

documentation has been used for many years. One example is the *Pascal User Manual and Report* [Jensen&Wirth75]. The user manual is an introductory description of Pascal, and the Report is a concise definition of the Pascal Language. The preliminary *ADA Reference Manual* [Ada79a] was accompanied by an excellent example of a rationale [Ada79b]. Some of the Java libraries are presented through *trails* that serve as a tutorial, and an API documentation, which corresponds to the entity documentation.

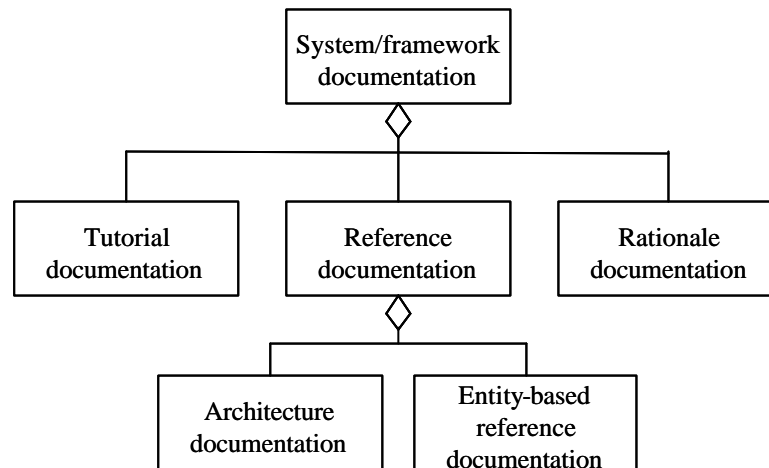


Figure 1. Overview of documentation types

With regard to documentation of OO software systems and frameworks, we will distinguish between the following types of documentation:

- *Tutorial documentation:*
Introductory material for novices.
- *Reference documentation:*
Ideally, a complete description of software for experienced users. The reference documentation is divided into:
 - *Architecture documentation:*
Describing the overall structure and transverse relations of the system.
 - *Entity-based reference documentation:*
Describing each entity in the system in its own right.
- *Rationale documentation:*
Discussions of design alternatives and decisions

Each documentation type will be described in details in the subsequent sections.

The main “clue” in separating the documentation into these four parts is to allow the reference sections to remain concise. This implies that it is not intended for novices to the system; hence they need a tutorial. Furthermore, the reference documentation does not contain the rationale, that is, it describes what but not why, again to make it concise. The why’s are important, especially for reasons of maintenance, and are included in the rationale documentation.

The reference documentation has been split into two, to emphasise that an entity-based documentation does not suffice. The pragmatic separation between what to document in the architecture part, and the entity part is that we will recommend that the entity based reference manual is generated by extracting source-code comments, e.g. using JavaDoc.

We do not discuss whether the four types of documentation should be put in different sections in one document, or in separate documents. This clearly depends on the tradition of the organisations and on the size of the system or framework. We will, however, discuss the advantages that can be obtained by making all the documentation available online.

Finally, we do not propose that the four types of documentation should be mandatory for all kinds of software. The type of documentation required for a software package clearly depends on the kind of package (cf. section 1.3) and other factors such as the number and type of users. A product used by a large number of external users clearly needs more documentation than an application-specific library package used by a small team working together.

In the following sections, we will provide a detailed description of the four types of documentation in the following order: tutorial-, rationale-, entity-, and architecture documentation. This order is not a matter of priority; in fact we consider the architecture documentation to be the most difficult part to write because there is no common understanding of how to structure the architecture documentation. By treating them in this order, it should be clearer what aspects we do not consider as part of the architecture documentation, in that we recommend that the same aspect is not described twice (at least not at the same level of detail) to minimise redundancy.

4. Tutorial Documentation

The tutorial is the first introduction to the system. The purpose of the tutorial documentation is to provide developers with an overview of the system and to present the major parts of the system. Its purpose is not to be complete, but to provide a mental model of the system, and to give an overview of the major architectural characteristics.

We currently know of the following kinds of tutorial information:

1. *Cookbook*

A cookbook is a description of how to solve typical problems by using a framework or enhancing a software system. An example is the Smalltalk cookbook for covering almost anything from writing a class to making graphical user interfaces. It can be seen at:

<http://www.objectshare.com/products/smalltalk/vworks/info/cookbook/httoc.htm>

2. *Textbook*

A textbook is a detailed presentation, typically used in a course on a given subject. This is perhaps the ultimate tutorial documentation. One of the first object-oriented frameworks, the discrete simulation framework DEMOS by Birtwistle [Birtwistle79], is documented in this manner.

3. *Pattern language description*

Ralph Johnson's paper [Johnson92] on using pattern languages to document frameworks. Only a few examples exist to illustrate the technique. Notice that a pattern language is only somewhat related to a design pattern. A pattern language is a set of

hints on how to design a thing, and how to address problems that typically arise during the design process, and hints on how best to organise one's work (in which order to do things) when designing.

4. *Concrete examples*

This is a collection of concrete examples of using the system/framework. The examples should be brief and easy to understand. Each example must be annotated in a fashion that shows what it does and how it works. It is important to notice that an example in this context is not a full application but a small program focusing on illustrating a specific point.

5. *Guided construction tour*

This is a complete example of how to develop an application by using a set of software packages. At the outset, it defines the requirements of the application and then proceeds to describe step by step how to develop the application. As opposed to the concrete examples, the guided tour presents the process as well as the product. Such guided tours are often online.

6. *FAQ (frequently asked questions)*

A FAQ is a list of questions and answers. The questions have really been asked, and the questioner has received an answer that actually helped. The FAQ should be structured in a fashion that groups together the same types of questions/answers.

It is neither necessary (nor recommended) to produce all of the above kinds of tutorials. Concrete recommendations will be given in Chapter 10. Conclusions and Recommendations.

5. Rationale Documentation

This documentation contains a description of the design choices that resulted in the architecture of the software, the choice of algorithms, or the selection of a 3rd party packages to depend on. An important aspect of the document is what alternatives were considered, and why they were not chosen. Sometimes one chooses an alternative, which is not based on any current requirement but on the expectation that requirements of a given type will soon appear. At other times, the best solution for a given requirement is either not mature or too expensive. Such information should not be intermixed with the architecture documentation, which is intended to stay concise and describe the structure of the system.

The rationale can be formulated as a supplement to the reference documentation, that is, it follows the structure of the reference documentation and discusses it in relation to the documentation. Or a rationale can be structured as a set of topics where the discussion of any particular topic contains references to reference sections.

Typically, the rationale explanations will refer back to the requirements to the system.

We believe that rationale documentation is important to maintain, but it has a long pay-back time. The rationale document will keep track of why a certain thing was done in one way or the other. As such, it is an interpretation of how to achieve the obligations from the requirement document. In later extensions to the system, the programmers will

then not try out solutions that have already been tried. The rationale documentation could also be produced as a video recording (see section 9).

The rationale documentation can also play an important role in respect to traceability, as discussed earlier in section 2.3. Normally traceability can be done by references from the requirement, but a more extreme position is that the rationale is the exact joint that explicitly addresses traceability, by deliberating why a given requirement is fulfilled the way it is.

6. Reference Documentation

The ideal reference documentation is complete, in the sense that a reader will find the answer to all structural and functional questions regarding the software described here. In practice, this is not possible to achieve, as one must find a reasonable balance between the cost of producing documentation and the cost of finding answers to questions regarding the software. However, the structure of the reference documentation should make it easy to locate the point where information should be available. It is important to point out that reference documentation is not intended as introductory material. Reference documentation is for developers who need answers to specific questions.

6.1 Entity-based Reference Documentation

Entity-based documentation is documentation associated with concrete entities in the software. An **entity** can for example be a package, a class, a procedure, a task/process, etc. In principle, entity-based documentation should be extracted from the software by appropriate tools (see section 8.1). Some information is directly available from the syntactic structure of the code and other information needs to be supplied by the programmer as comments.

We propose that the programmer provide the following information as comments in the source code:

- *Responsibility* (package or class).
Describes the overall purpose and rationale for the given item (package or class).
A one line description, and if necessary an extended description.
- *Functionality* (operations).
Both a one line description and an extended description. The extended version includes a description of parameter usage and return values, and the conditions in which exceptions are raised.
- *Invariant* (package and class).
Aspects of the class that must always be true between calls. This is typically a relationship between its attributes.
- *Pre- and post conditions* (operations).
What does this operation expect in order to function, both in its parameters and in the state of the object in which the operation is located, and what does the operation promise to be true afterwards. Pre/Post conditions are important tools in contract-based design [Meyer97].
- *Type information*

This is intentional information that may not be extracted from the source code itself, e.g. exception class, abstract class. This is only necessary for some languages, i.e. Beta unifies many mechanisms in the pattern construct, but the programmer uses the pattern as a class or procedure. Java and C++ has syntax for most types, Smalltalk does not syntactically differentiate between normal classes and abstract classes.

- *Extensibility requirements*

The extensibility requirements are specifications of how classes of a framework should be specialised. For a (white box) framework, it should be documented if a class may be specialised and to what extent a procedure may be redefined/specialised. Often this is covered by tutorial information. However, a systematic description is required. It is important to distinguish between methods that *must* be specialised for the application to function, and methods that *may* be specialised. In some languages, there are syntactic differences between these two (C++ pure virtual functions, Java abstract methods). It is also important to know if the method is intended to be called by the application code, or if it is called as the result of some internal action in the framework.

Contents	
	Packages and classes, but not attributes or methods.
Package ₁ "name"	
	<i>Responsibilities</i>
Class ₁ "name"	
	<i>Type information</i>
	<i>Responsibilities</i>
	<i>Inherits from</i>
	<i>Implements interfaces</i>
	<i>Invariant</i>
	Public interface:
	Attribute ₁ "name"
	<i>Purpose</i>
	Attribute _n "name" ...
	Method ₁ "name" (parameters): return value
	<i>Functionality</i>
	<i>Pre/post conditions</i>
	<i>Extensibility requirements</i>
	Method _m "name" ...
	Protected interface:
	...
	Private interface:
	...
	Class _n "name" ...
Package ₂ "name"	
...	
Package _n "name"	
...	
Identifier index	
	Sorted lexicographically

Figure 2. Example of entity documentation

This outline corresponds to what the typical tools can generate automatically, although it requires careful consideration to get the proper layout of responsibilities and other documentation. The identifier index is a very convenient aspect and must be generated automatically. A typical entry will have the structure of 'name:class:package', thereby making it easy for the user to find a specific entry, e.g. to find an update method in the right class. Similarly, the content part is necessary for large systems in order to navigate quickly to the appropriate class.

Please note that the inclusion of attributes in the public interface does not signify that all attributes should be public, but that sometimes it is part of the model that attributes can be accessed and manipulated directly. This corresponds to the choices in Java and C++ where attributes can be declared public. The private interface is included in the outline because maintainers need it. If the outline is intended for using the class as a server, neither the protected, nor the private interface should be part of the entity documentation.

The physical layout of the entity documentation will be different for paper and for a web-based version. The web-based version will have links from the content section and from the index and should contain cross-references, for example, to be able to easily browse to the class defined as a return type of a given method.

Appendix A. contains the background for which elements to document in the entity documentation. It is based on the IEEE standard on documentation [IEEE-std1016-1998], and the report [MJII-MIA-90-1].

6.2 Architecture Documentation

In order to understand a system or framework, it is necessary to understand how groups of logically related entities work together. The architecture documentation describes such related sets of entities, whereas the entity documentation describes the entities in isolation. The architecture documentation should describe the following properties:

- *Relation between multiple entities:*
Classification (inheritance), composition, associations, etc.
- *Interaction between entities:*
Active objects/processes, communication and synchronization between processes, shared objects, object states, calling sequence, etc.
- *Physical deployment of entities on the hardware.*
- *Organization of entities in the development environment:*
The static organization of the entities in terms of directories, files, etc.
- *Organization of entities in the target environment:*
How are objects and functions packed in terms of executables, link modules and/or binary components.
- *Architecture goals and constraints* [RUP-SAD-Artifact]
Requirements and objectives that have a significant impact on the architecture: use off-the-shelf product, portability, distribution and reuse. It also captures special constraints such as: design and implementation strategy, development tools, team structure, schedule, legacy code, etc.

Each of the above aspects will be further elaborated in the sections to follow.

Examples of architecture documentation:

- Description of a set of entities using UML such as package diagrams, class diagrams, state machines, sequence diagrams, etc., *accompanied by a textual description of the relations.*

- Documentation of the Danfoss Instruments USM II framework. Here the logically related elements of the USM II framework are grouped into so-called patterns¹, and each pattern is documented by using a number of different UML diagrams accompanied by textual descriptions.

In [MJII-MIA-90-1] the term *configuration* covers a group of logically related entities and it is proposed to *identify* such configurations and to document the *relations between its entities*, and to document the *relationship with other configurations*. It is proposed to structure the architecture document as a set of hierarchically organised configurations. Section 2.2 provides some general guidelines to be considered when writing architecture documentation. Section 6.2.1 presents an approach to documenting an architecture that takes its outset in documenting the configurations of a program.

In the USM II framework, the term pattern is used for the logically related entities being documented. The term pattern is, however, mainly used for well-established mini architectures that have proved their usefulness in several cases. A configuration that turns out to be reused in several contexts *may* evolve into a pattern.

Disclaimer: The structure and contents of the architecture documentation have not yet been verified experimentally to be an effective way to describe the architecture of a system. However, it is the best recommendation that we can provide at the time of writing.

6.2.1 Documenting Configurations

At the top level, the system may be divided into configurations that are subsystems, and each subsystem may in turn be divided into further subsystems. In some situations the configurations may correspond to a subsystem division of the system/framework, in other situations it may be necessary to identify a number of configurations within the same subsystem. A class/object may play a role in more than one configuration. When a system/framework is documented using design patterns, a class/object may also appear in more than one pattern.

The following means may be used in the architecture documentation:

- *Established patterns:*

It is attractive to use design patterns² to document configurations. A pattern and a configuration are very similar, as they both constitute a group of logically related classes/objects. When a configuration is an established pattern, or be inspired by one or more patterns, relating the configuration to this/these patterns makes the system easier to understand.
- *Diagram types:*
 - As already mentioned, diagrams are useful when documenting configurations. In general, there should be no mandatory rules for which diagrams to use since diagrams should only be used if they contribute useful information. In addition to diagrams, it may also sometimes be a good idea to use abstract algorithms. In general, any kind of picture, diagram and other notation useful for documenting

¹ The term pattern used in the USM II document is not always in accordance with the common use of the term design pattern.

² That is, design patterns in the form presented in [Gamma95].

a given configuration should be used. It is important that the description is concise and understandable. For reasons of consistency it is of course a good idea to use a common notation as much as possible. UML is an obvious notation candidate in most cases but may be insufficient in special situations.

- *Natural Language:*

No diagram or design pattern can stand alone, but should always be accompanied by an explanatory text. Sometimes a configuration is also of such a nature that it requires an explanation, which is not best described by a standard UML diagram, but through a free figure.

6.2.2 Structuring the Architecture Documentation

The purpose of the architecture documentation is to document the structure of the system/framework. As mentioned earlier, we have not surveyed different approaches to documenting architecture in great depth. The “4+1” view proposed by [Kruchten95] and later updated in [Kruchten98] has been the main inspiration for our proposal on how to structure the documentation of a configuration – that is a configuration used in the sense described above, as sets of logically interrelated entities. The main idea that we borrow from the “4+1” view is that there are multiple views from which a system should be documented, and that these views should be tied together by relating them directly to their requirement – the “+1” view.

We propose a model with three main views: **Requirement view**, **conceptual view** and **realisation view**, as these views are mandatory in all systems.

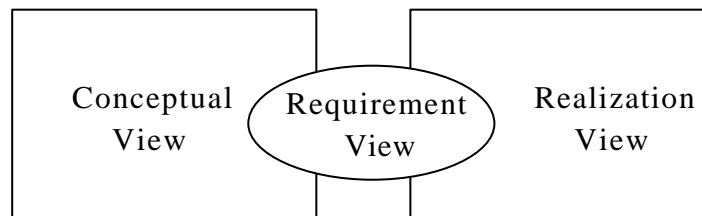


Figure 3. A model with three views

- **Requirement view:**

For a given configuration there is a set of requirements that motivates its existence. These requirements can be in the form of a set of Use Cases/scenarios *representing* all possible uses of the configuration. The description of the scenarios may be in terms of sequence diagrams and/or object diagrams. But not all configurations are motivated by customer requirements, some exist due to decisions made during design, because of lack of time for anything better, or because someone think it will improve the software in some way. The requirement view can be seen as an explicit method of ensuring reverse traceability.

- **Conceptual View:**

The conceptual view presents the architecture of a configuration from a logical point of view, which includes the static properties such as classes, inheritance, associations, etc. It also includes the dynamic properties such as state machine, object in-

teraction and domain intrinsic concurrency. To document the conceptual view, the following aspects/sub-views may be relevant:

- **Package View:** If many classes are present they may have to be organised in packages.
- **Class View:** This view describes classes/objects, inheritance relations, composition, associations, etc.
 - **State machines:** For classes with a state machine, State Charts may be included to document the state dependent behaviour³.
- **Behavioural View:** Concurrency in the form of active objects/processes should be documented, which may be carried out by using class or object diagrams with active classes or objects as well as UMLs sequence- or collaborations diagrams. But also non-trivial sequential behaviour, especially division of labour between a number of objects, should be documented. Again sequence- or collaboration diagrams are valuable tools.

In many contexts a conceptual view is often referred to as a **logical model**, an **analysis model** or **design model**.

- **Realisation View:**

The realisation view documents various aspects of the implementation of the conceptual view/model. The following views may be relevant:

- **Source Code View:** This view describes the static organisation of the software in terms of files and directories, etc. (e.g. with a UML component diagram).
- **Executable View:** This view describes the packaging of the software elements into components delivered to the customer. The software may be delivered as one or more executables, code libraries, etc. In a setting with Microsoft COM binary components, the view should describe how the elements of the conceptual model are packed onto binary components (e.g. with a UML component diagram).
- **Process View:** This view addresses the concurrent aspects of the system at runtime i.e. tasks, threads or processes and their interactions.
- **Deployment View:** This view describes the computers in the system and connected Hardware devices and the mapping of the conceptual model on these computers. This view can also be used to show the mapping of the concurrent processes onto nodes in a distributed system (e.g. with a UML deployment diagram).
- **Data View:** the mapping of parts of the conceptual model onto secondary storage. This mapping may describe which objects are persistent and how they are mapped into a traditional relational database, etc.
- **Performance view:** Sometimes a configuration must perform within critical limits, it might be time or space that it the limiting factor. The performance of the entities might be critical, and one needs to document average or worst-case consumption.

The realisation view is also referred to as the physical model (opposed to the logical model).

³ Notice that state machines are for a single class only, and could therefore be part of the entity manual. However, we want the entity manual to be extracted from an annotated source code, and it is not clear that it is practical to specify the state machine in a source code comment.

One of the difficult problems in writing the reference document for the software architecture is the requirement that it should be complete – complete in relation to what? One of the most attractive aspects of the “2+1” approach is that it allows us to define the completeness in relation to requirements. Each configuration addresses a number of requirements, and the architecture documentation is complete when all requirements (both customer and internal requirements) are covered.

It is also worth to notice that the relevance of a view depends on the abstraction level. E.g. at a detailed level, the entire realisation view might be implicit from the hierarchical structure of the system. Our recommendation is to really have good reasons to eliminate one of the three main views, conceptual, realisation or requirement, but feel free in which of the sub-views are used. The choice of sub-views depends on both the abstraction level and on the size and complexity of the system/framework to be documented.

Appendix B. contains a comparison between our “2+1” model and the original “4+1” proposed by Kruchten.

6.2.3 Hierarchical Design Descriptions

The configurations should be organised hierarchically, which should be reflected in the documentation. The top-level configuration should give an overview of the architecture. More detailed descriptions should be given for each sub-configuration, and sub-configurations should then be organised into new sub-configurations, etc. At present, it is difficult to advice on this part, since we have only little experience.

Figure 4 shows a proposal for concrete document types and their relations based on the information presented in the previous sections. The figure shows also an example of how a concrete documentation may be organized. Solid lines represent hierarchical decomposition, with the lower nodes being subordinate to the higher nodes. In this concrete example, the software consists of five classes in two packages, each class with one attribute and two methods. The architecture documentation consists of three top-level configurations, of which the middle configuration consists of two configurations etc. (Please note that it is unrealistic to have this many configurations for so few software entities). The special symbol within each configuration indicates that the configuration is documented by using the “2+1” principle discussed in the previous section.

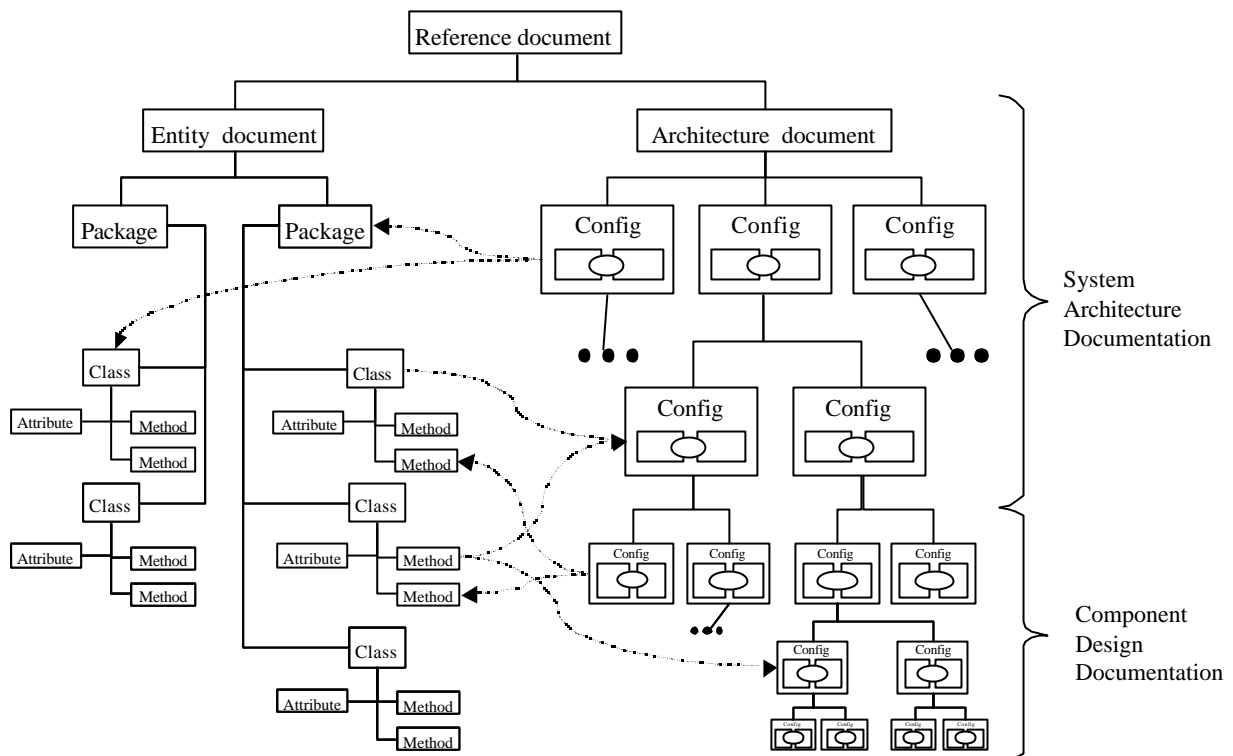


Figure 4. Documentation example

Figure 4 also illustrates the possible cross-references between the configuration documentation and the entity documentation. A given configuration can refer to several sections in the entity documentation, for instance to avoid repetition of parameters, pre-conditions etc. A given entity can play several configuration roles, thus relieving the entity-documenter of explaining those aspects in detail.

In the working group we have at numerous occasions discussed the difference or similarity between, what is sometimes referred to as system architecture documentation and component design documentation. The argument for a distinction is that there are important decisions to document for the entire system, and it is advisable to locate them explicitly in a top-level architecture document. Arguments to the contrary are that, at any level of a system, there will be important design decisions to be made for the entire subsystem. Hence there are no structural difference between the kind of documentation required for a full system and that required for a subsystem. The compromise represented in the figure is that some of the top-level configuration descriptions are referred to as system architecture documentation, while the rest are referred to as component design documentation. In our opinion it is a fruitful distinction, but we also find that it is difficult to set up precise rules as to when a configuration is part of the system architecture and when it is part of the component design.

7. Documentation in the Development Process

It is an overall goal for system documentation that it should be useful for both software developers and maintainers of software systems.

7.1 Waterfall Development

The traditional waterfall development process is usually document-driven, and the milestone of each development phase has normally been a reviewed and approved document for a particular phase. Document examples are requirement specification documents, system design documents etc. Another characteristic would be that the output of each development phase has been used as input for the next phase. The waterfall model approach has the advantage that the documentation is produced during the development process because it is required for finalising a given milestone and used as input to the following phase. Verification is normally done through a more or less formal review or inspection of the document.

The standard critique against the waterfall model is that all phases are executed sequentially, feedback comes very late before the implementation and integration test prove, or more normally find, flaws, errors and inefficiencies in the design. When feedback occurs, the design should be revised and all the documents should be updated, and it may then be necessary to carry out a new review. As these deficiencies are detected very late in the development process (if the method is followed strictly), corrections are made in the code only, without the corresponding document updates (because it becomes too expensive to return to earlier phases of the development). As a consequence, the documentation cannot be used for maintenance, as it does not correspond to the running code any more. It should be noted that in practice the waterfall model is not strictly followed because of these reasons.

7.2 Iterative Development

Modern development methods suggest an iterative and incremental development process. An example hereof is the *Unified Software Development Process* [Jacobson99] in which iterations are planned and driven by the identified Use Cases. This approach has been tested in the COT/Case 2 pilot projects where it was an effective and more flexible way of developing software compared to the traditional waterfall model.

7.2.1 Documentation Process

The question is how the documentation is produced in an iterative development process and how this process changes the way of working with documentation.

The documentation structure for the project must be defined before starting development—because documentation plays a very active role in the process. The documentation structure must be supported with concrete document templates for the different artefacts—this could be standard templates or templates adjusted for the specific project.

However, the documentation structure is also part of the iterative development, hence the structure of the documentation can also be made subject for redesign.

If the requirement specification is specified with the Use Case technique [Jacobson99] these Use Cases are used as the basis for planning the iterations. Detailed guidelines for planning this iteration are described in [Jacobson99].

The Use Cases selected for the first iteration must cover some of the architectural significant Use Cases or the main scenarios hereof. Object-oriented analysis and design models are developed for these Use Cases and the different document templates are filled with the information required to describe the state of the project. An (informal) review could verify the architectural ideas before implementing the actual iteration.

The different documents are only partly finished with the documentation for a given iteration and subsequently enhanced with information from following iterations. This produces a very dynamic document, which is updated after the completion of each iteration. In this way a given document plays a very active role in the development process as it is updated on an ongoing basis to correspond exactly to the running code.

In this iterative process, the important milestones are designed and implemented parts of the system where the errors have been corrected. The approval of a milestone can include the updating of documentation, which could be either formally or informally reviewed in a normal review process.

In the next iteration, the documents are enlarged by the introduction of new development information for the corresponding Use Cases for this iteration.

7.2.2 “2+1” View Model – as Process Driver

The concept behind the “2+1” view model with different development views can also be used as a driver in the process for developing the system architecture.

One of the first tasks in a project is to select the number of relevant views for the given project. As an example, a project specified to run on one computer will not have a deployment view, and a sequential programme will not have a process view. The next task is to define or adjust the document structure and the corresponding document templates to the needs of the actual project.

After selecting one or more Use Cases for the actual iteration, the corresponding object-oriented models are developed and documented in the logical view.

The next step is to develop the first version of the other views based on the information in the logical view. The executable- and source code view is developed for defining a layered model of source code components. The deployment view is developed for defining the actual computers/processors and hardware devices. If the system can be installed in different configurations this is taken into account and documented. If the Use Cases have concurrent aspects a first cut of the process view is developed.

The logical view can in many situations be adjusted or updated based on the knowledge gained by looking at the other views with the advantage of an easier traceability from the conceptual view to the other views.

By looking at the different aspects of each view, the architecture of the programme will be formed stepwise as well as the corresponding architecture documentation. The actual development iteration can now be designed in detail, implemented and tested.

The next iteration will proceed in the same way with selection of some new Use Cases and developing or updating the different views based on these new requirements. By carefully selection of the Use Cases for the first number of iterations the architecture of the programme will begin to stabilise.

The template for an architecture document could have sections corresponding to the different views as each of these views have different stakeholders. In this way a step in the development process could be to develop a given view and fill out the corresponding section in the architecture document template.

8. Tool Support for Documentation

A number of artefacts are produced in the development process. They may be textual, e.g. natural language descriptions or source-code or graphical descriptions like Use Cases, class diagrams, sequence diagrams, state-transition diagrams but they are not necessarily limited to texts and diagrams. In the pilot projects we have also used white board snapshots, i.e. photographic pictures of the white board, to capture more or less informal information. Descriptions are produced using different tools, e.g. a CASE tool for producing diagrams, and text editors for producing source code. The source code may be generated from the models in the CASE tool, or the diagrams may be generated from the source code or a mixture of both. Initially the primary purpose of producing the development artefacts was to support the analysis, design and implementation processes.

However, many of the same artefacts are also used in the system documentation. This chapter discusses tool support for producing the kind of documentation discussed so far in the report.

In an iterative and incremental development process, not only the final system but also the intermediate states of the system needs to be documented. Tools should support the documentation needs in these intermediate states as well as the final documentation.

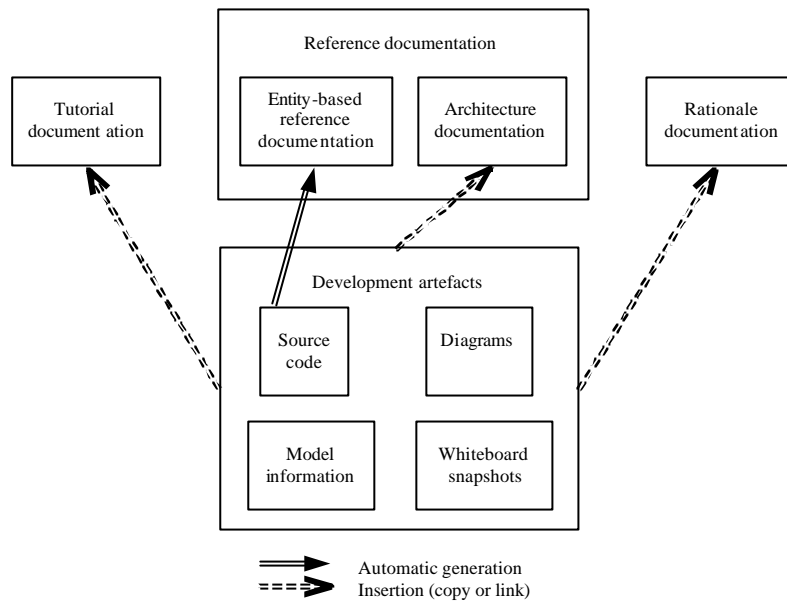


Figure 5. Documentation types and examples of development artefacts

In Figure 5, the types of documentation (that this report focuses on) are illustrated in the upper part of the diagram: tutorial documentation, reference documentation and rationale documentation. The lower part of the diagram illustrates some of the development artefacts: source code, diagrams, model information, and whiteboard snapshots. Model information refers to the model representation in a tool, typically a CASE tool. This information may be redundant, but it can also be information, which is not contained in the source code or the diagrams. One example is the Use Case descriptions produced in a CASE tool or in a word processor. Depending on the tools used, the model information may be non-existent. In the pilot projects, two different approaches were used for the responsibility descriptions. In some cases they were entered in the CASE tool and in other cases directly in the source code.

The arrows indicate the possible relationships between the documentation types and the development artefacts:

- Automatic generation of the entity-based reference documentation.
- Insertion (copy or link). Use of development artefacts in the documentation.

The two types of relationships will be discussed below.

8.1 Automatic Generation of the Entity-based Reference Documentation

The entity-based reference documentation can be (and should be) tool generated, based on the annotations within the source code of the program. The documentation generator will often be able to analyse the overall aspects of the programming language and provide appropriate information on inheritance, types of parameters etc., thus avoiding that it is written twice (i.e. in the program and in the documentation comments).

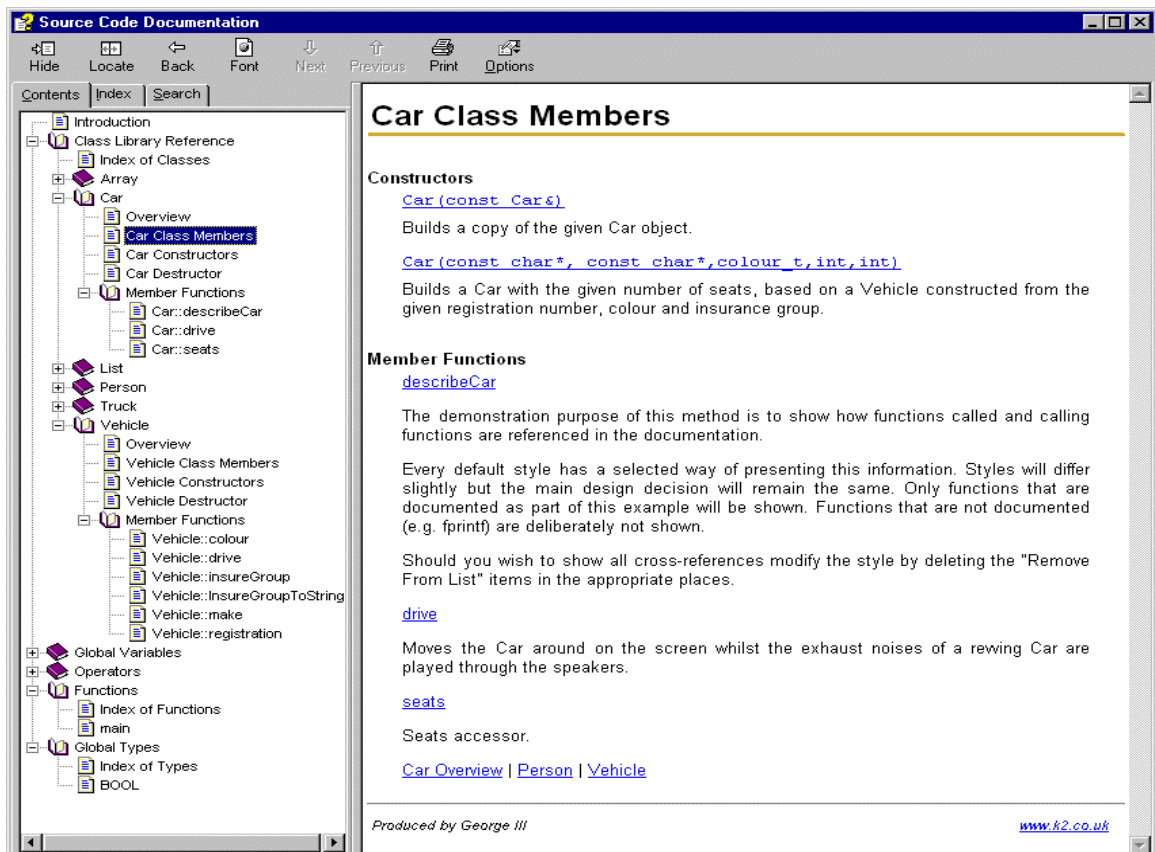


Figure 6. A screen dump from George

Tools like [JavaDoc] for Java, [George] and [BumbleBee] for C++ can generate the entity-based documentation. These tools are based on the *proximity* principle, which means that the code and its documentation are so closely connected that the programmer is likely to update the documentation when the program is changed. When the source code is changed, the entity-based reference documentation can automatically be regenerated. Proximity does not *solve* the problem of consistency, but experience shows that it *helps* in keeping the entity-based documentation consistent with the implementation.

A screen dump from George is shown in Figure 6. In the example, the constructors and member functions and their responsibilities are shown for a class called 'Car'. In addi-

tion there are links to an overview description of the 'Car' and to two related classes: 'Person' and 'Vehicle'.

8.2 Use of Development Artefacts in Documentation

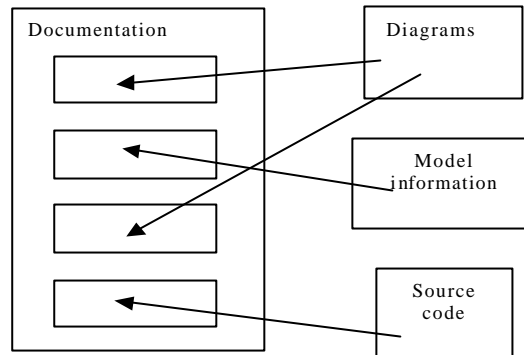


Figure 7. Use of development artefacts in the documentation

Parts of the different development artefacts are likely to be used in the tutorial, the architecture documentation and the rationale. Tool support for automatic insertion of these descriptions in the documents is desirable for at least two reasons:

- It facilitates the collection of all the different descriptions into one or more complete documents,
- New versions of documentation have to be produced repeatedly in the development process, especially in an iterative development process. Automatically inserting updated information in the documents makes it easier to spot inconsistent explanations during review.

Some CASE tools provide report generators, which allow the users to produce user-defined documents that combine textual descriptions with development artefacts. Examples are model information, diagrams and source code. After the report structure has been defined, the collection of relevant development artefacts is simply a matter of pressing a button. However, it is important to control the report generation in detail with regard to what kind of information should be included.

In the pilot projects, the best example of a report generator that used development artefacts in the documentation was [QualiWare]. However, the word processor used in QualiWare is an internal tool and the report is therefore not easily integrated with other documents. HTML generation is another possibility but we have no experience with this facility.

Rational SoDA (Software Documentation Automation) [SoDA] is a tool that adds document generation to Microsoft Word or Adobe FrameMaker+SGML, and enables information to be extracted from multiple information sources such as Rational Rose. We have no experience with SoDA, but it could be interesting to investigate further.

There are other ways of integrating text documentation with information from the CASE tool or development system. Modern document architecture technologies like [Microsoft COM] enables documents, or parts of documents, to be embedded in the document from other applications. If, for example, the word processor and the CASE tool support [ActiveX], the diagrams or the part of the source code could be embedded in the text document. If the documentation is written in HTML, different scripting languages like SGI scripts or JavaScript could be used to insert diagrams or parts of the source code. In both instances, additional tool support is desirable.

In general, the optimal solution would be to write documents in a standard word processor with inclusion links to diagrams, model information and code. Inclusion links are to be understood as hypertext links where the destination is inserted in the document. By doing so, the documents could be updated automatically when e.g. the code or the diagrams are changed. Another challenge would be to support the editing of inserted descriptions, which would then be followed by automatic updating at the source.

Experimenting with all these aspects could be interesting. One experiment could be to use the Devise hypermedia, which is an open hypermedia system to integrate a word processor like Word with a CASE tool or other tools of a development system.

8.3 Consistency

Above we have seen examples of how diagrams, model information and source code can be inserted in the different documents. It is important that the inserted descriptions are consistent, both mutually and with the enclosing text.

8.3.1 Consistency between Development Artefacts and Documentation

The documents may need to be updated when one of the development artefacts is changed. Manual updating, e.g. using copy-paste, should be avoided because it is a error-prone process where one easily forget to copy exactly what is needed. As described above, automated updating of documents may be carried out differently, but it is important that the writers of the documents are aware of the changes, since the explanatory text may also need to be updated. Such inconsistencies are easier to spot though, than inconsistencies between an explanation and an external (to the document) artefact. The dependencies between the descriptions must therefore be explicit and tool-supported. For example if a class diagram is changed, the writers of the documents, using this diagram should be notified. The writing the documentation will sometimes result in changes to the code and related descriptions.

8.3.2 Consistency between Documentation Types

Neither the architecture documentation nor the tutorial has the same proximity to the implementation as the entity-based reference documentation. As a result, they are inherently more prone to inconsistency. Since we also want to avoid redundant descriptions, it is particularly important to avoid repetition of what is described in the entity-based reference documentation. A tool that allows easy cross-referencing to the entity manual

is useful, as it makes it easier for the documenter to avoid redundancies. Currently we have no recommendations on how best to address this inconsistency problem.

8.3.3 Consistency between Development Artefacts

An important requirement for the CASE tool employed is to ensure mutual consistency between the inserted descriptions. In the pilot projects we have good experience with reverse engineering using [Rhapsody] and [WithClass]. Reverse engineering of the source code produces class diagrams that are consistent with the code. Class diagrams illustrate the static properties of the system. Sequence diagrams are an example of diagrams that illustrate the dynamic aspects of a system. In the COT pilot projects, a tool [COT/2-27] has been developed for automatic generation of sequence diagrams for Java and C++. The sequence diagrams are produced during execution of the application and this has been a valuable tool in the documentation, verification and testing processes.

We have experienced one consistency problem in the COT pilot projects with regard to responsibility descriptions: Not all CASE tools preserve the annotations in the reverse engineering process. When using round-trip engineering tools, it is important that all aspects of each representation (i.e. in both CASE tool, and in source code) are preserved when transforming from one representation to the other.

8.4 Links

In general we recommend that documentation is online. Online documentation can provide much richer navigation facilities that make it easier to home in on the relevant aspect, especially in large bodies of documentation. However, some prefer to read documentation on paper, especially those aspects of the documentation that are not reference documentation, e.g. the tutorial. We do acknowledge that paper based documents can contain cross references, but if they are used extensively, the reader soon find them tedious.

But there are advantages in an online tutorial, namely that it can provide links to the reference documentation. This saves the tutorial writer from repeating what is already in the reference documentation and at the same time teaches the programmer how to use the reference documentation. If the tutorial is online, the advanced descriptions can also be moved from the entity-based reference documentation into the tutorial, and have links from the entity based documentation to the tutorial (all the HTML generating document extractors, we have seen, allow the annotation comments to be full HTML, including anchors).

The same advantages and possibilities for writing tutorials apply to the architecture part of the reference documentation. Descriptions of the architecture documentation can make references to the entity-based reference documentation and the entity-based part can have reference to the architecture part where appropriate, thus avoiding redundancy.

Finally, links in both directions between all the documentation and the development artefacts could be very useful. It will not only help in reading the documentation, but also in understanding the implementation and in the maintenance of the system.

If the documentation is written in HTML, the linking facilities of HTML can be used. Many word processors also support hypertext links to some extent. The open hypermedia system Webwise provides an alternative.

9. Use of Videos as Documentation Media

In this chapter we will present a general introduction to the use of videos as documentation together with concrete experiences from Danfoss Drives A/S.

An article of Tom DeMarco [DeMarco90], who made some experiments in a fast growing company, inspired the concept of using video recordings as documentation media. The idea was to give new employees a chance to increase productivity by using video recordings in replacement of text-based system documentation. The motivation in this company to write documents on a working system was low; therefore a quick and inexpensive solution was to introduce video recordings. The result was very positively received, both among the “old” designers telling about the main architecture and the new employees acquiring the knowledge.

Another inspirational source was a lecture held by Alistair Cockburn at a COT seminar where he compared efficiencies among different communication forms. He claims that communicating through paper is the slowest and least effective medium, whereas video is the highest achievable communication medium.

Based on these positions, Danfoss Drives decided to proceed with their own experiments, the situation being that a prototype (non-documented SW) was adopted into a real project with a strict schedule.

Four video session attempts have been made so far—where a session lasts from 10 minutes to 1 hour. The experiment had 3 purposes:

1. To inform/teach team colleagues who were not part of the pilot project.
2. To archive this information in order to facilitate a subsequent transfer into relevant documents.
3. The maintainer or reuser could use it later when a speedy introduction to the software is required (tutorial).

Although based on limited experience, some practical conclusions can be made—first of all, when presenting a topic it is important to use the whiteboard as much as possible and to minimize the use of overhead projectors. An audience should be present during the session and play the role of “error filter” and partly as a “question generator”. The tapes should be stored in some sort of library after the session and, in order to maintain this library, a person responsible for the video recordings should be assigned.

In order to make the video recordings generally available even for a foreign colleague, it should be digitised/transferred to a server or at least a CD-ROM. This digitising the video makes it more useable, as most people will not spend hours in a meeting room with TV and video. In order to make it generally available a video grabber is required that can store onto a server or a CD-ROM/DVD. However, since this process will re-

duce picture quality, it requires that the sessions have large illustrative drawings/texts or alternatively that detailed drawings can be printed out separately. Whether the present technology is suitable is still unclear at this point—maybe we have to wait until better technology like DVD is more mature. It is worth to notice that there are currently a plethora of different file formats and video formats, and what is best technology today is hopeless tomorrow.

In general, we believe that video is a good medium to inform about dynamic behaviour and design choices, since these aspects usually are difficult to write in a document. Another guideline is to use video recordings to capture the original architecture design decisions made by the leading architect.

We see video primarily as a supplementary technique to produce tutorials and rationale.

10. Conclusions and Recommendations

The purpose of this document has been to provide guidelines for writing documentation of OO software. We recommend the following overall guidelines for handling documentation:

Recommendations regarding the development process:

We assume an iterative strategy where each iteration consists of analysis, design, implementation and evaluation. At the end of an iteration, the evaluation phase evaluates the requirements for the iteration and new requirements for the next iteration are defined or selected. These recommendations are similar to those for testing OO systems [COT-2-43].

- Documentation should be an integrated part of the development process.
- The architecture of the software should be reflected in the architecture documentation.
- The structure of the architecture and architecture documentation should be developed as part of the software development process.
- The requirements for the next iteration should specify the parts of the architecture documentation to be delivered.
- Most (all) design documents (class diagrams, state diagrams, etc.) produced during the development process should be included in the architecture documentation.
- Pictures of whiteboard drawings etc. produced during the initial phases are also candidates to be included in the architecture and rationale documentation.

Recommendations regarding the form of the documentation:

The documentation should be separated into:

- Tutorial documentation.
- Rationale documentation.
- Reference documentation.
 - Entity-based reference documentation.
 - Architecture documentation.

Recommendations regarding tools and media:

- Proper tools should be selected in order to provide consistency between the various parts of the documentation and to ease construction and maintenance.
- The documentation should be generally available.
- To ease availability, online documentation via the Web should be considered.
- Use of audio and video should be considered. This includes the use of multimedia documents.

Recommendations regarding tutorial documentation:

It is not possible to state absolute requirements since the volume of tutorial documentation required depends on the importance, complexity and number of software users (i.e. developers)

- Concrete examples are in most cases mandatory.
- A FAQ is in most cases mandatory.
- In the case of software frameworks, one of 1) Cookbook, 2) Textbook or 3) Pattern language description is mandatory in most cases.
- A guided construction tour is optional but often a good idea.

Recommendations regarding the rationale:

As for the tutorial documentation it is not possible to state absolute requirements about the volume and content of rationale documentation.

- It should be considered to write a rationale for central and critical software elements.
- The rationale may be used to document design decisions and trace original and new requirements.

Recommendations regarding the architecture documentation:

As stated previously, architecture documentation is probably the most difficult and complicated part of the documentation to understand. Again it is difficult to state absolute requirements for the architecture documentation.

- Architecture documentation may not be required for simple library packages.
- Architecture documentation is required for central and complex frameworks, application packages and applications.
- The ideas in the "2+1" view architecture model are recommended, with the main views as proposed in Figure 3 (i.e. Conceptual, Realisation and Use Case view).

Recommendations regarding the entity-based reference documentation:

- Entity-based documentation is a must.
- Entity-based documentation should be extracted from the (source) code.

The above recommendations are based on experience documented in the literature and the authors' experience from previous projects, including COT. We are, however, not aware of any project that has followed the guidelines exactly as described above. We hope to be able to test the above guidelines in forthcoming projects.

11. References

- [ActiveX]
<http://www.microsoft.com/>
- [Ada79a]
Preliminary ADA Reference Manual, Sigplan Notices, Vol. 14, No. 6, June 1979, Part A.
- [Ada79b]
Rationale for the Design of the ADA Programming Language, Sigplan Notices, Vol. 14, No. 6, June 1979, Part B.
- [Bell & Schmidt99]
A.E. Bell, R. W. Schmidt: *UMLoquent Expression of AWACS Software Design*, Comm. of the ACM, Oct. 1999, Vol. 42, No. 10, (55-61).
- [Birtwistle, 1979]
G. M. Birtwistle. *A System for Discrete Event Modelling in Simula*. Macmillian Press, 1979.
- [BumbleBee]
<http://www.bbeesoft.com>
- [COT/2-27]
C.J. Andersen: *Automatic Generation of UML Sequence Diagrams from Java and C++ code*. Is planned to be a public COT document later in year 2000.
- [COT/2-43]
M.Caspersens, Ole Lehrmann Madsen og Stefan Helleman Skov
Testing of object-oriented systems.
- [Gamma95]
E.Gamma, R.Helms, R.Johnson, J.Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley 1995.
- [George]
<http://www.k2.co.uk>
- [IEEE std 1016-1998]
IEEE Recommended Practice for Software Design Descriptions (SDD). IEEE standard 1016-1998
- [Jaaksi99]
A.Jaaksi, JM.Aalto, A.Aalto, K.Vättö, *Tried & True Object Development Industry-Proven Approaches with UML*, Cambridge University Press 1999
- [Jacobson99]
I.Jacobson, G.Booch, J.Rumbaugh, *The Unified Software Development Process*, Addison-Wesley 1999
- [JavaDoc]
<http://java.sun.com/products/jdk/javadoc/>
- [Jensen & Wirth75]
K. Jensen, N. Wirth: *Pascal User Manual and Report – Second edition*, Springer-Verlag, 1975.
- [Johnson92]
R. E. Johnson. *Documenting Frameworks using Patterns*. In OOPSLA'92 Proceedings, 1992.

- [Kruchten95]
P. Kruchten: *The 4+1 View of Architecture*. IEEE Software 12,6 (Nov. 1995), 45-50.
- [Kruchten98]
P. Kruchten: *The Rational Unified Process*, Addison Wesley 1998, p83-85.
- [Meyer97]
Bertrand Meyer: *Object-Oriented Software Construction, second edition*
Prentice Hall, 1997.
- [Microsoft Com]
<http://www.microsoft.com/com/>
- [MjII-MIA-90-1]
J. L. Knudsen et al: *Documenting Object-Oriented Systems*. Mjølner project II report, Mjølner Informatics, 1990.
- [Rhapsody]
<http://www.ilogix.com>
- [RUP-SAD-artefact]
Rational Unified Process Tool: Artefact for a Software Architecture Document.
- [SoDA]
<http://www.rational.com/products/soda>
- [T.DeMarco90]
Tom DeMarco, Curt Geertgens,
Use of Video for Programme Documentation (experience report), ICSE 1990, p. 126-128.
- [QualiWare]
<http://www.qualiware.dk/>
- [UML97]
Unified Modelling Language, Ver. 1.3 OMG standard nov. 1997.
<http://www.omg.org> and <http://www.rational.com>
- [WithClass]
<http://www.microgold.com>

A. Rationale for entity documentation

This section describes some of the sources used for establishing the recommendations and the document layout described in section 6.1. To read this appendix one must have read the IEEE standard on documentation [IEEE-std1016-1998], and the report [MJII-MIA-90-1]. The section contains background information but can be skipped without loss of understanding the remainder of the document.

The list of information aspects that the programmer must add in the source code as documentation is based on an analysis of what have been proposed in the IEEE standard on documentation [IEEE-std1016-1998], and on the experiences in documentation in the Mjølner project [MJII-MIA-90-1].

19 different categories have been proposed in the above mentioned literature, and when we add the category “responsibility” we have a total of 20. Not all categories need to go into the source code, and we categorize these 20 as follows:

Aspects to be extracted from the source code itself by a tool:

We believe that a tool can extract the information below from the source code itself, and hence the programmer need not repeat this information in the documentation.

- Identity [IEEE std 1016-1998]
- Operations: Partly BETA specific [MJII-MIA-90-1]
- Structural information [MJII-MIA-90-1] – language specific
- Type [IEEE std 1016-1998]

Aspects to be extracted from programmer-provided comments in the source code:

We believe that the best place for the programmer to provide this documentation is to write it in-lined in the code, and then have a tool (probably the same tool as mentioned above) extract the comments that produce the documentation outlined in the previous section in Figure 2.

- Description: extended and one line [MJII-MIA-90-1].
- Categorization: Type from IEEE [MJII-MIA-90-1] Same as IEEE type
- Usage: more type e.g. abstract class [MJII-MIA-90-1] Same as IEEE type
- Intentions: frameworks only [MJII-MIA-90-1]
- Assertions: pre/post conditions, invariants [MJII-MIA-90-1]
- Purpose: why [IEEE std 1016-1998]
- Function: what [IEEE std 1016-1998]
- Type [IEEE std 1016-1998]
- Responsibility

Aspects belonging to the architecture documentation:

These aspects of documentation do not belong in the entity-based documentation, but in what we refer to as architecture documentation.

- Configuration: relationship between objects [MJII-MIA-90-1]
- Resources: dependency of externals [IEEE std 1016-1998]
- Subordinates: parts [IEEE std 1016-1998]

- Dependencies [IEEE std 1016-1998]

Aspect belonging to the tutorial documentation:

In order to avoid redundancy, we categorize the following aspect as something that should be put into the tutorial.

- Terminology: short list of concepts [MJII-MIA-90-1]

Aspect not relevant for object-oriented systems:

In the IEEE standard, some of the aspects mentioned are not relevant in object-oriented programs. They will be described elsewhere due to the different structure of object-oriented programs.

- Processing: how [IEEE std 1016-1998]
- Data: how [IEEE std 1016-1998]
- Interface: [IEEE std 1016-1998] – Because the entity manual is the interface description of the entity!

B. Comparison of “2+1” with Kruchten

In this appendix, the recommendations described in the previous sections are compared to the original “4+1” model proposal by Kruchten in [Kruchten95] and [Kruchten98]. Below it will be an advantage if the reader is familiar with [Kruchten95] and/or the experience report in [Bell&Schmidt99]. As mentioned, the main rationale behind the “4+1” view is that a configuration should be documented by means of different *views*.

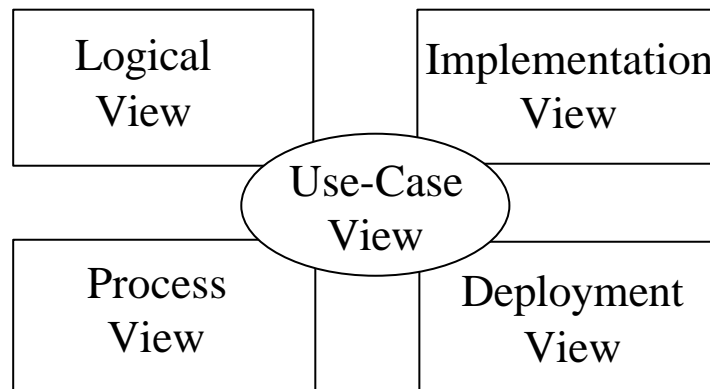


Figure 8. The “4+1” view model

Phillipe Kruchten proposes in [Kruchten98] the following 4 views:

- **Logical View:**
It basically corresponds to the conceptual view described in “2+1”
- **Process View:**
It basically corresponds to the realization view described in “2+1”. According to Kruchten this view addresses the concurrent aspects of the system at runtime i.e. tasks, threads or processes and their interactions. According to this the process view

is more an implementation view than a modelling view as it in addition to logical processes also contains implementation processes or tasks.

Information about active objects can also be documented in the logical view with the UML stereotype mechanism.

- **Implementation View:**

It seems to correspond to the source code and executable views described in “2+1”. This view describes the organisation of static software packages in the development environment in terms of packaging and SW layers and in terms of configuration management. Examples of information concerning this view would be source code files, data files, components and executables. UML’s component diagrams are used in this view by employing symbols for components and optional packages.

- **Deployment View:**

The deployment view of Kruchten seems to correspond to the deployment view described in “2+1”. Kruchten also mentions the need for a **data view**.

In addition there is the “+1” view:

- **Use Case View:**

There is some variation in whether the “+1” view is referred to as Use Case view or scenario view. This view contains a few key scenarios or Use Cases. These scenarios act as an illustration of how the other view works.

As mentioned, Kruchten has split the conceptual view into a **logical view** and a **process view**. At this point we are uncertain about whether or not it is useful to distinguish between the two views in this context. There is, however, no doubt that both aspects should be described. In the end it is mainly a matter of organisation.

The main differences between our “2+1” model and “4+1” is that we view Kruchtens views as sub-views of two more general views, conceptual and realisation view. Also, where Kruchten addresses views primarily for systems, we propose to apply the idea uniformly to all configurations in the system. Hence, where Kruchten proposes that the views to include in the documentation is decided on a per project basis, we propose to select sub-views on a per configuration basis. Primarily because configurations are used on several abstraction levels.

Another source of inspiration is [Jaksi99] which operatates with a “3+1” view mode: logical view, development view, run-time view and th “+1” scenario view. The run-time view corresponds in this model to a combination of Kruchtens proces and deployment view.