Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
and
LIFIA-UNLP - Argentina
1999

Patterns and frameworks for framework documentation

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange
project funded by the European Community)

By: Ilse Dierickx

Promotor: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promotor: Gustavo Rossi (LIFIA)

Abstract

The last decade, the object-oriented paradigm has become of great importance and one of the growing topics in this field, is certainly framework technology. A framework makes it possible to develop applications belonging to the same application domain in less time, because frameworks represent a general design for these applications. So, an application engineer only needs to be concerned about the variations of the application under construction. When an application engineer builds a new application using a framework, he needs a thorough understanding of that framework. In the framework understanding process, good documentation is of undeniable importance.

In this dissertation we propose a new strategy to document frameworks. During the lifetime of a framework, a lot of different people come into contact with it. These people have their own documentation requirements, which leads to different documentation techniques and patterns. We propose a framework (DocFramework) that combines the existing documentation techniques to form a generic documentation structure. On top of this documentation structure we define different views for each type of framework user. These views will be expressed as hypermedia topologies and configurations which guides the framework user through the framework documentation or understanding process following a certain navigational path. Another advantage is that the DocFramework will be used by all framework users during the entire lifecycle of the framework and this way all documentation can be found in one place. We are convinced that the proposed documentation technique will lead to a better framework understanding, which results in better- developed applications.

Acknowledgements

First of all, I wish to thank my advisor Dr. Gustavo Rossi for guiding me through this interesting research process. I also would like to thank him for his hospitality and his support during my stay in Argentina.

I would like to thank Mauricio Sansano for the fruitful discussions we had and for proof-reading my thesis.

I owe a lot of thanks to all the people from Lifia, they gave me a fun place to work. A special thanks goes to Gabriela for all the things she did for us. Especially in the beginning when my non-existing knowledge of Spanish was a barrier to get around in Argentina. Guillermo and Anabella: thanks for being there, for being my friends, for cheering me up in the times I was missing my family a little bit too much.

Next, I would like to thank my proofreaders at the VUB. They read several parts of my thesis and gave some comments on it over e-mail. Thanks to Christian Daems, Bart Wouters, and Marina De Vos. Also thanks to Dr. Koen De Hondt from MediaGeniX.

My stay in Argentina would not have been the same without my roommate Isabel. It's great to have a terrific companion, like Isabel, with you when you are so far away from home. Together with her I did lots of pleasant stuff,

but she was also there to support me when things became more difficult, when time was running out.

I also would like to thank the people that I have left behind at home. Luckily, there exists e-mail, which reduces the distance between my family and me a little bit. I would like to thank them for the encouraging e- mails during this sometimes difficult period. I am extremely thankful to my parents, because they supported me when I decided to continue studying for one more year.

I would like to thank all the people that were involved in the setting up of this career. I think they did a splendid job. Thanks to Dr. Carine Lucas and Wolfgang De Meuter from the VUB in Belgium, and of course Annya Romanczuk from EMN in France. Besides this, I also need to thank Annya for the support we got from her during our stay in France.

I also owe a lot of thanks to Prof Dr. Theo D'hondt. Not only because he was a good teacher for me during the past four years at the VUB, but also because he made it possible for me to follow this master program.

I would like to end this acknowledgement by thanking all my fellow students with whom I spent an incredible time in Nantes. I learned a lot of this master and not only in the classroom !

Ilse Dierickx

Augustus 1999

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

1.1 Intention

When a company decides to build or buy a software product, then this company expects to receive a product of high quality and it also wants the software product delivered as soon as possible and against the lowest possible price. This demanding task forced the software developers to look out for new developing techniques that made it possible to develop applications much faster. Research showed that software reuse is an approach, that is able to deliver the gains in productivity and quality that the software industry needs [MMM95].

In the beginning, only objects and classes were reused, but also design patterns, class libraries and object- oriented frameworks can be reused. The first attempts to reuse were mainly focused on reuse of code. However, reuse of design would be more beneficial in economical terms, since design is more difficult to create and re-create than programming code. This leads us the state-of-the-art in object-oriented reusable assets, i.e. object-oriented frameworks.

Object-Oriented Frameworks

Object-oriented frameworks make it possible to reuse large and abstract designs. An object-oriented framework is a reusable design for an application, in a particular domain, represented by a set of abstract classes and the way they collaborate. Thus, a framework outlines the main architecture for an application to be built in a specific domain. A framework then implies reuse of both code and design. A framework particularly emphasises those parts

of the application domain that will remain stable and the relationships and interactions among those parts. The framework will also point out those parts that are likely to be customised for the actual application under development.

Usually a framework application engineer will customise these parts to achieve the desired behaviour of the application under construction. The framework application engineer must be informed by the framework developer about which parts he has to customise and how these parts interact with the other parts of the framework. Thus, to be able to use a framework, the framework application engineer needs a thorough understanding of the framework's structure. In the understanding process of the framework, documentation plays an undeniable role. Without good documentation the framework will not be used correctly or will even not be used at all.

Importance of Framework Documentation

Framework documentation is not only important to make framework design understandable; other reasons are:

- Frameworks are not fully functional applications. The framework developer designed the framework in such a way that the application engineer only has to fill in the functionality that is typical for the application under development. The framework developer has an idea of how the framework must be reused and he should document his expectations to the reuser to make it possible for him to reuse the framework.
- A framework embodies an abstract design. A framework captures all the application domain knowledge that a framework developer gained. Since a framework is the result of many design iterations, the real world concepts have often been lost in the design. Without appropriate explanation of this design, the framework application engineer would have to learn the application domain all over again.
- Frameworks evolve: A framework application engineer needs to have documentation on how it evolved from one version to another, so that he can update his application to the latest version. The framework engineer also needs to know how the framework evolved over time.
- Framework development is expensive, but this cost can be justified if the framework is used a lot. Framework documentation can be used to promote the framework.
- Documentation is used to reduce the learning curve of the framework.

Framework Documentation Requirements

During the lifetime of a framework a lot of different users come into contact with it. The different audiences have their own documentation requirements. Not all users need to have the same level of detail and they do not have the same level of expertise. Framework documentation should take these differences into account and give the appropriate support to the user, considering his required level of detail and his experience.

Existing framework documentation techniques do not address this problem sufficiently. For example, cookbooks describe the framework in an informal way, which is sufficient for a user who needs to know only the purpose of the framework, but it lacks design information for the advanced user who wants to use the framework in unanticipated ways. Thus, a new documentation strategy is necessary.

Our approach

In this work we present a new approach for framework documentation. It is an approach that gives the different types of users the documentation that fits the user requirements. We identified the different needs of the users and determined which existing documentation approaches would be appropriate to assist them. Since none of the existing approaches can cover all the needs of the user, we propose to combine forces and define, for each user type, a set of techniques that he can use. We present the desired information to the user in a hypermedia way. This allows the user to access the requested information (and its related information) in a fast way.

To achieve this, we propose a framework (DocFramework) that stores the documentation made using existing techniques in a repository and that gives different navigational views for each type of framework user. A navigational view defines how the information can be accessed and how the parts are linked to each other. In this way we can define a navigational path that guides a user through the framework understanding process. The DocFramework is based on another framework, called OONavigator. OONavigator makes it possible to add hypermedia functionality to an object-oriented application without polluting the application model. In our case we can allow browsing between documentation techniques or documentation items, without these techniques being aware of the navigation.

While comparing existing documentation techniques we discovered that some of them are composed of several more basic techniques. The basic techniques can appear in several other complicated techniques in the same form or in a slightly alternated form. We promote reuse of documentation by putting these basic techniques in a separate layer and allow references to them by defining the composed techniques as wrappers [GHJV95] on the

basic techniques.

1.2 Structure of the Dissertation

This dissertation is structured in the following chapters: Frameworks, Framework Documentation, OONavigator Framework, The Documentation Framework, Evaluation and Characteristics of the DocFramework, Future Work and Conclusions.

Chapter two, Frameworks, gives some background information on frameworks. Some definitions of frameworks will be given and we will discuss some important characteristics of frameworks. The development, instantiation, maintenance and evolution of frameworks are described next. An overview of the different kinds of frameworks is given, which is followed by the enumeration of some advantages and disadvantages of frameworks. We conclude this chapter by comparing frameworks against related techniques such as design patterns or class libraries.

Chapter three will be about framework documentation. First some reasons for the importance of framework documentation are mentioned. Next, we will discuss how frameworks are used and who the framework users are and what they expect from framework documentation. We will also discuss what aspects of the framework need to be documented. The chapter closes with an overview of several existing techniques and we will compare them based on several issues.

Since the framework that we will present in this work is based on another framework, called OONavigator, we will spend chapter four on the explanation of this framework. We will also discuss OOHDM, which is a design methodology for developing hypermedia applications.

In chapter five, The Documentation Framework, we will present our approach to framework documentation. Next we will clarify the architecture of the DocFramework.

The characteristics of the presented DocFramework are discussed in chapter six and we will also evaluate our solution and argue about its advantages and disadvantages.

In chapter seven, Future Work, we present some research topics that are related to the work done in this dissertation. We will talk about possible ways to extend the framework. We will also consider how reverse engineering tools can be linked to our repository. A speculation about possible documentation patterns for framework documentation closes this chapter.

In chapter eight, Conclusions, will give an overview of the research that was performed during this thesis.

CHAPTER 2

Frameworks

After several years of software development it became clear that a lot of applications in the same domain are very similar to each other. They differ in some details, but the general structure and design of the program stay the same. As an example we take the programs used in the domain of shopping. The programs in this domain are used to support the activities that go on in a shop, like for example logging the goods that come in and go out or billing the client for the bought goods. Basically all these activities are the same for each type of shop, however an application used in a supermarket will contain some other functionality that is not applicable to small shops, e.g. other paying possibilities (credit card).

Software developers want to take advantage of the similarities between programs because developing a new application for every new company is very expensive and is time demanding. So developers searched for a way to reuse the design and code that is similar for all companies. Frameworks were found to be a solution. Frameworks allow reuse just like regular object oriented techniques, but with frameworks it is possible to reuse large and abstract designs.

But what exactly are frameworks? How can we develop them? Use them? Maintain them? In the following sections we will give a brief overview of the most important framework characteristics.

2.1 Definition

When we try to find an exact definition for a framework in the literature, then we have to conclude that no generally accepted definition exists. The

most widely used definitions are:

> *A framework is a set of classes that embodies an abstract design for solutions to a family of related problems. [JF88]*

This definition tells us that a framework consists out of a set of classes. These classes make up the abstract design of a set of programs that all are located in the same problem domain. For example the "shopping" framework is an abstract design for all future programs used by different kinds of shops. For a lot of problem domains we can create several frameworks, e.g. user interfaces, medical applications, etc.

Another definition would be:

> *A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact. [JR91]*

In this definition we get to know another very important property of frameworks, i.e. that the framework also defines the flow of control among the classes. It defines how the instances of the classes will interact with each other to perform the responsibilities of the framework. Or, in other words a framework contains besides static information (classes and relations among them) also dynamic information (interaction among the instances of the classes).

A last definition that we want to give is the following:

> *A framework helps developers provide solutions for problem domains and better maintain those solutions. It provides a well-designed and thought out infrastructure so that when new pieces are created they can be substituted with minimal impact on the other pieces in the framework. (Nelson, IEEE Computer, 1994, [Inc])*

While the previous definitions described how a framework consists out of interacting objects, this definition talks about how the framework should be used. A framework can be seen as a skeleton that predefines part of the structure and behaviour of future applications built using the skeleton. A developer has to provide new pieces, e.g. definitions of a new function or a new class, to complete the skeleton. The action of providing new pieces is often called "framework instantiation" and the places in the skeleton where to put the new pieces are often called "hot spots" (see section 2.2.1).

2.2 Characteristics

Briefly summarised we can say that a framework is an architectural design used to cover a family of applications in a given domain and that it typically consists out of abstract as well as concrete classes. The abstract classes encode the design of a set of classes and thus embody domain knowledge by making the commonalities between their subclasses explicit. An abstract class is a class that contains a number of abstract methods, i.e. methods without implementation. Template methods are methods that rely on abstract methods or other template methods in their implementation. Finally, there are concrete methods, which have a full implementation that does not rely on abstract methods. We can say that the template methods are the key to reuse. They describe the core behaviour of the class, i.e. the behaviour that is reusable for most of the applications. The abstract methods must be overridden to provide some application specific functionality, while the concrete methods just provide some directly reusable functionality.

For example a "shopping" framework is an abstract design for applications used in a shop. These applications must represent all the tasks that have to be done in a shop, e.g. store and sell groceries, paying, etc. The framework will have classes that represent the groceries, the shopkeeper, the client, etc (figure 2.1).

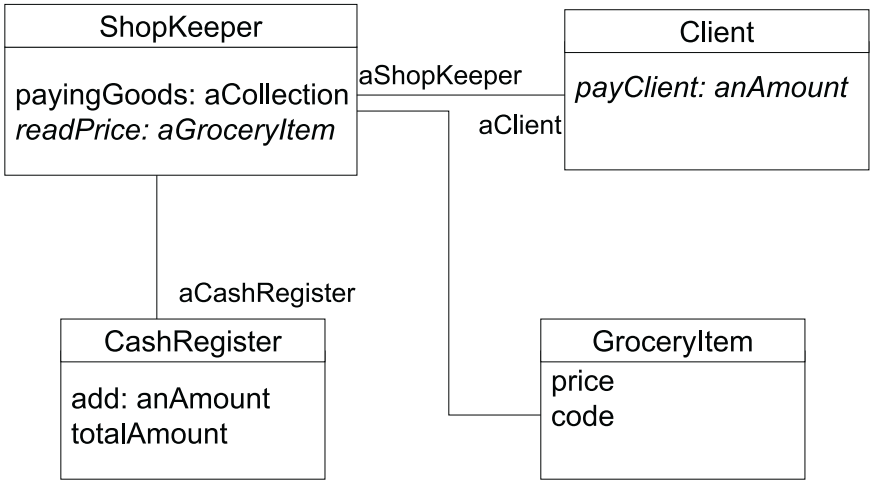

Figure 2.1: Class Diagram Shopping Framework

It defines at the same time how these classes will work together or in terms of the "shopping" framework, how the prices of the groceries are read and added by the shopkeeper and how the client pays for the groceries. The flow of control is fixed by the framework (using template methods), but how it is implemented depends on the requirements of the application

at hand. For example an application based on the "shopping" framework that will be used in a supermarket is different from the application for a small shop. In a supermarket the prices of the products will be read using a scanner, while in a small shop, the shopkeeper will type the prices on the cash register. In the example the flow of control is defined by the template method "payingGoods", i.e. this method decides when to call the abstract methods "readPrice" and "payClient" (figure 2.2, and 2.3). Instantiators

```
Object subclass #ShopKeeper
instanceVariableNames: 'aCashRegister'
        ShopKeeper methodsFor: 'shopping'

        payingGoods: aCollection client: aClient
                aCollection do: [item|
                        aCashRegister add: self readPrice: item].
                amount := aCashRegister totalAmount.
                aClient payClient: amount.

        readPrice: groceryItem
                self subclassResponsibility.
```

Figure 2.2: Implementation of the ShopKeeper Class

```
Object subclass #Client
instanceVariableNames: ' '
        ShopKeeper methodsFor: 'shopping'

        payClient: anAmount to: aShopKeeper
                self subclassResponsibility.
```

Figure 2.3: Implementation of the Client Class

of the framework just need to override these abstract methods to make either an application for a small shop or for a supermarket (figure 2.4, 2.5 and 2.6).

```
ShopKeeper subclass #SmallShopKeeper
instanceVariableNames: ' '
        ShopKeeper methodsFor: 'shopping'

        readPrice: groceryItem
                ^ groceryItem price.
```

Figure 2.4: Implementation of the SmallShopKeeper Class

```
ShopKeeper subclass #SupermarketShopKeeper
instanceVariableNames: ' '
        ShopKeeper methodsFor: 'shopping'

        readPrice: groceryItem
                ^aScanner readCode: groceryItem
```

Figure 2.5: Implementation of the SupermarketShopKeeper Class

```
Client subclass #CreditClient
instanceVariableNames: ' '
        ShopKeeper methodsFor: 'shopping'

        payClient: anAmount to: aShopKeeper
                aShopKeeper receiveCreditCard
```

Figure 2.6: Implemention of the CreditClient Class

In this section we will refer to some important characteristics of frameworks.

2.2.1 Hot spots & Frozen spots:

The functions in the framework design that need to be customised to change the behaviour of the framework are called "hot spots". Providing a new implementation for such a function makes it possible to add variation to the applications that are based upon the framework. The other functions in the framework are called "frozen spots" or kernel. They represent the static parts of the framework. These functions define the flow of control in the application. They will call the hot spots at the right time.

In terms of abstract, template and concrete methods we can say that the template and concrete methods are the frozen spots, while the abstract methods are the hot spots. The behaviour of template methods is inherited by the subclasses of the framework classes. These methods will call at least one template or abstract method that is implemented in the same class or in another class of the framework. The framework developer encodes some of his domain knowledge in the template methods. He knows which steps must be taken to perform one of the framework responsibilities and he transmits this knowledge by calling other methods (abstract as well as concrete) in the template method.

In the "shopping" framework we call the function "payingGoods"[1] a frozen spot because it defines the flow of control of the application and is the same for each instance of the framework. The functions "readPrices"[2] and "payClient"[3], which are called by "payingGoods" are hot spots because for each instance the way the task is performed can be different and so another implementation of the functions is necessary. Look at figures 2.2 to 2.6 to see how these methods are implemented.

2.2.2 Hollywood Principle ("Don't call us, we call you")

Framework-oriented programming requires a new way of thinking. In procedural programs, where a class library is used, it is the program itself that calls functions in the library. The developer of the program makes calls to the library whenever he wants to reuse some code provided in the class library. The flow of control stays mainly in the application and jumps sometimes to the library and returns immediately after a library method is executed (see figure 2.7). In contrast with library based programs the flow of con-

[1] A function that will be called when a customer wants to pay for his goods. This function must contain functionality to read prices as well as some functionality that represents the paying behaviour of customers.

[2] This function represents the task of the shopkeeper to read the prices on the goods. The prices can be read by a scanner and are transmitted to the cash-register or they are simply typed by the shopkeeper.

[3] This function represents the paying activity of the customer. He can pay cash or by credit card.

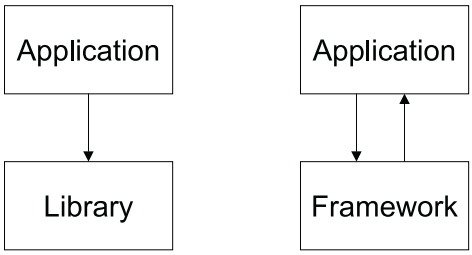

Figure 2.7: Change of control (Hollywood principle)

trol in framework-based programs can go in both directions (see figure 2.7). The framework can also call methods that are provided in the application. An application developer calls a framework whenever he would like to reuse some behaviour that is provided by the framework. The framework takes over the flow of control and decides which methods (framework as well as application methods) to call next.

The change of control is made possible because of the extensive use of dynamic binding, i.e. messages with the same name can be sent to instantiations of different classes and a different method is invoked, depending on the receivers class. The framework involves besides concrete classes also abstract classes. These abstract classes need to be subclassed in order to make a concrete application. Because dynamic binding exists in object oriented languages, a template method does not need to know the concrete subclass that concretises the abstract method to which a call is made. A concrete subclass of an abstract class understands all messages that can be send to the abstract class, so the abstract class can be replaced by the concrete one.

In the "shopping" framework, this reverse of control can be seen in the function "payingGoods". This function determines when the other two functions will be called.

2.3 Framework Life Cycle

Working with frameworks includes several phases. These phases are the development phase performed by framework engineers and the instantiation phase performed by application engineers. Like every software product a framework needs maintenance and sometimes the developers of the framework decide to make a new version of it to remove some errors or add some more abstractions.

First we go deeper in on the issue of developing frameworks, later we will take a look at the other ones.

2.3.1 Framework Development

Framework development differs from that of standard applications. A standalone application solves problems for one project, while a framework wants to be a solution for several projects in the same domain (figure 2.8).

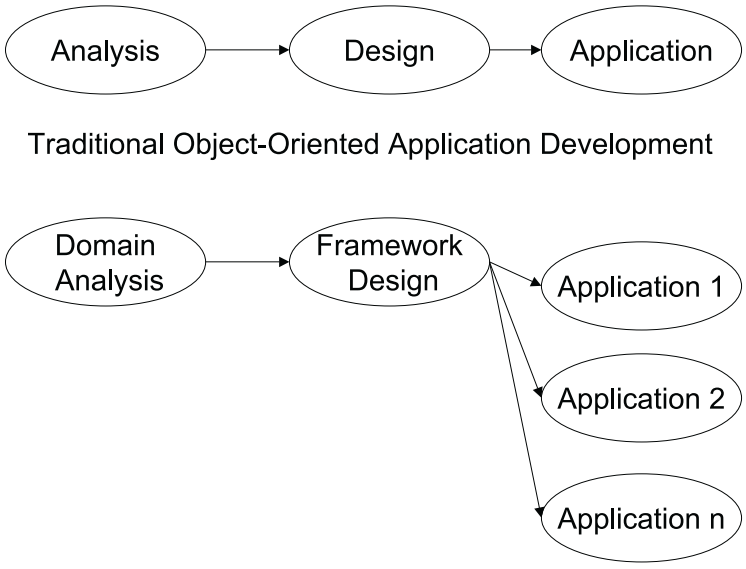

Figure 2.8: Traditional versus Framework-based Application Development [Mat96]

In [CHSV97] they said the following about *framework development*: Framework development requires an extensive domain analysis prior to framework design and that the ultimate goal of framework development is to build - through a small number of iterations - a software architecture that can be turned into a customised application by simply filling in the "hot spots".

The framework development process can be seen as a sequence of steps that must be taken whenever a new framework needs to be developed. These steps were outlined by Mattson ([Mat96]) using a schema depicted in figure 2.9.

- The first step in the development process is an exhaustive analysis of the problem domain. This is performed systematically or through development of one or a few applications in the domain and then some key abstractions are identified.
- The first version of the framework is developed utilising the key abstractions found.

- One or possibly a few applications are developed based on the framework. This is the testing activity of the framework. Testing a framework to see if it is reusable is the same activity as developing an application based on the framework.
- Problems when using the framework in the development of the applications are captured and solved in the next version of the framework.
- After repeating this cycle a number of times the framework has reached an acceptable maturity level and can be released for multi-user reuse in the organisation.

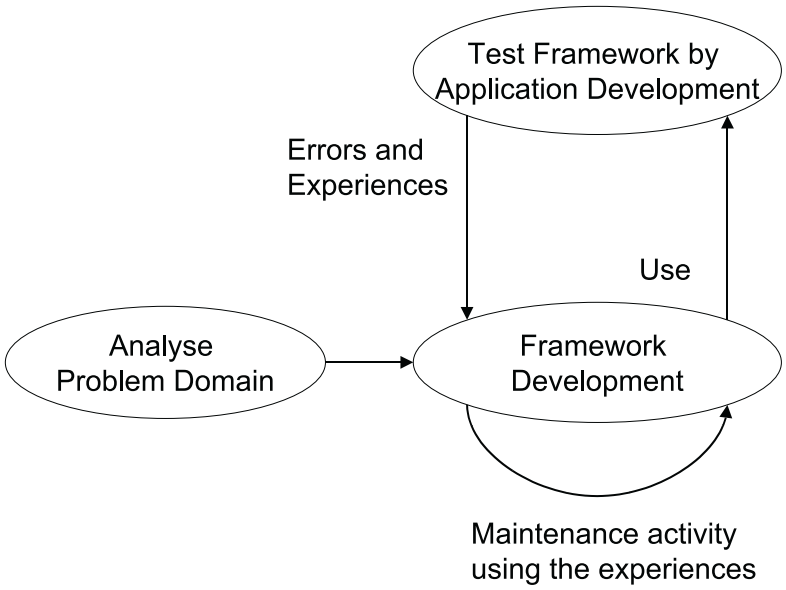

Figure 2.9: The General Framework Development Process [Mat96]

2.3.2 Framework Instantiation

The following phase is the framework instantiation phase and is in fact the process of completing the static variations of a given framework. The application engineer, who is responsible for this phase, needs to know the specific properties of the application under development to be able to fill in the hot spots of the framework. An application engineer needs to be an expert in the application domain to be able to fulfil his[4] task. Depending on the type of framework (white or black box cf. 2.4) other techniques must

[4]In the remainder of this document, wherever "he" and "his" are used, respectively "he or she" and "his or her" are intended.

be used to customise the framework behaviour. More about these different types of frameworks in the section 2.4.

2.3.3 Framework Maintenance and Evolution

In [CHSV97] they make the observation that a co-operation between application and framework engineers should exist. A framework is not finished when the framework developer delivers it to the application engineer. First because a business evolves over time, so the framework has to evolve as well. Second, the framework developer learns from the application engineer how his framework has been instantiated. With this information the developer can decide to make the framework better and more reusable by introducing new abstractions.

Software evolution and maintenance activities are divided into the following categories: [Mat96]

- **Corrective maintenance:** fixing failures to meet system requirements.
- **Perfective maintenance:** improving the performance of the system, maintainability and ease the use in ways that do not violate the requirements.
- **Adaptive maintenance:** evolving the system to meet changing needs.

When perfective maintenance is performed the requirements of the system are not changed, so this maintenance can be done without affecting the applications that are build upon the framework. Refactorings are a well known technique to perform this type of maintenance. Refactorings restructure operations with preservation of its behaviour and without introducing any defects in the program.

When a framework evolves, it is important that the applications built on these frameworks upgrade to that new version. If they do not upgrade, a proliferation of framework versions occurs. Consequently, there will exist different sets of applications that use each their version of the framework. All framework versions still need to be maintained by the framework developers, which results in an undesired increase of maintenance effort. To avoid this from happening it should be made easy for the framework user to switch to the new version. Minimising the impact of the changes or documenting the evolution can help to achieve this goal.

2.4 Different Kinds of Frameworks

Frameworks can be classified by their *scope* or by the way they have to be *customised* [FS97b]

The first classification of frameworks is by *scope*. Depending on the domain for which the framework is created, it is possible to distinguish three groups of frameworks: *system infrastructure frameworks*, *middleware integration frameworks* or *enterprise application frameworks*. For more information about this classification we refer to [FS97b].

Another, more important, classification of frameworks is based on the way an application engineer has to *customise or instantiate* the framework behaviour. White-box and black-box frameworks are the two groups in which frameworks can be classified. So if a framework engineer wants to customise a white-box framework, he has to subclass a base class of the framework (see [JF88]). In doing this it must abide by the internal conventions of its superclasses. Figure 2.10 shows schematically that white-box frameworks must be customised by subclassing some classes at fixed points in the framework.

An application engineer needs to understand the internal structure of a white-box framework to be able to instantiate it. He needs to know how the methods and classes in the framework depend on each other to find out how he can modify the behaviour of the framework. This kind of framework can be difficult to use because it requires a deep understanding of the frameworks design and code, what is almost the same as learning how it is constructed.

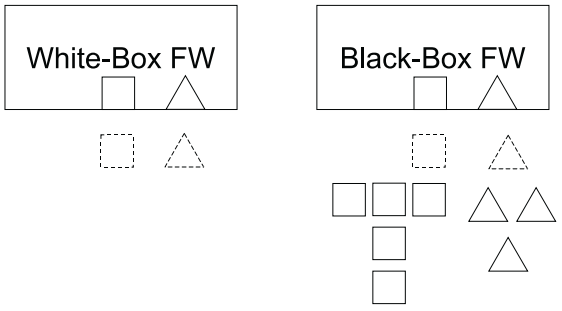

Figure 2.10: A schema of white and black box frameworks

A *black-box framework* can be customised just by composing new objects together. Each component has a well-defined interface that makes it possible to compose them. Black-box frameworks are easier to understand because application engineers only have to learn the external interface of the framework. On the other hand, the number of possible combinations of components is determined by the framework architecture. The framework is less flexible with respect to unforeseen changes. A framework developer

provides a set of components with the basic variations (look at figure 2.10), but users that want to do something slightly different can run into a lot of difficulties.

If we look at the evolution of a framework then we see that it starts as a white-box framework and turns into a black-box framework after it is reused several times, i.e. after the design of the system is better understood. We can conclude that a framework becomes more reusable as the relationships between its parts are defined in terms of a protocol, instead of using inheritance. ([JF88])

2.5 Advantages and Disadvantages

Frameworks represent an abstract design that can be easily and quickly extended to develop new applications. But frameworks also have some disadvantages like they are more difficult to understand. We give here some advantages and disadvantages of using frameworks.

2.5.1 Advantages

One of the most important advantages of frameworks is that they can significantly *increase software quality* and *reduce development effort*. A reason for this increase of quality can be identified in the fact that volatile implementation details are encapsulated behind stable interfaces. It is easier to localise the impact of design and implementation changes, so it is also easier to understand and maintain existing software. The actual development of the framework is rather expensive, therefore this must be seen as an investment from which the future applications, that are build on it, can benefit. The development of these applications will be less expensive then it would have been if they were developed from scratch. The good thing about frameworks is that they, besides enabling of code reuse, promote reuse of generic design, i.e. applications can be developed much faster than before.

Another advantage is that framework based applications can be *faster implemented* because frameworks are written in a particular programming language ([Joh97]). The algorithms and the data structure of the program can be automatically reused by every instantiation of the framework. The implementation of the framework only needs to be extended with the code for the variations. However using a particular programming language to define a framework is also a disadvantage because the framework is then restricted to be used only in that specific programming language.

Applications that are built using a framework are *cheaper to maintain* than independent applications with duplicate code. Maintaining applications implies that one needs to understand the applications code and has to

change it where necessary. Understanding a framework is a one-time effort and results in the understanding of all applications that are built using the framework. Changes to the frameworks code immediately results in adapted applications. The more applications that use a framework, the bigger the savings.

2.5.2 Disadvantages

If a company decides to develop or use a framework, then they have to take into account that the development of a framework is *expensive* and that it takes some time before they can benefit from it. The development of a framework is expensive because a thorough domain analysis must be done before a prototypical framework can be built. This prototype must be used and adapted to make it more abstract and reusable. So building a framework is a long-term investment and it is important that one looks at the cost benefit ratio before starting to build one.

A framework requires a lot of *time to learn it*. It is necessary for the users and new developers to understand the abstract design that was written by the developers of it. Good documentation can reduce this learning curve and is of great importance in the use of frameworks.

The design of a framework is implemented in a *particular programming language*. This property restricts the use of the framework to that language. Although with some effort the design can be reused and implemented in some other language.

2.6 Related Techniques

In the previous sections we explained how to develop and use frameworks. We showed some advantages and disadvantages. But we also want to compare frameworks to some other existing reuse techniques (design patterns and class libraries). [Joh97]

2.6.1 Design Patterns

Patterns reuse design information, i.e. they tell how a problem needs to be solved, they give a solution and a context in which the solution works [FS97b]. An advantage of patterns is that they install a certain vocabulary among software developers.

Most of the time the proposed solution is one that is used several times to solve that problem in other cases, it stood the test of time. We could say that patterns are similar to frameworks because frameworks are also implemented several times. Another similarity is that an application that implements a

framework has to follow the predefined design by the framework just like the design proposed by a pattern has to be followed.

Besides the similarities there are also some differences between frameworks and design patterns. Unlike design patterns a framework also gives a solution implemented in some programming language. This makes patterns more abstract solutions than frameworks. A framework uses patterns to implement the abstract design, so patterns are smaller than frameworks.

2.6.2 Class Libraries

Another similar technique to reuse design is a class library. A class library contains a set of components that can be reused in isolation [Joh97]. To use one of these components, only that class needs to be learned. Opposite to this the whole set of classes of a framework be learned all together before they can be used. The reason for the extensive learning process lies in the fact that the classes in a framework implement a higher level of design, they implement how the classes work together.

An application engineer who uses a class library is in charge of calling the methods in the library. When using a framework, the application engineer has to provide some methods that will be called by the framework, i.e. the framework is in charge of calling methods.

2.7 Summary

In this chapter we gave some answers to the questions: "What are frameworks?", "How to use them?" and "How to maintain them?" We gave some definitions of frameworks and discussed some of their characteristics. Also in this chapter we clarified what the different phases in the life-cycle of a framework are. We saw that frameworks are very interesting to use but also have some disadvantages. One important disadvantage was the learning curve of the framework. In the following chapter we will point to a technique that can reduce the effort of learning a framework, i.e. framework documentation. We will see that framework documentation is necessary for several reasons.

CHAPTER 3

Framework Documentation

3.1 Introduction

The previous chapter gave an overview of some general characteristics of frameworks. In that overview we did not add a discussion on documenting frameworks, because we would like to give it some special attention in the following sections, considering the importance of it during the remainder of this dissertation.

First of all, we point out why framework documentation is so important during the entire lifetime of the framework and to whom this documentation is addressed. Next an overview of some existing documentation techniques is given. These techniques will become important when we introduce our proposed framework.

3.2 Why is Framework Documentation Important?

The design of standard software products needs to be documented and this is not different for frameworks; in addition we acknowledge that the documentation of frameworks is of key importance in its use. Without good documentation the framework will not be used correctly or will even not be used at all. We will continue with explaining why framework documentation is so important.

Understanding a framework makes reuse possible: Documentation of applications is necessary because it allows to reuse and maintain the applications. Before a developer can reuse or maintain an application, he needs to understand the design of it, i.e. he needs to know why the origi-

nal developer chose to implement or design the application like he did. For instance, aspects like the interactions between instances of the framework classes certainly need to be documented since this information is important in understanding how it works. Without this information, the framework user needs to dig into the code, which takes a lot of time and which is even not possible when the uncompiled code is not available to the reuser. We can conclude that a framework needs to be documented as a whole: documenting the framework classes separately is not sufficient.

Make framework-based application development more attractive over developing from scratch: There are two major parties (application developer and framework developer) involved in using frameworks: they both have their reasons why using frameworks should be documented well. We will first discuss the motivation of an application developer.

Frameworks are introduced because the framework already contains the general design of a type of application that an application developer wants to build. Consequently, he gains time that he would have spent in developing from scratch. However, a disadvantage of frameworks is that application developers need some time to learn them. Good documentation can make the gain of time higher than the loss of time, consequently the application developers are more attracted to framework-based application development instead of starting from scratch.

Framework developers, the other party, want good framework documentation since the development of a framework is very expensive. This huge cost is justified by the fact that the framework will be reused to build other applications. If a lot of applications can be build using the framework, then money is saved that otherwise would be spend in building the applications from scratch. Thus, a framework developer wins by promoting his framework. Framework documentation can help a lot with this promoting activity.

Frameworks are not fully functional applications: Unlike standard applications, a framework is not a fully functional application. It needs to be customised by an application engineer. The framework developer designed the framework in such a way that the application engineer only has to fill in the functionality that is typical for the application under development. The framework developer has an idea of how the framework must be reused and he should document his expectations to the reuser to make it possible for him to reuse the framework. Expectations such as which methods he needs to write, what the methods need to do and where these methods must be plugged in.

Abstract Design: The nature of a framework makes its understanding not very easy. The definitions in section 2.1 illustrate that a framework developer writes a framework for a particular application domain, so he can

capture all his knowledge of the application domain in the framework. The more a framework is reused, the more feedback the framework developer gained from reusers, and this allows him to make the design more abstract, i.e. more reusable. Consequently, a framework is a very abstract design in which most of the real world concepts are lost. Framework documentation is necessary here to explain the abstract design, without this documentation a reuser is forced to learn the whole application domain and that is just what he wanted to avoid using the framework.

Frameworks evolve: Documentation is not only necessary to understand a framework and to learn how to use a framework, but it is also important that reusers of a framework are kept up to date with the evolution of the framework. When application developers took the time to learn the framework, which was time demanding in the beginning, then they want to continue using it for a certain time. Therefore frameworks need to have documentation on how they evolved from one version to another, so that the reuser can update his application to the latest version.

3.3 Using Object-Oriented Frameworks

In the following section we will discuss about the different ways to use a framework and the different types of framework users. This will make it easier to argue about their requirements for framework documentation.

3.3.1 Ways to Use a Framework

Not all framework users use the framework in the same way. Each of them requires a different amount of knowledge about the framework and a different level of skill in using it. At this point framework documentation can become handy to provide the necessary information. Taligent [Inc] defines three main ways in which frameworks can be used.

- **As is:** A framework user will use the framework as it is delivered, without modifying or adding something. The framework is treated as a black-box, i.e. the user only needs to put together some components to build an application. In other words, the framework developer delivered together with the framework a library of components that can be used to built an application.
- **Complete:** The framework user adds components to the framework by filling in parts left open by the framework developers (hot-spots). Completing the framework is necessary if it does not come with a full set of library components.

- **Customise:** The framework user replaces part of the framework with custom code. Modifying the framework in such a way requires a detailed knowledge of how the framework operates.

3.3.2 Framework Users

Besides the different ways a framework can be used, it is also possible that people with different goals will use the framework. In [FHLS98], four different types of users are distinguished.

- **Users who have to decide what framework to use:** The user who selects the framework, that will be used to create new applications, needs to have a clear view of the requirements that need to be fulfilled by the future applications. Once he has all the requirements, he has to browse framework libraries to find the framework that is the best choice considering his requirements. If no good framework is found, then he can propose to build a framework from scratch.
  To select the appropriate framework the user needs to know the context, the intended use, the features and also the limitations of that framework.

- **Regular user:** Many users will use a framework in the way that it was meant to be used. There are two ways to do this: the first possibility is that they can built an application by putting components, that were delivered together with the framework, together. The second possibility is that they create their own components to work with the framework. These components follow strict the intentions of the framework developer. They customise the hot-spots like the framework developer has intended it.
  Consequently, a regular user needs to know only enough about the framework to enable him to effectively use it, and typically do not require detailed knowledge of the framework. He only needs to know how he can customise the framework and not why he has to do it that way.

- **Advanced user:** Some users will want to use the framework in unexpected ways; ways that the framework developers never anticipated or planned for. They will use the framework in the same way as regular users, i.e. they will also implement a component to customise a hot-spot in the framework. However, the advanced user will customise the hot-spot in a way that was not expected by the framework developer. For instance, an extra function call is added to extend the functionality of the framework. While doing this the obligations and constraints of the design are respected.

For this kind of customising behaviour, the advanced user needs an in-depth knowledge of the framework. He needs to know where he has to change the implementation of the framework to alter its behaviour.

- **Framework developer:** A framework can evolve by adding functionality or fixing errors; specialised frameworks can be derived by adding specialised classes, or the framework can be generalised to accommodate a wider domain.
  The framework developers, who perform these activities, need to know all the details of the design and implementation of the framework and must keep in mind how changes will affect applications that were already developed using the framework.

These different types of users each have their own special activities for which they need a different amount and a different kind of knowledge. When documenting a framework one should take into account the differences between these types of framework users.

3.4 What Needs to be Documented?

In the previous section we discussed who needs framework documentation and what their specific task is. In this section we will elaborate on what needs to be included in the documentation. Later we will describe different levels in the documentation, with different contents according to different types of users.

According to [Mat96] the following information has to be included into the documentation for understanding and using classes in a class library:

- **Structural information:** a description of the class, that includes things such as its name, its superclass if any, information about attributes and operations
- **Descriptions:** a description in natural language of each class, describing the purpose of the class and the abstraction it represents.
- **Usage:** a description that tells if the class is intended to be instantiated in a particular way, or not instantiated at all.
- **Terminology:** the terminology introduced regarding the concept the class captures.
- **Configuration:** a description of how classes are related to each other, and intended to be instantiated in certain configurations.
- **Assertions:** semantic constraints stating preconditions and postconditions for operations and class invariants.

- **Operations:** for each operation include some structural documentation such as parameters, results of operations and the corresponding types.

The list of items displayed above are the aspects that need to be included in the documentation of class libraries. This is equally true for frameworks, but in addition to these aspects, the framework documentation must be described on different levels of abstraction since it must address the needs of developers with varying levels of experience. The aspects introduced above are not enough to document a framework. They describe each class of the framework in isolation, while in frameworks it is also important how these classes collaborate with each other. Therefore we need in addition to this some specialised documentation. Johnson gives in [Joh92] three types of documentation needed for frameworks:

- **The purpose of the framework:** a description of the framework domain, the requirements it is meant to fulfil, and any limitations it has.
- **How to use the framework:** A description of the way the framework builder intended the framework to be used. It captures the framework developer's knowledge of how the framework can be used.
- **The design of the framework:** A description of the structure and behaviour of the framework.

The three types of documentation show that framework documentation should include more than a detailed design description of the framework using for instance "class diagrams". While the detailed description is necessary for the advanced users of the framework, regular users will have to do too much effort to learn information that they do not need to know. They just need to know what the framework is used for and how it can be used, to accomplish their task.

3.5 Current Documentation Techniques

Here an overview of some existing techniques to document frameworks is given. For each technique a short description is supplied. An evaluation follows this overview and compares the techniques based on different criteria.

3.5.1 Cookbook

Cookbooks guide reusers step by step for building new applications starting from the framework. A cookbook contains recipes that describe in an

informal way, through natural language, how to use a framework and they usually do not explain the internal design and implementation details of the framework. The first recipe in the cookbook acts as an overview of the cookbook with cross-references to the remaining recipes. Each recipe can cross-reference to some other related recipes.

One of the first well-known frameworks, i.e. Model-View-Controller, was already documented by a "Cookbook for Model-View-Controller" [KP98].

A similar approach to cookbooks and recipes was presented in [Joh92] using an informal pattern language to document a framework in natural language. The documentation of the framework is organised in a set of patterns, sometimes called a "pattern language". Each pattern describes a problem that occurs over and over again in the problem domain of the framework, and then describes how to solve that problem. Each pattern has the same format. First they give a description of the problem and a detailed discussion of the different ways to solve the problem, with examples and pointers to other parts of the framework. The pattern ends with a summary of the solution, followed by pointers to other patterns. Considering the fact that patterns describe a problem and how to solve it, they can be seen as the recipes of a cookbook. The pattern language in [Joh92] gives also an organisation to the cookbook, the patterns are organised in such a way that the patterns for the most frequent forms of reuse are represented early, and where concepts and details are delayed as long as possible.

In [LK], to avoid confusion with design patterns, call the patterns that Johnson introduced *motifs*. In their work they use a template for a motif description that has a name and intent, a description of the reuse situation, the steps involved in customisation, and cross references to other motifs, design patterns, and contracts. The design patterns provide information about the internal architecture, and the contracts provide a description of the collaborations relevant to the motif.

A difference between a cookbook and the pattern language of Johnson is that a cookbook gives a step-wise guidance to solve a problem, while the patterns only give an informal description. Another difference is that the pattern language gives an order in which to browse the different patterns.

Evaluation

A major weakness of a cookbook approach is that it describes the normal way to use the framework, but lacks to means of knowing how the framework will be used in the future. A cookbook cannot describe every use of the framework, especially not the use that was not foreseen by the developers. Therefore the cookbook approach is not appropriate for users that want to go beyond the normal use of the framework, but it is qualified for addressing the purpose of the framework and presenting some examples.

3.5.2 Design Patterns

In general, design patterns try to record experience in designing object-oriented software in a form such that people can effectively reuse it. A pattern presents a solution (a description of objects and classes that participate in the design, and their responsibilities and collaborations) to a commonly occurring design problem and besides that it also gives a name to the design construct that solved it [GHJV95]. As a side effect, we can say that design patterns form a vocabulary that can be shared among teams which allows them to refer without confusion to a particular design construct.

Each design pattern, in general, has four essential elements:

- a name
- a problem description
- a description of the solution
- a discussion of the consequences

These four elements are described through informal textual notations, diagrams (using notations such as OMT, UML), and programming language samples.

In addition to providing a solution for a design problem, design patterns are also useful in the context of describing parts of a framework design. The description of the design pattern includes both the classes and the collaborations between the classes, which is crucial to understand the framework. Besides that, patterns describe the design on a higher level of granularity. A framework can be viewed as interwoven patterns instead of a collection of low level classes.

Another important reason to use design patterns in documentation is that they give a description of the problem it solves and the context in which it is used. This information makes it easier to understand why certain design decisions were made.

Evaluation

Considering the fact that design patterns are an informal description of the design, makes design patterns not appropriate to use as the only documentation of the design. Design patterns are mostly used to describe the purpose of a framework and to let application programmers use a framework without having to understand in detail how it works. They give design information but no usage information.

3.5.3 Exemplars

An exemplar is an executable visual model consisting of instances of concrete classes together with explicit representation of their collaborations. For each abstract class in the framework, at least one of its concrete subclasses must be instantiated in the exemplar [GM95]. The explicit representation of the collaboration among classes makes it possible to view these collaborations and, more important, to interactively explore the relationships between the instances of the classes with a visual tool. The interconnections between the main classes of the framework will be better understood when following this way of working.

Exemplars help application developers in adapting the framework to their own needs. To be able to do that, they must know where to make the necessary changes, they must know how classes and methods depend on each other. An exemplar is provided by a framework developer and application developers can use the exemplar by following several steps (a sort of recipe):

1. **Representation:** Framework developers provide an exemplar, making explicit the architecture of a framework. An exemplar consists of one or more instances of at least one concrete class for each abstract class in the framework.

2. **Exploration:** A visual tool lets the reuser interactively browse the exemplar in order to understand the responsibilities and relationships of the objects. The visual tool not only shows the structural relationships of the objects, but it also shows the collaborative relationships via observing message passing among these objects. The reuser can actually see how the framework executes in a limited way, which facilitates the understanding of the framework.

3. **Selection:** After the framework user has gained an understanding of the framework through exploring the exemplar, the user can select objects from the exemplar that need to be replaced or modified to fit the requirements of the application being built.

4. **Finding alternatives:** The tool allows the user to explore the inheritance hierarchy for a selected object. It displays the abstract class and the framework library classes derived from the class that corresponds to the object and the user can select the one that is needed for the application. By restricting the search to the inheritance hierarchy of the selected object, the tool cuts down on the number of classes the user has to search through.

5. **Adaptation:** If no appropriate replacement class exists within the framework, then a new one has to be created. The replacement can,

for example be a subclass of the existing abstract class corresponding to the object selected, or it can be a composition of more than one existing class.

An object of the replacement class can then be inserted into the exemplar model, and can be executed to see if it has the desired behaviour. Framework users repeat this process for every object they wish to replace and can, in part, prototype the application by selectively replacing objects.

The exploration tool gives several views on the framework using different types of diagrams. Each step has its own diagram to show the necessary information. In the first step the tool shows an "Object Model View" to show the static relationships among the objects. This Object Model View is very similar to a UML object diagram. Also in the first step the user can browse the responsibilities of each object. The tool uses for this an "Object Property Viewer". Again we find some similarities with another documentation technique, i.e. CRC cards. CRC cards give the name of a class together with the responsibilities it has in the design. In the second step the reuser can have a look at messaging interactions. A diagram, called "Event Scenario Diagram", shows these interactions. This diagram is comparable to an "interaction diagram" in UML. In the fourth step one has to find alternatives for an object in the Object Model View. Exemplars propose to browse the hierarchy of that object, to be able to do this one first has to lookup the class of the object in the property view and then locate the class in a class diagram to see the hierarchy. This class diagram is very alike to the "class diagrams" that exist in UML.

Evaluation

The exemplar technique is a good way to show how the framework works and it aids the user with examples on how the framework should be used, should be instantiated.

3.5.4 Interaction Contracts

A contract is a specification of obligations and collaborations [Hol92]. Interaction contracts describe object interactions, i.e., a group of objects that interact via message passing to accomplish some system task. Interaction contracts aim at providing an explicit formal textual representation of object interactions. Each contract consists of the following parts:

- **Contractual obligations:** The obligations define what each participant must support. The obligations include both type obligations (variables and interfaces) and causal obligations. The causal obli-

gations consist of sequences of actions that must be performed and conditions that must be met.

- **Invariants:** The contract also specifies any conditions that must always be kept true by the contract, and how to satisfy the invariant when it becomes false.
- **Instantiation:** Preconditions from the final part of the contract which must be satisfied by the participants before the contract can be established.

Evaluation

These contracts show the users of the framework how the objects of the framework collaborate to achieve the goal of the framework. A contract writes down how the objects interact and which methods are important to achieve that behaviour. Thus, interaction contracts are appropriate to give detailed design descriptions.

3.5.5 Reuse Contracts

Reuse contracts have been developed to provide a way to document software, software reuse and software evolution and to provide support for change propagation [Luc97]. Documenting evolution is an important issue in reusing components because reusers need some support when they decide to update to the new version of the component. Upgrading to a new version of the component is not without risk: the behaviour of the evolved component can be changed or assumptions that could be made before do not hold anymore. It is important that the reuser can detect conflicts and therefore the assumptions, made during component development, should be documented. Reuse contracts are exactly developed to aid in that task.

The reuse contract model has two main concepts: *collaboration* and *reuse contracts*. A collaboration contract describes the collaboration between participants, i.e. software entities that have an interface and that invoke each other's operations. A collaboration contract consists of two parts: the static structure and the interaction structure. The static structure shows the participants of the contract and how they are related or acquainted to each other. The interaction structure shows the messages sent between the participants. The interaction structure also shows which method invoked the message-send, this makes it possible to see which methods rely on which other methods. Doing so, collaboration contracts make the implicit assumptions of the developers explicit. A more formal definition of collaboration contracts is:

> *A **collaboration contract** consists of a name and a set of participants. Each **participant** has a unique name within the collaboration contract, an **interface** holding methods and an **acquaintance clause** holding acquaintance relationships. A **method** has a **method signature**, an annotation abstract or concrete, and a **specialisation clause**. A specialisation clause is a set of **method invocations**, associating an acquaintance name with a method signature. An **acquaintance relationship** is an association between an acquaintance name and a participant name. [DH98]*

The second aspect of the reuse contract model is the *reuse contract*. Reuse contracts are used to document how a part of the system is reused. In general, a *reuse contract* is a contract between a provider and a reuser. It comprises a *provider clause*, that states what is actually provided, a *reuser clause*, that states what is actually reused, and a *contract type*, that states how the contents of the provider clause is reused.

The provider clause contains a collaboration contract that states what is provided and the reuser clause describes the changes that were made to the provider clause. The contract type associated describes how the provider clause is reused. Several contract types exist, each defining a different kind of reuse. A contract type expresses how the provided component is modified. Possible contract types include extension, cancellation, refinement and coarsening. The contract type imposes obligations, permissions and prohibitions on the reuser. For example, the extension contract type requires reusers to add new elements to the provider, but prohibits overriding of existing elements. These contract types are the basis for detecting conflicts when components evolve.

Collaboration contracts give an overview of how the methods of the classes in the framework relate to each other. But more important, reuse contracts show how a system should be reused or how a system has evolved. They can aid the framework developer in estimating how much effort is needed to perform a planned change. Besides that they also give a good insight into the effect of the changes. A proposed change can be modelled in a reuser clause and the framework developer can then apply it on the already existing design, modelled in the provider clause. It can see if some conflict occur through conflict detection.

Here we present a collaboration contract for a part of the mini "shopping framework". In figure 3.1 a collaboration contract shows how the methods in the framework depend on each other. The static structure presents the relations among the participants. We can see in the dynamic structure that the method "payingGoods" of the participant "ShopKeeper" calls several methods of the other participants. It makes a call to the method payClient

of the Client participant or it sends the message "readPrice" to itself.

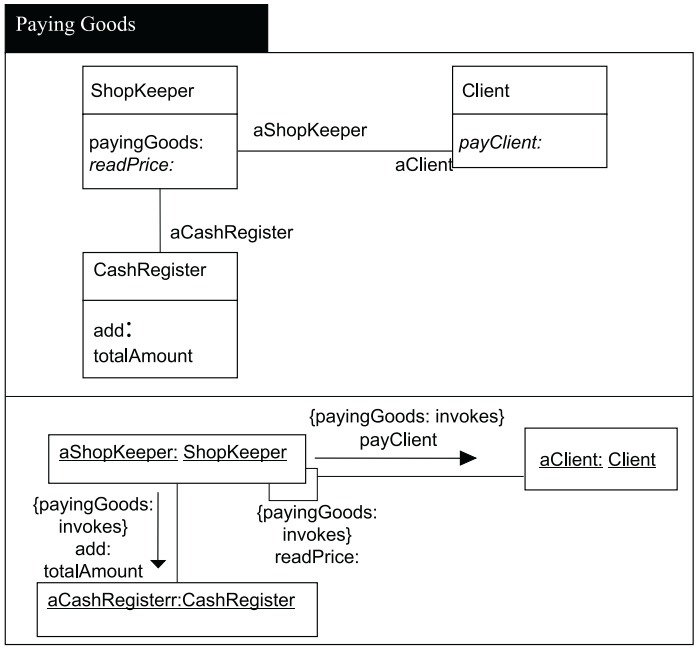

Figure 3.1: Paying Goods Collaboration Contract

The collaboration contract in figure 3.1 showed the relationships between the participants. In figure 3.2, we show how to adapt the framework to allow a client to pay by using a credit card. Therefore the method "receiveCreditCard" must be added to the participant "ShopKeeper". This method will do whatever is necessary to accept a credit card. This extension of the participant is depicted in the first reuser clause with contract type "participant extension". The other reuser clause of type "Participant Refinement" documents that a method call is added.

Evaluation

Reuse contracts are appropriate to be used in the design description of a framework. They explain how methods are related to each other. Regular and advanced users can use reuse contracts when they need some design info. But reuse contracts also can help framework developers and maintainers when they want to document the changes they made to the framework.

3.5.6 Examples

Examples provide another means of learning the framework and complement all of the other types of documentation. The examples can be complete

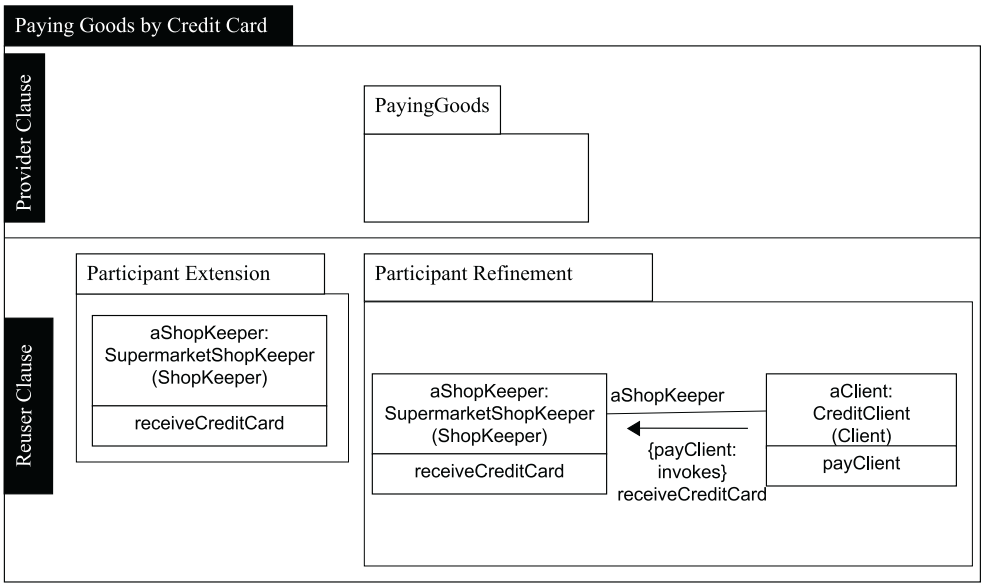

Figure 3.2: Paying by credit card Reuse Contract

applications developed from the framework, or smaller examples to demonstrate how a particular hot-spot can be used, or how a given design pattern works. The examples are valuable because they make an abstract framework concrete and easier to understand. They provide a specific problem and show how the framework can be used to solve that problem. Examples are also valuable to framework evolvers to see if the provided examples of framework instantiation are still valid for the changed framework. However, examples cannot cover every possible use of the framework and so the other forms of purpose, intent and design documentation are still necessary (Johnson, OOPLSA 92 Conference proceedings, 1992[Inc]).

3.5.7 Evaluation

In the previous sections we gave a description of some existing documentation techniques. In this section an evaluation is made of these techniques. They are compared against each other based on different criteria. First, we summarise the different users and their need for a different level of detail in the description, after that a table shows the main characteristics of the techniques and at what level of detail they describe the framework. We conclude with a table that gives for each type of user one or more techniques that are the most appropriate to support him in his tasks.

Different types of users need a different level of detail:

Like we mentioned before, a framework has a long life cycle and there are several types of users (section 3.3.2) that come into contact with the framework. Each of these users has a different task to accomplish. Different tasks can possibly require different skills of the users and can call for a different level of detail in the documentation (section 3.4). Not every user needs to have the same detailed description of the framework.

For example, a user, who has to select a framework, does not need to know more about the framework than its purpose.
This is in contradiction with the advanced framework user. He also needs to know the purpose of the framework to provide him with some background information of the framework. However, in addition to this information, he needs to know the regular use cases of the framework, so he can see how the framework can be used without adaptations. And finally, the advanced user needs to have a detailed description of the framework design, so he can detect how he can alter the behaviour of the framework.

	Purpose	Use	Design
Has to choose	Yes	No	No
Regular	Yes	Yes	No
Advanced	Yes	Yes	Yes
Developer	No	No	Yes

Table 3.1: Different audiences require different level of detail

In table 3.1, we give an overview of the user types and their required level of detail in the framework description.

Characteristics of documentation techniques:

In table 3.2, an overview is given of the main characteristics of the documentation techniques. The third column of the table shows at what level of detail the technique describes the framework.

For example, cookbooks give a stepwise guidance in using and learning the framework. This technique describes the purpose and the use of the framework, but it does not give sufficient information about the design of it.

Different types of users, different techniques used:

Since the different types of users reuse the framework in a different way (see section 3.3.2), they need to need to know other information about the framework. Documentation techniques can provide this information to the

	Characteristics	Level of detail
Cookbook	Stepwise guidance in solving problems	purpose use
Design Patterns	Names design constructs Higher level of design description	purpose design
Exemplars	Viewing framework behaviour by completing the framework template	use design
Interaction Contracts	Formal descriptions of interactions	design
Reuse Contracts	Document collaborations Document evolution Support in change propagation	use design
Examples	Describe possible run-time behaviour	purpose use design

Table 3.2: Main characteristics of each technique

framework users. However, there is no documentation technique that can fulfil all the requirements of the user. We will give an overview the users and explain which techniques they need and how the techniques are used.

- **User who chooses the framework:** Since this user needs to select an appropriate framework, he needs to have an overview of all the frameworks he can select from. A textual description of the framework together with some examples can cover the requirements of this user.
- **Regular user of black-box framework:** The regular user needs to know the different hot-spots and what the available subclasses (in the component library) are. Using a cookbook with some examples will help the user in his task. Each recipe of the cookbook describes a hot-spot. With the Exemplars technique, the user can browse an example application and choose another subclass for the hot- spot. Now, the Exemplar, with the changed subclass, shows the altered behaviour of the application. The user can decide if it is the behaviour he wants or can continue browsing subclasses.
- **Regular user of white-box framework:** In contradiction with the previous user, this user needs to implement his own subclass to customise the hot-spot following the steps expected by the developer. Again a cookbook with references to examples and interaction or reuse contracts should cover the requirements of this user. Each recipe in the

cookbook describes a hot-spot. The recipe should describe each step that should be taken in the customisation process. Things like which classes should be subclassed and which methods should be overridden. In this description, links to reuse contracts and interaction contracts are valuable, since they explain the interaction between the objects. In the Exemplar technique the user can browse the interaction, which can give him a better understanding. The Exemplar also allows him to view a prototype of his solution.

- **Advanced user:** The advanced user uses the framework in unexpected ways, but still respects the constraints and obligations of the hot-spot. A cookbook with recipes describing the constraints and obligations of each hot-spot is necessary. A recipe uses design patterns, interaction contracts, reuse contracts to clarify these restrictions. Exemplars provide him with an easy way to understand the interactions among the objects.

- **Framework Developer:** The framework developer requires information about the intent and the rationale behind the choice of the hot-spots. A cookbook can provide recipes for each hot-spot describing the intent. Design patterns, interaction contracts and reuse contracts will describe in a more formal way. Reuse contracts will also document how the framework evolves and what the impact is on the existing applications. Examples are also necessary since they can help the developer in checking if no unexpected changes occurred at other places of the framework.

Table 3.3 summarises for each type of user what techniques are the most appropriate for him.

3.6 Summary

In this chapter we gave an answer to the questions why we need special documentation for frameworks. An overview of the different ways to use a framework and the different users of the framework was given. A big part of this chapter was a discussion of several existing techniques to document frameworks.

In the following chapter we will propose a new approach towards documenting of frameworks.

	Documentation Technique
User who chooses	Textual description Cookbooks Examples
Regular user * as is	Textual description Cookbooks Motifs/Patterns Examples Exemplars
Regular user * complete	Textual description Cookbooks Examples Exemplars Motifs/Patterns Reuse Contracts
Advanced user	Textual description Cookbook Exemplars Design Patterns Motifs/Patterns Interaction contracts Reuse Contracts
Developer	Cookbook Examples Design Patterns Interaction contracts Reuse Contracts

Table 3.3: Appropriate documentation techniques for each type of user

CHAPTER 4

OONavigator Framework

4.1 Description of the Architecture

In this section we introduce the *OONavigator* framework. OONavigator is a framework for extending object-oriented applications with hypermedia[1] functionality. We will use this framework to define navigational views for each type of framework user and to add some navigational features to the framework documentation. First we will explain what we mean by navigational views. Navigational views are used in a hypermedia application to define the way in which a user is exploring the hypermedia and this in order to avoid redundant information and to prevent the user from getting lost in the hyperspace. Inside a navigational view it is possible to create different navigational contexts, which are defined as sets of nodes and the links between these nodes. A navigational context groups nodes and links that satisfy the same property. Navigational contexts allow users to have a smaller space to browse, i.e. it gives the user only the information he is interested in.

Use

The OONavigator framework can be used in two different ways. One possibility is that it is used to add some navigational behaviour to an object-

[1] A hypermedia is a special version of a hypertext, which is a collection of documents (or "nodes") containing cross-references or "links" which, with the aid of an interactive browser program, allow the reader to move easily from one document to another. Hypermedia simply extends the notion of the text in hypertext by including visual information, sound, animation, and other forms of data.

oriented application and the other one is to define a hypermedia application with the semantics of an OO model (see [GR]). In our case we will use the framework to add navigational behaviour to an OO application.

Design

The OONavigator framework is designed in such a way that a separation of the navigational features from the application model is possible. All navigational features are defined on a separate layer on top of the base model of the application. In this way the framework allows extending applications with hypermedia functionality through a direct mapping of objects to nodes and links, thus viewing objects as nodes and accessing related ones by link traversal. The application remains unaware of navigation, allowing making reusable, maintainable and less complex extensions.

The framework architecture is build out of 3 layers. These layers are:

1. **Object level:** This level consists of the classes of the application domain that will provide the data to be shown, the relations among data, and the behaviour that will be extracted by the hypermedia level.

2. **Hypermedia level:** In this level the hypermedia framework components are defined. The designer will be able to instantiate these components in order to define a navigational view over the first level.

3. **Visual level:** In this level, the visual appearance of the nodes is specified. It is possible to define several graphical representations for the same node.

We will explain more in detail the *Hypermedia Level Architecture*: we will explain what nodes and links are, what can be understood under access structures and finally we will say something about navigational views, node views and representations.

- **Nodes and Links:** Like we mentioned before, the OONavigator design is developed in such a way that the base model of a hypermedia application, built using OONavigator, is unaware of the navigational features that are defined on top of it. In the hypermedia level of the framework we will define some nodes and links that will give a view of the first level, i.e. the application model. Nodes and links are defined as empty templates that only know how to be navigated, but that require to be "plugged" with an application object in order to be instantiated. Every *node* that is derived from the object model is said to "depend on" or "observe" the correspondent object to which it is plugged. Interface events, which will be defined on top of these nodes

and links, that the node does not understand as link activation, are directly delegated to the object that is plugged to the node. In this way all application behaviour is preserved. It is even possible to map a node to several objects in the object model. This would be preferred when one or more objects depend on another and they are not relevant enough to be isolated in a node.

The hypermedia level also contains a special node, i.e. a *hypernode*. This node does not need a subject in the object model level. It is used for hypermedia purposes only. For example to hold an image, movie, etc.

The other important class in the hypermedia level of the OONavigator is the *Link* class. This class represents the association between two or more nodes. Links are accessed through anchors contained in nodes or in access structures, and arrive at link- endpoints. The link-endpoints resolve the target to which should be navigated.

- **Access structures:** With the definition of *access structures* the OONavigator framework tackles the problem of information overhead that a hypermedia may produce. They present three different types of access structures: Indexes, Guided-tours and Iconic structures. Each access structure has a set of target nodes, a set of selectors and a logic predicate on the target nodes to allow the definition of conditional access structures.

- **Navigational Views, Node Views and Representations:** The OONavigator implements a way to have different navigational views in the hypermedia for distinct roles or profiles. They associate different *node views* to a node for each identified role or profile. This provides different navigation alternatives when for example different outcoming links have been defined in each navigational view.

  Besides the different views of the node it is also possible to define some *representations* inside each node view and this in order to change the appearance of the media of the same piece of information. For example should a comment be available as text or as an audio file to which the user can listen?

  So we can say that each node view defines the set of data to be displayed under a given navigational view, and each representation inside the node view will define how to display that data.

Summary

To build a hypermedia application using OONavigator, one should first define an object model of the application without hypermedia functionality.

On top of this object model some nodes and links should be defined to make the objects in the model navigable. Nodes and links can be seen as wrappers of these objects. For each of the nodes several node views can be defined, which allows to alternate the outlook of the node according to the navigational view in which it is seen.

4.2 OOHDM: A Design Methodology for Hypermedia Applications

In the previous section we discussed the framework to add navigational features to an object-oriented application without polluting the base model. Here in this section we present a methodology that aids developers in building such an application. There are two important aspects in these applications: the navigational and interface structure.

The Object-Oriented Hypermedia Design Method (OOHDM) [SR98] is a model-based approach for building hypermedia applications. This approach separates the design process into four different activities namely conceptual design, navigational design, abstract interface design and implementation. During each activity a set of object-oriented models describing particular design concerns are built or enriched from previous interactions. We will give here a short overview of the different activities.

4.2.1 Conceptual Design

In this step a conceptual model of the application domain is built using well-known object-oriented modelling principles. Conceptual classes may be built using aggregation and generalization/specialization hierarchies. There is no concern for the types of users and tasks, only for the application domain semantics. The result of this activity is a conceptual schema built out of sub-systems, classes and relationships.

4.2.2 Navigational Design

Navigation design is expressed in two schemas:

- the navigational class schema
- the navigational context schema

A navigational model is built as a view over a conceptual model, thus allowing the construction of different models according to different users profiles. Each navigational model provides a subjective view of the conceptual model.

We describe the navigational structure of a hypermedia application by defining *navigational classes*, which reflect the chosen view over the application domain. A *navigational class schema* describes all navigable objects using navigational classes. There are predefined types of navigational classes: nodes, links and access structures. *Nodes* are characterised by a set of attributes holding perceivable information and anchors for links outgoing from that node. *Links* reflect relationships intended to be explored by the final user and are also defined as views on relationships in the conceptual model.

The navigational structure is defined in terms of *navigational contexts*, which are induced from navigation classes such as Nodes and Links. A navigational context is a set of nodes, links, *context classes* and other navigational contexts. It may be defined intentionally or extensionally, by either defining a property that all nodes and links in the context posses, or by enumerating its members.

A context can be defined in six basic ways:

1. **Class derived contexts:** instances of a node class fulfilling a certain condition.
2. **Link derived contexts:** as the set of all targets of a Link when a link is 1-to-n.
3. **Composite derived:** a context made out of all parts of a composite node.
4. **Arbitrary:** those navigational contexts built opportunistically by picking different nodes (perhaps from different classes). Arbitrary Contexts are quite useful when we want to provide the reader with a guided tour on some information items.

A *Navigation context schema* presents the relations between navigational contexts that were defined for an particular application.

Context classes complement the definition of a navigational class (a node) indicating which information is shown and which anchors are available when accessing the object in a particular context. This mechanism achieves a layering effect whereby the information in a node can be further customised depending on the context in which the node is being looked at. Navigation within a context is achieved by extending the navigational class with a context class.

4.2.3 Abstract Interface Design

The abstract interface model is built by defining perceptible objects in terms of interface classes. Interface classes are defined as aggregations of primitives classes (such as text fields and buttons) and recursively of interface

classes. Interface objects map to navigational objects, providing a perceptible appearance. Interface behaviour is declared by specifying how to handle external and user-generated events and how communication takes place between interface and navigational objects.

4.2.4 Implementation

To obtain a running implementation, the designer has to map the navigational and abstract interface models into concrete objects available in the chosen implementation environment.

4.3 Gained Functionality with OONavigator

In this section we give an overview of the functionality that we receive by using OONavigator.

- Navigating among related items of interest
- Defining different navigational view for different users profile, providing each user with the information that suits their needs
- Enhancing the graphical user interface with the possibility to define anchors for links over any type of data.
- Enabling rapid and easy access to the desired information by way of indexes and backtracking
- Guiding in the search for information that fits the user profile with guided tours
- Adding annotations and bookmarks
- Providing history lists

CHAPTER 5

The Documentation Framework

5.1 Preliminaries

The previous chapter taught us that several different types of users use a framework. All these users have their requirements of the documentation. These requirements are related with the tasks that these users have to perform. To be able to perform a task, some information about the framework is needed. The kind of information can change according to the type of the task. For example, a framework selector needs to compare different frameworks to select the most appropriate framework that fits the requirements of the future applications. To be able to decide this, he needs to know at least the purpose of all the frameworks. Another example is the regular user. His task is to instantiate the framework in ways that were foreseen by the developer. Therefore he needs to understand the framework. This understanding starts with understanding the purpose of the framework and continues with a detailed description of all the customisable points in the framework.

For each type of information that is requested, some documentation techniques were already proposed. All of them have their advantages and disadvantages. Cookbooks, for example, provide too informal usage information, but this can be completed with some practical information delivered by techniques like Exemplars. Thus, we propose to combine forces and provide a framework documentation that consists out of different documentation techniques. In addition we also want to define different views on the documentation according to the user type. Consequently, every user will get only the information he needs to accomplish his task.

Researching the domain of framework documentation leads us to the following requirements for the representation of the framework documentation:

1. Provide links between related documentation items.
2. Provide different framework user views of the documentation.
3. Put available documentation at one place.
4. Reuse documentation
5. Provide link with target framework (the framework for which we are providing the documentation)

Having the requirements in mind, it seems that a hypermedia application would be ideal to read the framework documentation. Hypermedia applications provide easy access to information and its related information.

We chose to develop a hypermedia application using the OONavigator framework (see chapter 4). This framework allows separating hypermedia features from the items to be browsed. Thus, our documentation will be unaware of the fact that it is being navigated.

Using OONavigator we also can satisfy the requirement to have several views of the framework documentation. OONavigator provides the functionality to define different views of the items being navigated. In these views it is also possible to define some guided tours; these guided tours can steer the user in his search for information.

We propose to store all framework documentation in a generic way using a repository. For each technique a model is defined to store the documentation. Some techniques are composed of more basic techniques. These basic techniques can also be reused as a component of another technique. Thus, we enable reuse of documentation.

5.2 Framework Architecture

In this section we explain in more detail the architecture of the DocFramework. We start with a general overview of the architecture and introduce the three layers of the framework. In the following sections, these layers will be explained in more detail.

5.2.1 Overview of the 3-layered Framework

In the preliminaries we identified some requirements to present framework documentation. The solution that we proposed, was to create a hypermedia application on top of a documentation repository. In this section we

present a framework that shows the abstract design of such a hypermedia application.

Navigational Layer One way to create this hypermedia application is by using the OONavigator like we described in chapter 4. This framework provides a clean separation between the navigational features, that we want to define on top of the documentation and the documentation itself. Doing this way our models of the documentation techniques will be unaware of the fact that they are navigated. Until now we can say that we have two layers: the navigational layer and the documentation repository layer.

Documentation Repository Layer The documentation repository layer can also be divided into two layers. The reason to do this is because we want to reuse documentation. First, similarities of techniques will be extracted and put in a separate layer. Another way to reuse documentation is to store the components of composed techniques into a separate layer. This makes it possible for other techniques to also use this basic information. Thus, we have two layers in the Documentation Repository Layer. We will go in further detail in the following sections.

Three layers Thus, in fact the proposed framework (DocFramework) consists out of 3 layers (shown in figure 5.1). The top layer provides the navigational behaviour and the two underlying layers represent the object models of the documentation techniques (spread over the two layers like discussed above). We call the first layer the *Basic Techniques Layer*, the second one is called the *Composed Techniques Layer* and the last layer is called the *Navigational Layer*.

Instantiating the DocFramework This framework has to be instantiated to become a hypermedia application for browsing in the framework documentation. According to the layers, the DocFramework instantiator has to decide first what documentation techniques will be used to document his framework (TargetFramework). The object models of these techniques will be located on the second layer, with possible references to models of more basic techniques on the first layer. After that he has to specify the desired navigation between the different techniques and in which views these techniques will be accessible.

Default Instantiation of DocFramework In the following sections we discuss in more detail the design of each layer and we will also show how we designed our default instantiation of the DocFramework. This default instantiation of the DocFramework should be implemented and delivered

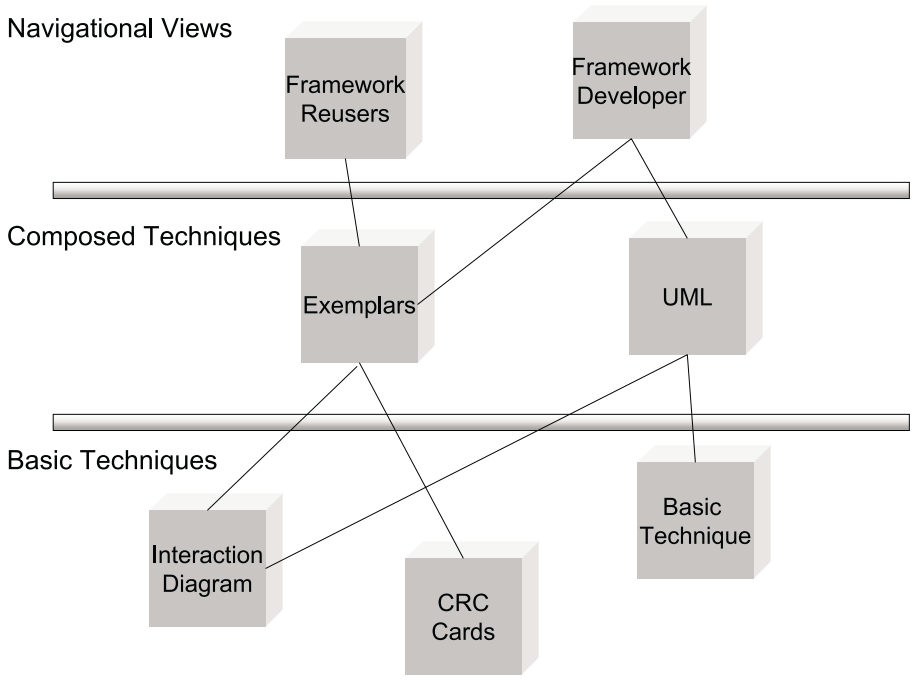

Figure 5.1: Three layered architecture of DocFramework

together with the implementation of the DocFramework. This allows users (framework documenters) of the DocFramework to save time. Normally, the framework documenter would have to go through the whole instantiation process of choosing documentation techniques and defining navigational behaviour for it. By providing the framework documenter with a default implementation allows him to act himself as user of the hypermedia application. He will have a navigational view that allows him to create documentation using the techniques of the second layer. Only framework documenters that require more functionality (e.g. another navigational behaviour, more documentation techniques, etc.) have to make changes to the default instantiation. A cookbook (or some other framework documentation) should describe how he can adapt the default instance or define a new instance from scratch.

5.2.2 Second Layer: Composed Techniques

Since in our layered structure (see figure 5.1) the *Basic Techniques Layer* (first layer) and the *Navigational Layer* (third layer) depend on the *Composed Techniques Layer* (second layer), we start our description of the different layers with the last mentioned.

Problem

For the default implementation of the framework we had to decide which techniques will be used to document the TargetFramework. We chose for the following documentation techniques: Exemplars (see section 3.5.3, [GM95]), CRC-cards [WBWW90], UML [Rat], Reuse Contracts [Luc97] , etc. While comparing these documentation techniques we discovered two things.
First, the Exemplars technique is a technique composed out of several basic techniques: Object Model Diagram, Object Property Diagram, Event Scenario Diagram and Class Diagram.
Second, we found that the UML Interaction Diagram represents the same information in about the same way as the Event Scenario Diagram of the Exemplars technique. They both document the interaction pattern of some objects and show it on a diagram with two dimensions: the vertical dimension represents time, the horizontal dimension represents different objects. Normally time proceeds down the page.
A good solution, to improve reuse of documentation, would be to represent the Exemplar as a composed technique and to make its basic techniques available for future reference without being coupled to the Exemplar. It would also be good to extract the similar information out of the Event Scenario Diagram (Exemplars) and the Interaction Diagram (UML). Doing so, the information shown in the diagrams only needs to be stored once in the repository. Information specific for a particular documentation technique should be stored outside this object. In the following paragraph we give an abstract description of a solution to this reuse problem.

Abstract Description

Like we mentioned in the introduction we will have a repository of all the documentation. We divided this repository into two layers because of the following reasons:

- **Extraction of similarities:** At the documentation repository we will have all the documentation techniques that will be available to the framework users. While comparing these techniques we discovered that a lot of documentation techniques are in some way similar to each other (e.g. Interaction Diagrams (UML) and Event Scenario Diagrams (Exemplars)). We suggest to extract the similarities of these documentation techniques and to put that representation in a new layer (Basic Techniques Layer). In the layer on top (Composed Techniques Layer), techniques will be described as wrappers of items in the underlying layer.
- **Composed techniques:** We also learned that some techniques are composed out of several other more basic techniques (e.g. Exemplars).

In the two-layer structure, the Composed Techniques Layer describes how the composed techniques are constructed out of the basic ones represented in the Basic Techniques Layer.

Design

We will present here the architecture of the documentation technique models accordingly to the separation discussed above. In the first place we need wrappers [GHJV95] on the BasicTechniques. These wrappers will add documentation technique specific information or behaviour on top of the BasicTechnique. The architecture of these wrappers is shown in figure 5.2.

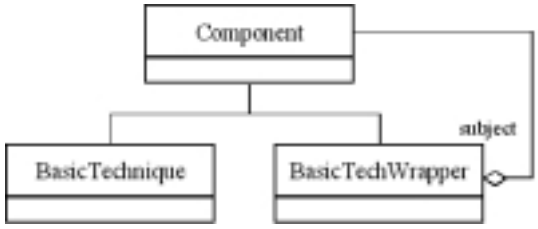

Figure 5.2: Wrappers on basic techniques

For each documentation technique approach we have to define a wrapper on top of a basic technique. We can define several wrappers for the same basic technique, because the basic technique can be reused by several different documentation techniques. The information that is different for both documentation techniques should be saved in the wrapper for that particular documentation technique. Figure 5.3 shows a possible extension of the framework with the documentation technique called Interaction Diagram for UML. Like mentioned before we can extract an Interaction Diagram basic technique out of this model. In figure 5.3 we see that InteractionDiagram is a subclass of BasicTechnique and that we defined two wrappers to enable reuse of this object in both the UML approach and the Exemplar approach.

The wrapper design we discussed before, is to create a specific representation for each basic technique, besides this we also need a representation to group these wrappers into composed techniques. Figure 5.4 shows how composed techniques are composed out of BasicTechWrappers. Each time we add a composed technique to our framework we need to subclass the class ComposedTechnique and add some BasicTechWrappers to it. This is shown in figure 5.5. The composed technique Exemplar has composition relations with four wrappers.

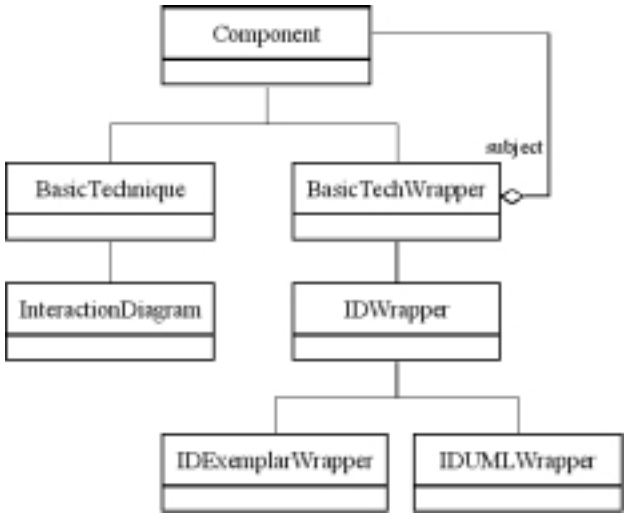

Figure 5.3: Interaction Diagram Wrapper

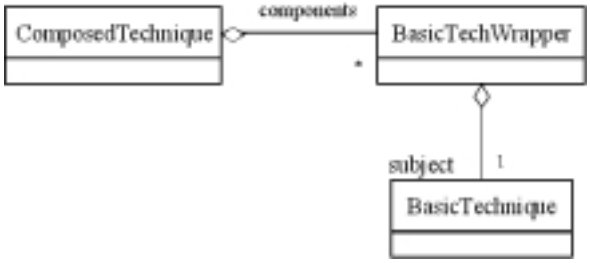

Figure 5.4: Model of composed techniques

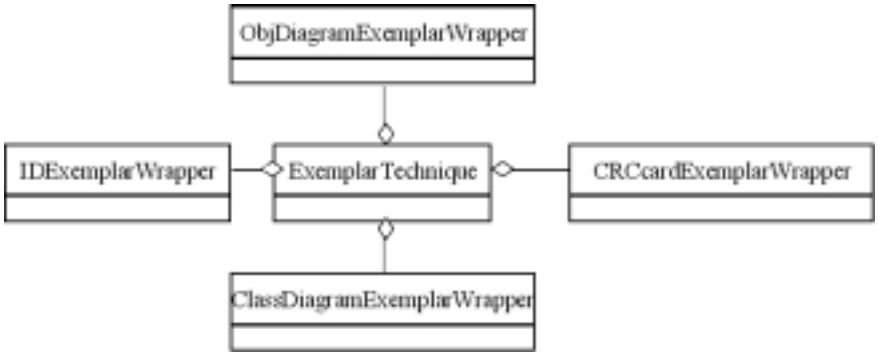

Figure 5.5: Exemplar composed of wrappers

Documentation Creating Tools

Until now we presented how documentation technique models can be added to the framework. These models of the techniques will be instantiated using

creating tools. A creating tool for Event Scenario Diagrams in the Exemplars approach creates a new instance of the class Exemplar and also of the IDExemplarWrapper. The creating tool allows incremental development of the Event Scenario Diagram. This means that step by step objects and method invocations are added to the Event Scenario Diagram. These requests are forwarded to the instance of the IDExemplarWrapper, which was created for this event scenario diagram. It is the IDExemplarWrapper who has to decide where the created documentation should be stored. This object knows which information is typical for Exemplars and will store the information in the wrapper, the other information is stored in an instance of InteractionDiagram. A possible creating tool[1] is shown in figure 5.6. Some adaptations must be made to make the tool fully operational with the DocFramework.

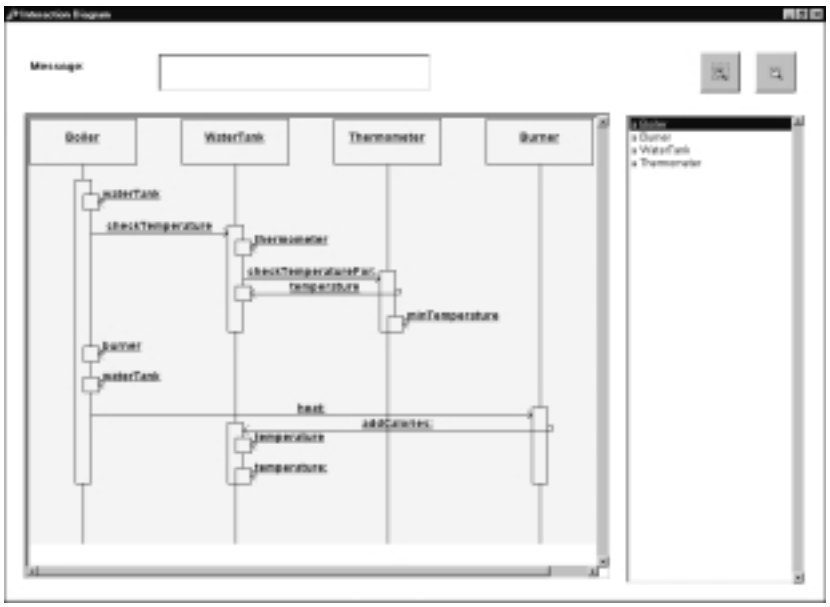

Figure 5.6: Creating tool for Interaction Diagrams

Relation with first layer

Like we mentioned in this section we have a relation between the first and the second layer of our DocFramework. We can summarise it as followed: for each technique that is composed of other techniques, we create a model of the technique on the Composed Techniques Layer. This model describes the components of the composed technique and all the components are themselves wrappers of basic techniques that are independent of a documentation

[1]This tool was created by Luis Matricardi and Marcos Godoy of the LIFIA lab

approach.

Relation with third layer

The models that we created in the Composed Techniques Layer will be made available for browsing in the Navigational Layer of the DocFramework.

5.2.3 First Layer: Basic Techniques

Meta models

The first layer of our architecture contains the models of the basic techniques. A model of the basic technique describes the different concepts that are available in the technique and how they are related to each other. For example, the model for interaction diagrams shown in figure 5.7 describes how an interaction diagram should be represented. Each interaction dia-

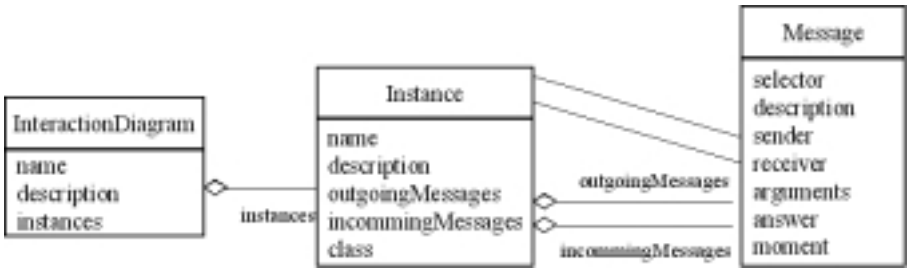

Figure 5.7: Interaction Diagram Model

gram knows its name and has a description of the special situation it is representing. An interaction diagram is also composed by a collection of instances that are involved in the interaction pattern that is presented using the interaction diagram. An instance plays a role in the interaction diagram. It knows two kinds of messages: the ones it sends to other instances (outgoingMessages) and the ones it receives from other instances (incommingMessages). A Message represents a message sent by an instance (the sender) and received by another instance (the receiver). The Message has a selector and a description. The selector is the name of the message. It also knows its arguments and the answered value, and has a reference to the moment when the message was sent. This latter is necessary to allow ordering the different messages in the interaction diagram.

Thus, we have for each basic technique a meta model that explains how the technique should be represented. Instances of the classes in the model will be made when documentation is created using a creating tool of the Composed Techniques Layer. In the previous section we discussed that the creating tool instantiates a wrapper (according to the approach used). This

wrapper will create also an instance of the subject it is wrapping. For example, the IDExemplarWrapper will create an instance of InteractionDiagram. All actions, such as "add an object" or "add a method invocation", that are valid for all possible approaches of interaction diagram will be forwarded to the InteractionDiagram of the Basic Techniques Layer. So, when a new object is added to the interaction diagram, then a message is sent to the subject of the wrapper, i.e. InteractionDiagram. This message sent results in creating a new instance of the Instance class in meta model of interaction diagram.

Reuse of Basic Techniques

A reason why we decided to extract similarities out of documentation techniques was to make it possible to reuse this information for several Composed Techniques. This results in saving space in the repository, but more important it saves the framework documenter in creating documentation.

The desired functionality is that the framework documenter is prevented from creating documentation that was already created before or that can be extracted from existing documentation. For instance, the documenter wants to create an interaction diagram describing the interaction pattern of the object A, B and C. It is possible that this interaction pattern is already described in the repository using an instance of InteractionDiagram Class. This does not absolutely imply that the documenter already made an interaction diagram including these objects. It is more likely that he used another documentation approach (for example Exemplars) that also described this interaction. Since we chose to represent the similarities of the different approaches on another layer, an instance of a basic technique class (here InteractionDiagram) can already exist.

To benefit from this property, the documenter should first give the main properties of the documentation he wants to create using a particular technique, so a search can be performed among all stored basic documentation items. As a result a small overview of several documentation items is shown in the way the basic information should be displayed according to the technique the documenter wants to use. Out of this list, he has to choose the documentation item that he intended to create or he has to create the documentation item from scratch, since none of them described the situation that was intended.

A fast way to implement this functionality is to use the possibilities of the navigational layer. In this layer we will define several views on the documentation for each user. Also a view for the framework documenter should exist. In this view he should be able to browse all available basic technique instances and it should be possible to show them according to all different documentation approaches. For example, the basic technique class Interac-

tionDiagram can be shown as part of interaction diagrams of UML or as part of the Exemplars approach.
It should also be possible to make some queries on the hypermedia application. We can use a query language (OOHQL [RDG+]) to select only those basic documentation items that fulfil some properties. The result of this query is a navigational context in which the user can browse. So, before the documenter starts with the creation of an interaction diagram of UML, he defines a query asking for all InteractionDiagram classes containing class A, B and C. He can then browse the result to see if one of the documentation items describes the same situation that the documenter meant to document.

Relation with TargetFramework

A framework documenter writes documentation for classes, methods or other items of the framework. A link between the DocFramework and the TargetFramework is desired because of several reasons:

1. **Automatic Extraction of Documentation:** If the DocFramework has a link to the TargetFramework we can make some extraction tools that can extract interesting documentation. This is of great importance for a documenter, since he wants to document very fast. Another advantage is also that the documentation will be correct according to the code.

2. **Understanding purposes:** In some documentation techniques, a link to the source code is necessary to increase the understanding of the framework.

3. **Keep documentation up-to-date:** Changes in the TargetFramework are propagated to the documentation. Because of this link we have the possibility to notify the documentation that it is not correct anymore. Or we can update the documentation automatically according to the changes.

The first two relationships are obvious and not hard to accomplish. The third relationship is less obvious. We would like to have an "update-relationship" but we do not want to hard couple the DocFramework and the TargetFramework. The documentation of a certain item in the framework should observe that item and should take some action when the item changes. This is a typical observer relationship. In the implementation we should find a way to implement the subject observer relationship without having to inform the TargetFramework classes about the fact that they are being observed. They cannot send an update message to the observers when they change. Figure 5.8 depicts this observer relationship between the TargetFramework and the DocFramework.

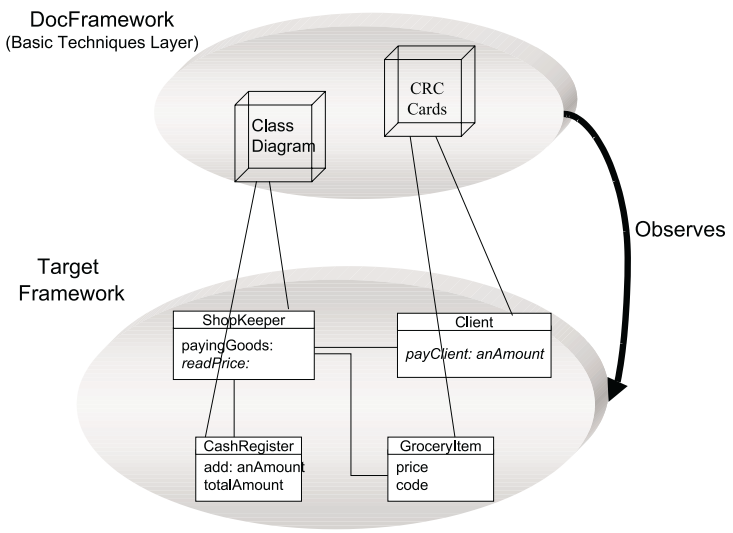

Figure 5.8: Relation with the target framework

We suggest that whenever documentation is created for a certain part of the framework, this should be registered in an object called DocumentationObserverManager. This object contains a table with two entries (see figure 5.8). The first entry is the item of the source code being documented and the second entry is a reference to the instance of the basic technique class that documents that item. The only problem that we see is that the TargetFramework should send an update to inform the DocFramework about the change. Since the TargetFramework can not know about the DocFramework we suggest to solve this problem on a meta level. We will make the development environment aware of the link between DocFramework and TargetFramework. The development environment should send a signal to the DocumenationObserverManager every time an item of the TargetFramework is edited. The update strategy is find first the item of the TargetFramework in the first column of the table. And second, inform the associated basic technique instance about the change. It is the basic technique class that takes care of the appropriate action. Possible actions are: show the documentation in another colour to make clear that something changed, or extract the new information out of the source code.

We saw in the refactoring browser[2] this kind of functionality. It is possible to open several Smalltalk browsers on the same method. If the developer changes a method in one browser and switches then to another browser, then the changed method is displayed in another colour.

[2]Refactoring Browser was created by John Brant and Don Roberts. The Refactoring Browser is an advanced browser for VisualWorks, VisualWorks/ENVY, and IBM Smalltalk. It includes all the features of the standard browsers plus several enhancements.

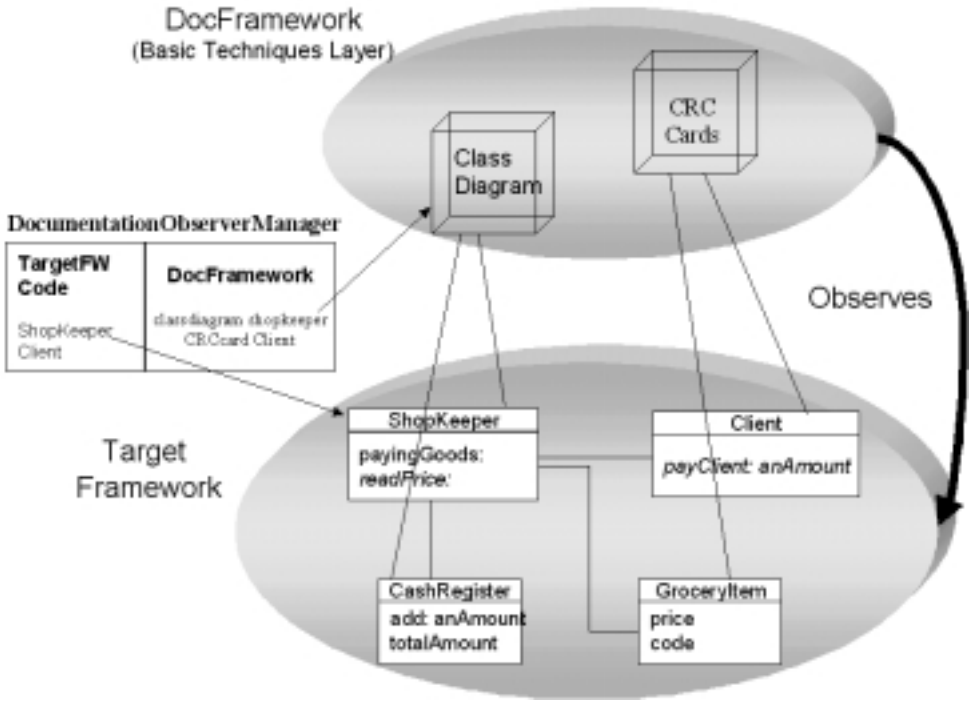

Figure 5.9: Relation with the target framework and DocumentationObserverManager

5.2.4 Third layer: Navigational Views

In this layer we will give an example of the navigational views that must be defined on top of the documentation repository. To do so we will use the OONavigator framework presented in chapter 4. The different navigational views will be defined using the methodology OOHDM.

OONavigator makes it possible to add navigational features to an object-oriented base model. In our solution, this base model is the same as the Composed Techniques Layer of our DocFramework. On top of these classes we have to define navigation.

We will define in this section the navigational view of *the regular user of a white-box framework* (discussed in section 3.5.7). We will define a navigational space for this user where he gets the information he needs by following links between documentation items the framework documenter provided for him.

Navigational View for white-box framework user:

A white-box framework user is a user that needs to subclass a framework class to customise a hot-spot in ways the developer expects it. In section 3.5.7 we already discussed which tools where most appropriate to assist this user in his task. We can summarise that a cookbook with recipes describing hot-spots is the basis for this documentation. From each recipe the user should be able to browse to examples, interaction contracts, reuse contracts and exemplars. We refer again to section 3.5.7 for the reason why these techniques are appropriate.

In the methodology OOHDM there are four activities to design a hypermedia application. We will only consider the first two activities: conceptual model design and navigational design. The abstract interface design and implementation are left as future work.

- **Conceptual model design:** In this phase one should design the base model of the hypermedia application without any reference to users or user profiles. The conceptual model of the hypermedia application is already designed: the Composed Techniques Layer of the DocFramework. We have defined on this layer the documentation techniques that this user wants to use: cookbooks, recipes, reuse contracts, interaction contracts and exemplars. On top of these classes we will define navigation. An example can be seen in figure 5.5

- **Navigational design:** In this phase, we define a navigational class schema on top of the conceptual model. This class schema consists of navigational classes for each conceptual class that has to be navigable.

  Navigational class schema
  An overview of all the navigational classes that need to be defined in the navigational class schema is given below. These navigational classes are defined on top of the conceptual model. For example the navigational class *Recipe* defines navigation for the Recipe wrapper of the Composed Techniques Layer of DocFramework.

  - Cookbook
  - Recipe
  - Reuse Contract
  - Interaction Contract
  - Exemplar
  - Examples

  Navigational Class for Recipe:
  In the navigational class of a recipe we define which attributes of the

recipe should be visible in the hypermedia application. We also define some anchors that allow to go to related information. Consequently this navigational class will contain the following items:

- Name of the recipe
- Description of the recipe
- Anchor to related examples
- Anchor to related reuse contracts
- Anchor to related interaction contracts
- Anchor to related exemplars

Navigational Class for Reuse Contracts:
The navigational class of reuse contracts should make the diagram of reuse contracts visible. From a reuse contract we should be able to browse to the source code of a class described in the diagram.

- Name
- Diagram
- Anchor to source code
- Anchor to related recipe

Navigational Context Schema for this user view:
In this schema we have to give an overview of all the navigational contexts that exist in this view, together with that we also need to define how these contexts are related to each other. How we can go from one context to another, if possible at all. Some of the navigational contexts that need to be defined are:

- Guided tour: This is a navigational context of type arbitrary. Here some nodes and links are put together. They do not have a special relationship. For this user view we define first the node cookbook, the navigational context "all reuse contracts", "all exemplars".
- Cookbook: This navigational context is of type composite derived. This means that all components of this cookbook form a navigational context. User can browse sequentially. Using this we can define that when clicked on cookbook, we will go to the first recipe of the cookbook and we will have anchors to go to the next recipe of the cookbook.
- Related reuse contracts navigational context: This is a context defined by all instances of a node class that fulfils a certain condition. We can go to this navigational context from a recipe. All reuse contracts displayed in this context must all have this recipe as their related recipe.

- All reuse contracts: This navigational context is accessible from the guided tour. The user can browse all reuse contracts without any restriction. In this context, he will have some links to a related recipe. This recipe gives him then a description of the hot-spot described.

Context Class for Reuse contract in context of "Related reuse contracts"
If a reuse contract is accessed via the link provided in a recipe we only allow the user to browse to other related reuse contracts and back to the recipe. Therefore we need to define a context class on top of the navigational class which will add these context links.

Context Class for Reuse contract in context of "All reuse contracts"
If we access the reuse contract via a the link in the guided tour to see all reuse contracts, then we will allow to user to go sequentially through the whole list. It is also possible to change from context and to go to the related recipe. These links are also defined in another context class. This means that we have one navigational class for the reuse contract, but we have several context classes on top of that navigational class to add some extra context specific behaviour.

Remarks

In this section we gave a textual description of the models that should be created using OOHDM. We did not discuss the complete design of the navigational layer either, because of the space limitations.

The navigational space of the *Regular user of a white-box framework* exists of nodes representing the available documentation techniques. We provide a guided tour for the user to guide him through the documentation. The cookbook with recipes gives links to other related documentation. When the user is browsing documentation starting from a recipe, he is limited to see only those documentation items that are related to the selected recipe. Besides a guided tour, we also offer the user to see all documentation items made by a particular documentation technique. From that view; he can browse to related items made by the same technique or he can get some more information by browsing to the recipe that is related to that documentation item. Doing so, the user is kept from getting to much information at once.

CHAPTER 6

Characteristics and Evaluation of DocFramework

In this chapter we will discuss the characteristics of the DocFramework. Who are the users of the framework? How can we extend the framework? How can we instantiate it? After we answered all these questions, an evaluation of the framework is given. We argue why somebody should use our DocFramework for documenting his framework. What are the benefits for both the framework documenter and the user who reads the provided documentation?

6.1 Users of the DocFramework

In fact there are two groups of people that can benefit form using the DocFramework. The first group of people, the framework documenters, are those persons that need to instantiate the DocFramework to become the hypermedia application that will provide the documentation of their framework to the other group of people, the future users of their framework.
The future users of the documented framework do not have immediate contact with the DocFramework. They only use an instance of the DocFramework. These users will use the hypermedia application to browse through the provided documentation.

6.2 Instantiating the DocFramework

In the previous chapter 5, we discussed the architecture of the DocFramework. It demonstrated the abstract design of the different layers of the framework. This framework still needs to be instantiated to be able to use it. Below we will describe the different steps that should be taken to go from the abstract DocFramework to a usable application that lets people read framework documentation.

1. **Decide on documentation techniques to use:** The first thing the DocFramework user needs to do is decide which documentation techniques will be used to describe his framework.

2. **Compare and analyse chosen documentation techniques:** The next thing the DocFramework user needs to do is to analyse the techniques. Are the techniques composed of more basic techniques? He also needs to compare the techniques. Maybe it is possible to extract some similarities out of a group of selected techniques.

3. **Create meta models for the basic techniques:** From the previous step the DocFramework user gained some extra knowledge about the documentation techniques. He identified some basic techniques. For these basic techniques it is possible to write down a meta model, i.e. to write down an object model that represents the structure of the basic technique. The main class of this meta model should be a subclass of the abstract class BasicTechnique (section 5.2.2). With main class of the meta model we mean the class that represents the basic technique as a whole. For example the InteractionDiagram class in the example 5.7.

4. **Create wrappers and composed techniques:** Besides the similarities of techniques, the DocFramework user has also discovered some composed techniques. First, he needs to define wrappers on basic techniques found in the previous step to be able to store differences from approach to approach. Therefore he needs to make a subclass of the BasicTechWrapper class (see figure 5.3). He also needs to define a subclass of ComposedTechnique that will store all the wrappers of the techniques that compose the composed technique (see figure 5.5).

5. **Identify different users:** The almost final step in the instantiation of the DocFramework is the definition of the different navigational views for each user type. The user of the DocFramework needs to use OOHDM to accomplish this task.

Now the DocFramework user has created a hypermedia application that he can use himself to create the documentation using creating tools of tech-

niques situated at the second level of the DocFramework.
The hypermedia application can be delivered to the users of the target framework.

We realise that instantiating the DocFramework from scratch is an exhaustive task, it is therefore that we present a default implementation of the DocFramework. The users who want to use other documentation techniques or want to define new user navigational views has to do it himself by subclassing the right classes.

6.3 Extending the DocFramework

This work showed a possible instantiation of the DocFramework. We are convinced that the framework can be and should be extended. There are two possible ways to extend the framework. First of all new navigational views for other user types should be defined, second, research should be done to find some documentation techniques that can provide special assistance to a certain user type.

6.3.1 Variations of Navigational Views

In this work we presented a navigational view for a regular framework user. We already mentioned some other users of the framework, such as advanced framework user. In fact, if these users do not require special documentation techniques, than creating a new navigational view is just a matter of using OOHDM to define another navigational view on top of the already existing repository.

6.3.2 Variations of Documentation Techniques

Research should be done to find other documentation techniques that can assist the framework user in a new way and make its task a lot easier.

6.4 Benefits

We can identify benefits from using the DocFramework for two groups of people, the ones that make the documentation and the ones that read the documentation.

6.4.1 Benefits for Framework Documenter

- **Reuse of basic documentation:** The reuse of basic documentation makes documenting a framework faster, because the framework

developer can before he writes the documentation first check if the requested documentation item already exists.

- **Up-to-date documentation:** Because of the link between the target and the DocFramework, the framework developer will get information about where the documentation became inconsistent after he changed something in the target framework.

- **Motivated to document framework:** Using the DocFramework to document the framework can work motivating for the documenter. This because he knows that the documentation he creates will be presented in a good way; in a way such that the probability that the reader will read the documentation is much higher than when the documentation is presented in several manuals.

6.4.2 Benefits for Documentation Reader

- **User specific:** A great advantage of the DocFramework is that the documentation can be organised in a way that is the most appropriate for the user type. This means that the user does not loose time with reading documentation that is of no importance considering the task he has to perform.

- **Easy access to information:** Since all documentation is organised in a hypermedia application it is very easy to find related information to the information that is currently being looked at. When viewing a cookbook that describes in natural language which methods to override to customise a hot-spot, a link to a collaboration contract can show this in a graphical way. The user gets a more complete documentation. Before the development of the DocFramework, both these documentation techniques already existed, but to use them in combination demanded some effort from the documentation reader. He needed to open several tools to view the documentation, or he had to take several manuals and put them next to each other to get the same result we got with a link traversal.

- **Up-to-date documentation:** Because of the link to the target framework it is possible to get feedback from changes that occurred in the target framework. So the documentation will stay up to date. Or some assistance is provided to keep the documentation up to date. The documentation should be updated automatically if possible, if not, the documentation should display a warning. So the user always knows if he is working with the correct documentation. He will not do any useless effort in understanding the wrong documentation.

6.5 Disadvantages

The long instantiation process of the DocFramework to create a hypermedia application can be seen as a disadvantage of using the DocFramework. The framework documenter has to decide which documentation techniques he will use, he has to create appropriate models for them (separated over the two layers). After this, he also has to create for each user type that will use his framework a navigational view. Until now, only the navigational structure among documentation items is defined. The documentation items created by the different documentation techniques still need to be created. So, now he can start with the creation of reuse contracts, cookbooks, etc. We can reduce this effort by providing a default instantiation of the DocFramework. This means that the documenter can immediately start with the creation of the documentation items. The navigation among the items is already defined.

CHAPTER 7

Future work

During this research, several ideas came across our minds. Not all of them could be given the attention that they deserve. Hereafter we will discuss some of these ideas as part of the future work.

7.1 Implementation of a Prototype

In this dissertation we presented the abstract design of the DocFramework. We described how the different layers are organised. But this is still theory, a prototype of this architecture should be implemented to validate our design. Already some effort has been done on the implementation of the base level of the DocFramework. Some meta models of the basic techniques have been implemented. We choose to implement the DocFramework in Smalltalk, since our design is based on the OONavigator framework, which is implemented in Smalltalk.

7.2 Support for Reverse Engineering

Normally, a framework developer or the framework documenter should document the framework under construction. He is the most appropriate person to document the framework since he made the design decisions. The DocFramework can assist him in this task, therefore he just needs to instantiate the DocFramework and link it with his framework. Using the creating tools linked to the DocFramework repository he can start documenting his design decisions.

Having the documentation provided together with the framework would

be the ideal, but in practice documentation is not always available or is not up-to-date. Therefore the code of a framework needs to be reverse engineered to recover its design. The recovered design artefacts should be stored for future reference. It should be possible for other developers to benefit from the effort that was performed by the reverse engineer. A lot of research is going on in this area and some tools where created to assist a reverse engineer.

Future research is necessary to make it possible to couple these reverse-engineering tools to the DocFramework. This makes it possible for the reverse engineer to store the recovered design artefacts in the repository of the DocFramework. Therefore the DocFramework needs to be extended with the model of the recovered artefacts and they must be included in some navigational views defined on the Navigational Layer. Depending on the reverse engineering tool we can extract different information. For example we can use a tool to extract interaction diagrams or reuse contracts out of source code.

7.2.1 Classification Browser

The Classification Browser [DH98] is based on a general model to organise software entities, called the software classification model. Software entities can be grouped in user-defined ways. The Classification Browser makes it possible to browse interaction structures and to classify classes, methods, acquaintance relationships and method invocations.

The Classification Browser supports the incremental reverse engineering of collaboration contracts: participants in a desired collaboration contract are collected in a classification, this is followed by the classification of methods and acquaintances in those participants. It is equally possible to recover reuse contracts. This is based on the recovery of an initial and a derived collaboration contract. Then the missing reuser clause that represents the adaptation from the initial to the derived collaboration contracts is calculated.

Further research is necessary to find out if it is possible to store the software classification model in the repository of the DocFramework. Consequently the extracted collaboration contracts will become available for browsing.

7.2.2 Other tools

Several other tools are developed to perform reverse engineering. [Die99] gives an overview of some tools. Not all of them are implemented in Smalltalk, but the idea stays the same. As long as the tools store their recovered data into a model, we can add this model to the DocFramework repository.

7.3 Compare Documentation Techniques

In the DocFramework, we made a separation between composed techniques and basic techniques. The basic techniques represent similarities that exist between documentation techniques. For example, the Event Scenario Diagram of the Exemplars technique and the Interaction Diagram of UML both describe the interaction pattern between objects. The representation of this information is extracted out of the two documentation approaches. They both refer to this extracted information if they have to represent in the interaction pattern in some diagram.

Further research should be performed to find out if more documentation techniques exist that are not similar in the first place, but are in fact. This would enable us to reuse even more documentation. For example, collaboration contracts describe also the interaction between objects, but still there are some differences. For example collaboration contracts describe participants while interaction diagrams instances of classes. In collaboration contracts the sequence of the method invocations is not of importance.

7.4 Documentation Patterns for Framework Documentation

While performing the domain research for creating the DocFramework, we found some recurring strategies in framework documentation. For example, the documentation of a white-box framework should contain, among other documentation, an explanation of the framework at code-level. Without this explanation, the user of a white-box framework will not use the framework correctly.

What we propose here as future work is to detect more of these recurring "design decisions" in the framework documentation. We want to capture these design decisions in a fixed format of a pattern. We want to create documentation patterns to assist in creating the documentation of a framework. The documentation patterns will have a format similar to the format of the design patterns described in [GHJV95]. These patterns describe the context in which the pattern should be used, a motivation, a solution and some consequences. In this way framework documenters can profit from the experience gained by others in documenting frameworks.

To achieve this goal, more research should be performed. The influence of some framework properties on the documentation should be identified. Therefore we should examine different types of frameworks; different types of framework users, frameworks of different sizes, etc.

CHAPTER 8

Conclusion

In this section we will repeat our initial goal we wanted to reach, we will explain how we reached that goal and draw our conclusions from the results.

8.1 Motivation

Currently, object oriented frameworks are very popular to describe large and abstract designs. These designs are very hard to learn by the reusers of the framework. Among several other reasons, framework documentation is of vital importance to lower the learning curve of the framework.

Already some research was performed in the area of framework documentation. Most of the techniques used to document frameworks only document parts of the framework or only document some aspects of the framework. For example, interaction diagrams document the interaction pattern of some instances, but they do not say anything about the purpose of the framework. All of this information is of importance for a complete understanding of the framework.
Another issue that was neglected in framework documentation is that several different types of users work with a framework. Each of these users have their own requirements of the documentation according to the way they use the framework and their level of expertise.

So, the goal of this dissertation was to develop a new approach for framework documentation. We choose to develop a framework (DocFramework) to assist in framework documentation. Applications built using this DocFramework are hypermedia applications that give support for different types of users and also provide links between the different documentation techniques.

These links make it possible to cover all aspects of the framework with the appropriate documentation.

8.2 Summary

To achieve this goal we had a long way to go. We will describe here how we reached that goal.

Since we decided to develop a framework to achieve our goal we had to do a thorough analysis of the domain for which we want to create the framework, i.e. the domain of frameworks and framework documentation.

First, we collected lots of information about frameworks. In chapter 2, we reported about this research. We gave an overview of the following topics: the hot-spots, the life cycle of frameworks, the different kinds of frameworks, advantages and disadvantages of using frameworks and some related techniques

Also for the domain of framework documentation we accumulated lots of information. We learned why framework documentation is so important and to whom framework documentation should be addressed. We also studied several already existing documentation techniques. These techniques were compared based on several aspects. We can conclude out of this research that different types of users need a different level of detail in the description of a framework. Besides that we acknowledged the fact that some techniques are similar to each other. For example Event Sequence diagrams of the Exemplar approach and Interaction Diagrams of UML both document the interaction pattern of a set of instances. A report of this research is reflected in chapter 3.

Having this information in mind we started to develop our DocFramework. We decided to store all documentation in a repository according to a model of the technique used to create the documentation. We made a separation between basic and composed techniques. Basic techniques are techniques that represent the extracted similarities of several techniques. For each of these basic techniques a wrapper is defined to store the remaining information that was different for these techniques. The other kind of techniques is composed techniques, which can be by formed by putting together wrappers of the basic techniques. This design of the documentation repository allows us to reuse basic information.

On top of the models that were defined for these techniques we created using OONavigator and OOHDM some navigational views for each type of user. For each user type a guided tour was defined that made the user possible to go from one documentation item to another to receive the requested information.

The result of this dissertation is the development of a framework for

framework documentation. We are convinced that both the framework documenter and the reader of the framework documentation will benefit from this new approach to framework documentation.

We are certain that this work is not the end of the research we started. An implementation of the DocFramework should validate our design. Further research is necessary to determine in which degree existing documentation techniques are similar to each other, this to increase the reuse of primitive information even more.
Another interesting research direction we started here is the detection of documentation patterns. These patterns should help framework documenters in creating good framework documentation for cases in which our DocFramework can not be used.

BIBLIOGRAPHY

[BD97] Greg Butler and Pierre Denommee. Documenting frameworks. Position paper for 8th Workshop on Software Reuse (WISR-8), Columbus, Ohio, March 1997.

[BD98] G. Butler and P. Denommee. *to appear in Object-Oriented Application Frameworks (M. Fayad, D. Schmidt, and R. Johnson)*, chapter Documenting frameworks to assist application developers. John Wiley and Sons, New York, 1998.

[BGK+97] Dirk Bäumer, Guido Gryczan, Rolf Knoll, Carola Linienthal, Dirk Riehle, and Heinz Züllighoven. Framwork development for large systems. *Communications of the ACM*, 40(10):53–59, October 1997.

[BKM] Greg Butler, Rudolf K. Keller, and Hafedh Mili. A framework for framework documentation. To appear in ACM Computing Surveys, special symposium issue on Object-Oriented Application Frameworks.

[BMA97] Davide Brugali, Giusee Menga, and Amund Aarsten. The framework life span. *Communications of the ACM*, 40(10):65–68, October 1997.

[BMMB96] Jan Bosch, Peter Molin, Michael Mattsson, and PerOlof Bengtsson. Object-oriented frameworks. Problems & Experiences, 1996.

[CHSV97] Wim Codenie, Koen De Hondt, Patrick Steyaert, and Arlette Vercammen. From custom applications to domain-specific

framworks. *Communications of the ACM*, 40(10):71–77, October 1997.

[dFdLAC99] Marcus Felipe M.C. da Fontoura, Carlos José P. de Lucena, Paulo S.C. Alencar, and Donald D. Cowan. On expressiveness: Representing frameworks at design level. To be submitted to TAPOS, February 1999.

[DH98] Koen De Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems.* PhD thesis, Vrije Universiteit Brussel, december 1998.

[Die99] Ilse Dierickx. Capita selecta: Program understanding techniques. Capita Selecta, EMOOSE, march 1999.

[DMN97] Serge Demeyer, Theo Dirk Meijlar, and Oscar Nierstrasz. Design guidelines for "tailorable" frameworks. *Communications of the ACM*, 40(10):60–64, October 1997.

[FHLS98] Garry Froehlich, H. James Hoover, Ling Liu, and Paul G. Sorenson. *CRC Handbook of Object Technology*, chapter Using object-oriented frameworks. CRC Press, 1998.

[FS97a] Mohamed E. Fayad and Douglas C. Schmidt. Lessons learned building reusable OO frameworks for distributed software. *Communications of the ACM*, 40(10):85–87, October 1997.

[FS97b] Mohamed E. Fayad and Douglas C. Schmidt. Object-oriented application frameworks (special issue introduction). *Communications of the ACM*, 40(10):39–42, October 1997.

[GAL97] Adele Goldberg, Steven T. Abell, and David Leibs. The LearningWorks development and delivery frameworks. *Communications of the ACM*, 40(10):78–81, October 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Desgn Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GM95] Dipayan Gangopadhyay and Subrata Mitra. Understanding frameworks by exploration of exemplars. In *Proceedings of 7th International Workshop on CASE*, 1995.

[GR] Alejandra Garrido and Gustavo Rossi. A famework for extending object-oriented applications with hypermedia functionality. Technical report, LIFIA.

[Hol92] Ian M. Holland. Specifying reusable components with constracts. In *Lecture Notes in Computer Science 615*, pages 287–308. Springer-Verlag, 1992. ECOOP'92.

[Inc] Taligent Inc. White paper: Building object-oriented frameworks. http://www.taligent.com.

[JF88] R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[Joh92] Ralph E. Johnson. Documenting frameworks using patterns. *ACM SIGPLAN Notices*, 27(10):63–76, October 1992. *OOPSLA '92 Proceedings*, Andreas Paepcke (editor).

[Joh97] Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[JR91] Ralph E. Johnson and Vincent F. Russo. Reusing object-oriented designs. Technical Report UIUC DCS 91-1696, University of Illinois, May 1991.

[Kot98] Jeffrey Kotula. Using patterns to create component documentation. *IEEE Software*, 15(2):84–92, March/April 1998.

[KP98] G.E. Krasner and S.T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented programming*, 1(3), 1998.

[LK] Richard Lajoie and Rudolf K. Keller. Design and reuse in object-oriented frameworks: Patterns, contracts, and motifs in concert.

[Luc97] Carine Lucas. *Documenting Reuse and Evolution with Reuse Contracts.* PhD thesis, Vrije Universiteit Brussel, 1997.

[Mat96] Michael Mattson. Object-oriented frameworks: A survey of methodological issues. Licentiate thesis, 1996.

[MMM95] H. Mili, F. Mili, and A. Mili. Reusing software:issues and research directions. *IEEE Transactions on Software Engineering*, 21(6), June 1995.

[PLV97] Edward J. Posnak, R. Greg Lavender, and Harrick M. Vin. An adaptive framework for developing multemedia software components. *Communications of the ACM*, 40(10):43–47, October 1997.

[Rat] Rational. Uml notation guide version 1.1, september 1997. http://www.rational.com/UML.

[RDG+] Gustavo Rossi, Alicia Diaz, Silvia Gordillo, Mauricio Sansano, and Federico Arambarri. Querying hypermedia applications in an object-oriented framework.

[RG98] Dirk Riehle and Thomas Gross. Role model based framework design and integration. *ACM SIGPLAN Notices*, 33(10):117–133, October 1998.

[Sch97] Hans Albrecht Schmid. Systematic framework design. *Communications of the ACM*, 40(10):48–51, October 1997.

[SR98] Daniel Schwabe and Gustavo Rossi. An object oriented approach to web-based application design. *Theory and Practice on Object Systems. Wiley and Sons*, October 1998. To appear.

[WBWW90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

INDEX