Design Principles for Reusable, Composable and Extensible Frameworks

Jilles van Gurp
Jilles.van.Gurp@ipd.hk-r.se
http://www.student.hk-r.se/~jvg
12 March 1999

Abstract. *Frameworks have been used since the early eighties. Now that frameworks are becoming increasingly popular, several problems are surfacing. Those problems can be categorized into evolution problems (i.e. problems with changes over time) and composition problems (i.e. problems that occur when more than one framework is used in an application). This master thesis focusses on preventing these problems in an early stage in the development of a framework. Guidelines for building OO Frameworks are presented and the guidelines are tried out in the domain of communication protocols.*

Högskolan Karlskrona & Ronneby
Institutionen för programvaruteknik och datavetenskap (IPD)
Soft Center
372 25 Ronneby, Sweden
http://www.ipd.hk-r.se/

University Utrecht
Department of Computer Science
P.O.Box 80.089
3508 TB Utrecht, The Netherlands
http://www.cs.uu.nl/

Supervisor: prof. Jan Bosch

Table of Contents

Section 1 Introduction ... 1

Section 2 OO Frameworks ... 5

1 Introduction ... 5

1.1 Frameworks vs. OO programming ... 6

1.2 Blackbox vs. whitebox frameworks ... 6

1.3 Documenting frameworks ... 7

1.4 Composing OO Frameworks ... 7

2 Evolution of OO Frameworks ... 8

2.1 Changes ... 8

2.1.1 Reasons for change ... 8

2.1.2 Types of change ... 9

2.1.3 Obstacles for change ... 10

2.2 Designing for evolution ... 10

2.2.1 Structure ... 12

2.2.2 Specialization ... 12

2.2.3 Classes are similar to frameworks ... 12

3 Building OO Frameworks ... 13

3.1 Different levels of reuse ... 13

3.2 Framework boundaries ... 13

3.3 Adapting the behavior of Frameworks ... 14

3.3.1 Configuration ... 14

3.3.2 Extension/subclassing ... 15

3.3.3 Adaptation ... 15

3.3.4 Change of OS/Language ... 15

3.3.5 Framework Use levels. ... 16

3.4 Designing frameworks ... 16

3.4.1 Domain model ... 17

3.4.2 Tasks ... 17

3.4.3 Control flow ... 18

3.4.4 Component model ... 19

3.4.5 Programming language ... 19

3.4.6 Language framework ... 19

3.4.7 Other frameworks ... 20

3.4.8 Interaction with the OS ... 20

3.4.9 Other quality attributes ... 20

3.5 Design principles ... 20

3.5.1 Abstractions ... 20

3.5.2 Framework core ... 21

4 The guidelines ... 22

4.1 *Process* 22
4.2 *Design* 23
4.3 *Component* 23
4.4 *Implementation* 24
Section 3 Communication Protocols 25
5 Introduction 25
5.1 *IP* 25
5.2 *TCP* 26
5.3 *FTP* 27
5.4 *HTTP/1.0* 28
6 Communication Protocol Frameworks 28
6.1 *The X-Kernel* 28
6.1.1 Abstractions 28
6.1.2 Scheduling 29
6.2 *Conduits* 30
6.2.1 What is Conduits 30
6.2.2 The Conduits+ design 32
7 Java Conduits 35
7.1 *How to use Java Conduits* 35
7.2 *Session Conduits* 36
7.3 *Blackbox configuration* 36
8 Analysis of Conduits 37
8.1 *Limitations of Conduits* 37
8.2 *Composition-issues in Java Conduits* 37
8.3 *Evolution issues* 38
8.4 *Other quality attributes* 38
8.5 *Guidelines* 39
8.5.1 Guidelines that were followed 39
8.5.2 Guidelines that have not been followed 40
8.6 *Conclusion about the Conduits Design* 40
9 Axis Network Protocol Framework 41
9.1 *Design* 41
9.2 *Conclusions* 42
Section 4 Building Communication Protocol Frameworks 45
10 Building Communication Protocol Frameworks 45
10.1 *Building a protocol stack* 45
10.2 *Building individual protocols* 47
10.2.1 Procedural languages 48
10.2.2 The OO approach 49
10.3 *The state pattern does not solve everything* 49
10.3.1 Delegation based languages could help 51
11 Enhancing the State pattern 52
11.1 *The new pattern* 53
11.2 *An implementation of the enhanced State Pattern* 54
11.2.1 Classes 54
11.2.2 Event trace 56
11.2.3 The helper classes 57
11.3 *FSMGenerator* 57
11.3.1 XML 58

11.3.2 FSMs in XML ... 58
11.3.3 The Generator ... 59
11.4 What does the pattern solve ... 59
11.5 Composition & Evolution ... 60
11.5.1 Specialized framework ... 60
11.5.2 Loose coupling ... 61
11.5.3 Dealing with changes ... 62
12 Evolution of the FSM Framework ... 62
12.1 Nested State Machines ... 62
12.1.1 Adapting the FSM Framework ... 63
12.2 Orthogonality and broadcasting events ... 65
12.2.1 Requirements for an implementation of Orthogonal states 65
12.2.2 Implementation ... 66
13 Change scenarios ... 66
13.1 A case ... 67
13.2 Description of the changes ... 68
13.3 The state-pattern approach ... 69
13.4 The enhanced state-pattern approach ... 70
13.5 Conclusions ... 71
14 Integrating the FSM Framework with Conduits ... 72
14.1 Stack specification ... 72
14.1.1 Simplified Conduits ... 73
14.1.2 Simplified TCP ... 73
14.1.3 Simplified IP ... 74
14.2 Implementation ... 74
14.2.1 Simplified conduits ... 75
14.3 Example ... 75
14.3.1 A scenario ... 75
14.4 Experiences ... 76
14.4.1 Problems with the FSM framework ... 76
14.4.2 Composition ... 76
Section 5 Conclusion ... 79
15 Summary ... 79
15.1 The Guidelines ... 79
15.2 Existing Communication Protocol Frameworks ... 80
15.3 Addressing the shortcomings of the State pattern ... 80
15.4 Validation ... 81
16 Lessons Learned ... 81
17 Future work ... 82
17.1 Communication protocol frameworks ... 82
17.2 Framework composition & evolution issues ... 82
References ... 83
Appendix ... 85
Appendix A Meeting at Axis January 25, 1999 ... 85
A.1 Framework Usage ... 85
A.2 The Protocol Stack ... 85
A.3 Individual Protocols ... 86
Appendix B Enhanced State Pattern ... 87
B.1 FSM ... 87

B.2 FSMAction 89
B.3 FSMContext 90
B.4 FSMEvent 91
B.5 State 92
B.6 Transition 93
Appendix C TCP Connection Protocol 94

Section 1 Introduction

"Designing object-oriented software is hard, and designing reusable object-oriented software is even harder" ([Gamma], p1). This is the first sentence of "Design Patterns, elements of Reusable Object-Oriented Software", a famous book in the software engineering community by Erich Gamma et al. It could also be the first sentence of this thesis since it also applies to object-oriented frameworks.

The term object oriented framework can be defined in many ways. I will use the following definition: "a framework is a partial design and implementation for an application in a given domain" [Bosch]. So in a sense a framework is an incomplete system. This system can be used to create complete applications. Frameworks are generally used and developed when many (partly) similar applications need to be made. A framework implements the 'similar' behavior between those applications. By doing so, it reduces the effort needed to build such an application.

In [Bosch] and [Mattsson 96a] a number of problems with (domain specific) frameworks are discussed. The problems discussed center around two classes of problems:

- *Composition problems*. When developing a framework, it is often assumed that the framework is the only framework present when applications are going to be created with it. Often however, it might be necessary to use more than one framework in an application. The (possible) problems that have to be solved when two or more frameworks are combined are called composition problems.
- *Evolution problems*. Frameworks are typically designed and developed in an iterative way [Mattsson 96b] (like most OO software). After the framework is released, it is used to create applications. After some time it may be necessary to change the framework. This process is called framework evolution. Framework Evolution has consequences for applications that have been created with the framework. If API's in the framework change, the applications that use it have to be changed too.

Research Question. These two classes of problems were the starting point for this thesis. While [Bosch] and [Mattsson 96a] concentrated on solving those problems in existing frameworks, I wanted to find out whether it is possible to construct frameworks in such a way that these problems are avoided and if so, how.

To make things more concrete, I took a domain and narrowed the question to how frameworks in this domain can be improved and how frameworks in this domain should be structured.

Domain. The domain I chose to do so was communication protocols. Communication protocol frameworks are used to build protocol stacks (like in the OSI model [Tanenbaum]). The reason this domain was chosen was that it allowed me to work together with Axis Communications in Lund and Ericsson Mobile Applications Lab[1] in Ronneby who are both active in this domain.

1. Was recently taken over by Symbian

Methodology. The work was done in four phases. First I tried to analyze how frameworks should be developed in general. Based on this analysis, I made a list of guidelines and recommendations that should be followed when building frameworks.

Then I studied existing communication protocol frameworks to find out how these frameworks worked, what type of problems they tried to solve and where they failed. The frameworks I analyzed, came from both the academic world and the industrial world. For the latter Axis proved to be really helpful. They were kind enough to give me access to documentation of their framework for communication protocols.

From this domain analysis I learned that existing frameworks were trying to address two problems:

- Protocol Stack Configuration & Management
- Individual protocol implementation

Most frameworks I saw were pretty successful at solving the first problem but none of the frameworks was very successful at solving the second problem. Based on my guidelines I concluded that those two problems should not be solved in the same framework at all. So, as a third phase, I developed a separate framework to provide a solution for the second problem.

Protocols are usually represented as Finite State Machines[2], yet the frameworks I saw did not provide convenient ways to implement those finite state machines. For this reason I concentrated on providing better FSM support.

Finally, to further test my guidelines, I changed my framework in a couple of ways to test whether it was prepared for evolution. Also I demonstrated that it was a relatively easy to use framework both in development of protocols and in maintenance of protocols.

Related work. A good introduction to frameworks can be found in M. Mattsson's licenciate thesis[3]: "Object Oriented Frameworks" [Mattsson 96b]. In this work the principles of developing a framework are explained. Also a develop method for frameworks is suggested. Important for all object oriented software are the design patterns. An important book on this subject is "Design Patterns, elements for reusable object oriented software" by Erich Gamma et al [Gamma].

In "Object-Oriented Frameworks - Problems and Experiences" [Bosch] and "Framework Composition: Problems, Causes and Solutions" [Mattsson 96a] by Jan Bosch et al, composition and evolution problems in Frameworks are identified.

Important work in the area of communication protocol frameworks was done by N. C. Hutchinson and L. L. Peterson [Hutchinson][xkerneltutorial] (the x-kernel); J. M. Zweig [Zweig] (inventor of the Conduit abstraction); Hermann Hüni, Ralph Johnson and Robert Engel [Hüni] (they improved the Conduit approach by using Design Patterns) and Pekka Nikander and Arto Karila [Nikander][jacob] (who made a Java implementation of Conduits).

My improvements to frameworks for implementing protocols were largely based on the work of David Harel on Finite State Machines and statecharts[Harel 86][Harel 88].

Outline. In Section 2 composition and evolution problems in frameworks are discussed. Also a set of guidelines is presented. In Section 3 a few example protocols are presented first. After that the x-kernel and Conduits are discussed. This chapter also contains a description of the framework, Axis is developing. In Section 4 a solution is presented and validated

2. I will use FSM throughout this thesis to denote Finite State Machines
3. In the Swedish system Ph. D. students deliver a licenciate thesis after two years when they are half way their Ph. D.

for a problem found in Section 3. In the appendix a summary is given of an interview held at Axis. Also the source code for the solution in Section 4 is given.

Acknowledgments. I would like to thank my supervisor, Jan Bosch, for supporting me in writing my master thesis. Also I would like to thank Jan Mark de Haan, Matthew Kenrick and Jeroen Kolner for proof-reading my thesis. Also I would like to thank Jelte Janssons who was so kind to become my opponent at the last moment (due to sudden illness of my original opponent).

Section 2 *OO Frameworks*

Since the late 1980's, object oriented software frameworks have become increasingly popular. Software frameworks offer reusability of both design and implementation whereas traditional object oriented programming offer only reuse of implementation. By reusing the design, the software-developer is forced to design his program in a way that has proved to work before. The framework takes away the need to reinvent the wheel when designing a program.

In general "a framework is a partial design and implementation for an application in a given domain" [Bosch]. This means that using a framework in developing a software product reduces the amount of work that needs to be done and allows the developer to focus on the essence of the product being developed.

Very familiar examples of software frameworks are the user-interface frameworks provided with languages like Smalltalk or Java. These frameworks provide the developer with a model to design a user-interface. A GUI framework deals with things like Events and drawing things on a screen. The framework also provides the developer with a wide range of ready to use components like buttons, drop-down-list, etc. By extending certain classes in the framework, customized GUI components can be added.

Frameworks can be developed for all sorts of domains. Most programmers are familiar with frameworks that hide system-level functionality (i.e. the already mentioned GUI frameworks). But there are also frameworks to model things like stock-markets, alarm-clocks, mathematics. Some companies have started to wrap their domain software into frameworks. By using these frameworks, new applications can easily be developed.

1 Introduction

In the Taligent paper [taligent], frameworks are grouped into three domains:

- Application frameworks, usually a GUI with some additional functionality
- Domain frameworks, these frameworks can be helpful to implement programs for a certain domain.
- Support frameworks, these programs cover only a small aspect of an application.

This classification is very common. An example of an application framework is MFC. MFC (Microsoft Foundation Classes) is used to build applications for MS Windows. Another application framework is the Java Foundation Classes (JFC). The latter is more interesting from an OO point of view since it incorporates many ideas about how an OO framework should look. Many design patterns from Gamma's book [Gamma] were used in this framework.

However, application frameworks are only one class of frameworks. A different class of frameworks is the domain framework. One could argue that an application framework is just an instance of a domain framework. Usually the term domain framework is used to denote frameworks for very specific domains though. An example of a domain might be banking or alarm systems.

Domain specific software usually has to be tailored for a company or even developed from scratch. Frameworks can help reduce the amount of work that needs to be done to implement such applications. This allows to companies to make better software in a shorter period of time for their domain.

1.1 Frameworks vs. OO programming

Frameworks continue where OO programming stops. In traditional OO programming, reusability is achieved by collecting objects in libraries. When certain functionality in a program is needed, it can be 'borrowed' from a library. Then either the object is extended or parameterized to customize it. After that, the object can be used in the program.

Frameworks work differently, instead of borrowing objects, the framework has to be provided with the information it needs to customize it (i.e. parameters or additional classes based on the framework's classes). The thread of control usually lies within the framework, the framework calls objects as they are needed. This is also known as the Hollywood principle: "Don't call us, we'll call you".

When, for example, an application framework is used, a class may be used that represents a window. A few properties may be changed and a menu-bar can be added. When the program is used, the framework creates the window, initializes the menu, makes sure certain events are fired when the menu is used and takes care of default behavior like redrawing the window after a resize.

1.2 Blackbox vs. whitebox frameworks

Building a software framework requires a lot of effort. This effort is justified by the time and money saved when the framework is used to build applications (framework instances). Building a framework can be seen as a long time investment. Once it is finished it functions as part of the infrastructure in a company. A good framework makes application development both easier and cheaper. Developing a framework may not be justified if it is only needed for a single application. It is justified if at least three [Johnson], similar, applications have to be developed.

Most frameworks start as something small: a few classes that cover the basics of a domain. In this stage the framework is hard to use and one has to have a good understanding of the framework-design to be able to use it since it does not provide much functionality. Usually, inheritance is used as a technique to enhance such a framework for use in an application.

When the framework evolves, custom components are added that cover frequent usage of the framework. Instead of inheriting from abstract classes, a developer can now use the predefined classes, which are much easier to use. These predefined classes can be customized by changing some properties or by inheriting from one of the components or abstract classes.

Frameworks that can be used by inheritance only, are called whitebox frameworks. Frameworks that can be used by enhancing existing components, are called blackbox frameworks.

Blackbox frameworks are easier to use. The internal mechanism is hidden from the developer. The drawback is that this approach is less flexible. The capabilities of the framework are limited to what has been implemented in the set of provided components.

For that reason frameworks usually offer both mechanisms. By using the predefined components, the developer has easy access to the framework's features. If more is needed, the developer will have to make a custom component (either by inheriting from one of the abstract base classes or by inheriting from one of the predefined components).

1.3 Documenting frameworks

Whether blackbox or whitebox frameworks are used, it is essential to know how a framework works to be able to use it in applications. Of course, if a blackbox framework is used, it may not be necessary to know as much of the framework as would be necessary when a whitebox framework was used. Usually a whitebox framework provides the developer with a set of abstract classes that cover the underlying design of the framework. Because it is very hard to deduce this architecture from just these abstract classes, a framework provider should also deliver some example applications in addition to the normal documentation.

To be able to use a framework a developer will have to learn how to use it. If the learning process is to hard there is a risk that the framework is not used at all. To prevent this from happening, it is essential that there is adequate documentation for the framework. The documentation will have to show how the framework is put together. Also it will have to show how the framework should be used. Usually example programs that demonstrate the framework's features are provided.

For the description of the framework's architecture, design patterns and class diagrams can be used [Mattsson 96b][Gamma]. Design patterns appeal to a common set of ideas and terminology in object oriented design. Most OO developers know something about patterns so expressing a design in terms of design patterns, helps developers understand a framework.

1.4 Composing OO Frameworks

Most existing frameworks are built to be used as a starting point for building an application. When a developer starts using multiple frameworks, several problems may occur. In "Framework composition: problems, causes and solutions" [Mattsson 96a], a number of problems is listed and an attempt is made to provide solutions for these problems.

Possible problems in composing two frameworks (the same problems exist if more frameworks are composed with each other) are:

- Both frameworks assume to have control over the entire application. To use both frameworks the developer will have to write code to keep both frameworks synchronized. This may very well require intimate knowledge of both frameworks and may require so many changes in the frameworks that it may be more profitable to abandon the use of one or even both frameworks.
- Both frameworks provide functionality for the same real world entity. If the way this entity is implemented in both frameworks is more or less the same, a solution would be to provide an adapter class for one of the implementations that makes this implementation compatible with the other framework. However, if the implementations are different, this won't be so easy.
- The frameworks only partly cover the desired domain. This problem is also known as the framework-gap. To solve this problem, a developer will have to write mediating software. When one or both frameworks assume to have control over the application this can be extremely difficult. Moreover the resulting code will be dependent on both frameworks and will require maintenance as the frameworks evolve.
- The framework will be used together with legacy components. Adapter components will have to be written to wrap the legacy components. This can be tricky if, for instance, there is no source available for the legacy components or if the legacy components cannot be subclassed easily.

As the use and composition of software frameworks is becoming more and more popular, these problems are becoming increasingly important. Designing frameworks with future composition with other frameworks in mind, can solve part of these problems. It is not clear though what the guidelines for such an approach are.

For a more detailed discussion about composition related problems, I refer to the already mentioned paper by M. Mattson [Mattsson 96a].

2 Evolution of OO Frameworks

OO software is usually developed in an iterative way. Since software frameworks are usually object oriented, this also applies to frameworks. Development of a system does not stop after its release though. The development that takes place after a software release is also known as maintenance or evolution (mainly depending on the type of change).

After a product release the maintenance phase starts. Bugs are fixed, features are added, after some period of time a new version can be released. These changes in a system are the result of evolution. The system is changed each time to be a little better.

Unfortunately this means that when frameworks are considered, each change causes the latest version to resemble the first version a little less. For normal OO systems this would not be a problem but for frameworks this means that the programs created with older versions of a framework (framework instances) may not be compatible with the newer versions.

These incompatibilities pose a problem to the developers of a framework. They can choose to port the old applications to the new framework. This is a lot of work and probably does not improve the applications very much. They can also choose to keep the old framework intact just for those old applications. This effectively results in two versions of the framework that need to be maintained. The third option is not to change the framework. In that case incompatibilities are avoided by not creating them.

Neither option is very attractive. In practice developers will try to avoid radical changes in their frameworks. Small changes may happen as long as they don’t break the frameworks API and the applications that use it. Over time the framework will gradually become more complex. New features that are added to the framework are implemented in such a way that they preserve the API. Often these implementations are less than optimal.

This growing complexity makes maintenance increasingly difficult. Also adding new features will become more difficult. Eventually it will be nearly impossible to change the framework without breaking the earlier programs.

Most of the problems identified here are also mentioned in [Bosch]. In that paper several aspects of OO frameworks are analyzed. Another important book in OO Frameworks is [Mattsson 96b]. In this book survey is made of methodological issues.

2.1 Changes

2.1.1 Reasons for change

A common reason to change software is a software fault. When such a fault is discovered it will have to be fixed. In the case of a normal software system this is not much of a problem. When the fault is located in the framework though, the fault can only be corrected by someone who has the knowledge to do so (i.e. knows the internal structure of the framework) and is allowed to do so.

If it is done by someone who doesn’t understand the framework anyway, the fix may introduce new faults. Or worse, the framework’s carefully designed behavior will be altered. For the same reason any change to a framework should be done by somebody who understands the framework.

Several reasons can exist to change a framework:

- New functionality is needed.
- The framework needs to be restructured (for instance because there have been so many changes to the design that it becomes more difficult to do any new changes).
- Something in the domain has changed that requires a framework update.

2.1.2 Types of change

As pointed out in paragraph 1, all changes in the framework should be considered carefully. Therefore it is important that the person who conducts the changes is aware of the consequences of these changes. Examples of changes that should be handled with care might be:

- API change in the framework.
- Changes in the semantics of the framework
- Changes in the internal structure of the framework

API changes. This means that all software that uses the framework may become incompatible with the new version of the framework. Those applications depend on the API and assume that the framework works in a certain way. If the framework changes these assumptions may no longer hold true. This may manifest itself in a subtle way (for instance an application that in very specific situations behaves irregularly) or the application will simply not compile anymore.

If this type of change happens, an important decision has to be made: what applications are going to be 'ported' to the new framework. Preferably, all applications should be ported to the new framework. This is not always possible, however. Changing all applications may take a lot of time and might not improve them at all. Choosing not to update them causes maintenance costs to rise. Instead of one framework there are now two (slightly) different frameworks and their applications to maintain.

Since one of the reasons for building frameworks is being able to reuse software, this does not seem an ideal solution as well. All bugs will have to be fixed twice (if the bug is in the common part of the frameworks).

There are two types of API changes:

- *API extensions*. New methods are added.
- *API modifications*. Existing method signatures are edited or removed

The second type of change is the most problematic. All applications that used the changed or removed method are going to be affected. The first type of change has less impact unless applications are required to use these new methods (for instance, because they execute some initialization code).

Semantical change. Some changes do not require API changes but do affect the way in which the framework works. If for instance, a method is changed to do something new in addition to what it already did, this may have some unexpected effects on (some) applications.

If on the other hand functionality is removed from a method, applications that relied on this functionality are affected. Reasons for such removal could be changes somewhere else in the framework that make the functionality redundant. The effect of semantical change is hard to predict because possibly not all applications are affected. So a change that seemed to work at first, may fail with certain applications.

Internal structure change. The internal structure of the framework should be hidden from the framework users. This enables the framework users to change this structure to change it without affecting applications that respect the frameworks design rules.

Since those rules are often implicit, not all applications are programmed according to these rules and are in some way dependent on the internal framework structure. A reason for not respecting these rules might be that the framework does not offer certain crucial functionality. In order to get this functionality, knowledge of how the framework works internally is needed.

Of course the right way to do this would be to change the framework in order to incorporate the new functionality. This might be too difficult or too expensive though. An example of such a case could be a storage framework. Lets assume that the frameworks maintains a list of objects sorted on some key. For some reason the framework does not offer a method to find out what the number of objects in the datastructure is.

If a application developer needs this functionality and knows that the framework uses an ordinary array to implement the storage facility. A method could be made in the application that works on the array instead of using the framework's API. If later the framework is changed to use something better than an array (a hash table for instance) that application is broken. It will have to be fixed either by adapting the application to work on the hashtable or by adding the functionality as a new feature to the framework.

2.1.3 Obstacles for change

If for one of the pointed out reasons applications will break if a certain change is made in the framework, this might be a reason not to change the framework or to delay the change.

Since frameworks can grow quite complex, it requires some knowledge of how the framework works to make changes to a framework. If developers don't have this knowledge (for instance because of a lack of documentation) [Mattsson 96b], they might be hesitant to change the framework ("if it aint broke don't fix it").

To avoid damaging the framework, fixes can be implemented on the application level. This type of change is less reusable because other applications will have to implement the fixes as well if they are needed. This might be a reason to change the framework despite the lack of understanding of how the framework works. This way the framework's design rules may be violated, thus leaving the framework in a less usable state than before the change.

Because all of this, developers will likely develop frameworks in a conservative way. Methods are not removed, even if they are redundant out of fear of breaking applications. API changes are avoided because that would require changes in applications. The list of not implemented or badly implemented changes will grow.

After some time, the wishlist for new features, API changes and bug fixes can grow large enough to justify a framework change. Then developers do the changes and take the time to test them properly. At the same time the most important applications are ported.

So frameworks generally develop in a more revolutionary way than normal systems. Rather than changing gradually over time, a framework will change little (probably only bugfixes and minor API changes) until enough reason exists to dramatically change the framework. Then the framework is changed (in a proper way) and tested.

Each time this happens a lot of work needs to be done to upgrade all the framework instances (I use the word instance to denote applications that use the framework). If a lot of framework instances exist, it is not very likely that the framework is changed a lot. Nor is it very likely that all the framework instances are ported if the framework is changed.

2.2 Designing for evolution

Instead of dealing with evolution afterwards, evolution should be taken into account from the start of the development of a framework. Many problems can be avoided if they are recognized in the early stage of framework development.

OO frameworks, like normal OO systems, are developed in an iterative way [Mattsson 96b]. This means that a simple version of the framework evolves to a complete version by going

through several design and implementation cycles. After ‘release’ the framework is used in other applications.

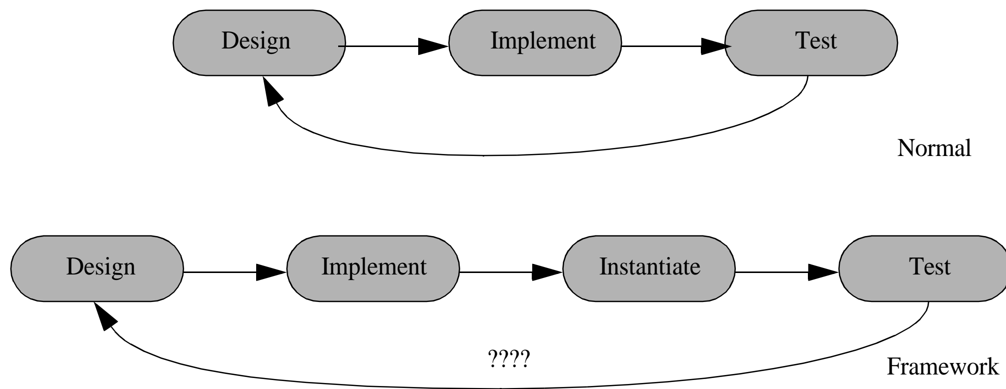

FIGURE 1. The development process.

The problem with frameworks is that in order to be able to evaluate a framework, framework instances are needed [Bosch]. Creating framework instances puts a weight on the shoulders of the framework builders because those instances become an obstacle for further development of the framework (also see paragraph 2.1). Future versions of the framework will have to be compatible to some extent with the early applications

Effectively the iterative development cycle is broken as soon as the framework is used to create applications. From then on it is no longer possible to radically change the framework without breaking existing applications (also see Figure 1). OO frameworks, like normal OO systems, are developed in an iterative way [Mattsson 96b]. This means that a simple version of the framework evolves to a complete version by going through several design and implementation cycles. After ‘release’ the framework is used in other applications.

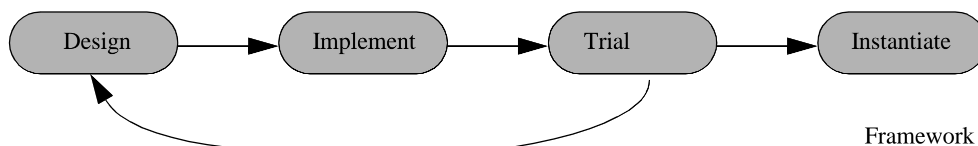

FIGURE 2. Development with a trial period.

An improvement of the process would be to give the framework a trial period (as in Figure 2) in which no guarantees are given about the API’s. This is the common practice for Java frameworks. The Java Foundation Classes [javasoft] for instance were released for evaluation long before the final release. During this trial period a lot of changes were made to both API and functionality. A few months before the final release the API’s were frozen and attention shifted to bug-fixing.

This process only works for large user-groups that are willing to play with the framework. For smaller user-groups (as would typically be the case in a company) a different approach is needed. Here it is not feasible to let users play with the framework. It is simply too expensive.

Yet applications need to be created in order to test the framework. Furthermore the applications should use all of the frameworks features. So for smaller user-groups it might be a good idea to develop the framework in parallel with a small number of applications. As features are needed in these applications, the framework can be changed to deliver them.

2.2.1 Structure

One of the things I mentioned in the paragraph 2.1, was that API changes are really bad. Especially if existing method headers are changed. So an obvious design goal would be to reduce the chance that API's need to be changed.

To do this, the API needs to be sufficiently abstract. Abstract API's do not contain any implementation details. If we recall the example in paragraph where data was stored in some way we see that the fact that an array is used to implement the array is irrelevant for the API of the data storage framework. Yet, as the same example shows, it can be dangerous to make it too abstract because then framework users have to use dirty tricks to get the functionality they want.

Making an abstract API is difficult because the developer will have to anticipate what functionality might be needed in the future. Typically an abstract API will offer more methods than are needed at the time it is created. This is where input from other developers is very important.

2.2.2 Specialization

Another thing that was mentioned is that frameworks tend to grow quite complex over time. Over time features are added, bugs are fixed and the framework becomes larger and larger. Because of the obstacles mentioned in paragraph 2.1.3, many changes will be less than optimal which means that redundant code is created, API's are preserved even though they should be replaced.

One approach to prevent this kind of framework erosion is to keep the framework strictly limited to it's domain. This means that developers should move away from the monolithical frameworks that are used today in companies towards more specialized frameworks. Just like one should not put too much functionality into a class, one should not put too much functionality in a framework.

Specialized frameworks are small, offer a high amount of abstraction for some small domain and typically depend on other (equally small) frameworks for their implementation. The reduced size and complexity allows for more elegant designs. Also the impact of API changes is not so big as in a large framework.

As the frameworks evolve and grow larger, at some point the decision must be taken to split the frameworks into multiple specialized frameworks.

2.2.3 Classes are similar to frameworks

If frameworks become smaller they start to resemble ordinary classes more and more. Like classes, frameworks encapsulate behavior that is presented through some interface. Like classes depend on other classes, frameworks depend on other frameworks. In classes information hiding and encapsulation is used to hide details about the class works from users. Something similar would be useful for frameworks.

Unfortunately the current generation of OO languages hardly supports frameworks. So dependencies between frameworks are always implicit which makes it hard to track them. Also encapsulation is hard to enforce. Some languages like Java offer some support for this though (through private classes and packages).

3 Building OO Frameworks

Before a framework is built, several questions should be answered concerning reusability and applicability of the future framework. The answers will have a large impact on the design. So it is good to be aware of those questions.

3.1 Different levels of reuse

The first thing that a framework developer should be aware of is the level of reuse that is required for the framework. Code can be re-used at different levels:

- within an application
- within a company (with a set of related applications)
- within a group of related companies (that share some software)
- the whole world

For software frameworks this means that one can assume more about the context if the level at which the framework will be used is lower. If, for example, a framework is going to be used within a certain company, part of the framework's context is the other software in use in that company. If the framework becomes dependent on this software, the framework can only be used in a situation where this software is present.

If a framework is to be applied on a higher level, it should abstract from this software either by not using it and defining the needed functionality within the framework or by providing a very abstract interface to it. The latter is more expensive but yields a more reusable framework.

3.2 Framework boundaries

Another important thing that needs to be established before framework development can be started is the domain of the framework. It is tempting to put as much functionality into the framework as possible ("the framework takes care of everything"). With such a framework it is very easy to implement programs (framework instances).

That is, as long as the framework offers everything needed by such a program. If additional functionality is needed there are a few options:

- Implement it ad-hoc in the framework instance. This approach is probably a fast solution. The result has lousy reusability properties though since it is tightly mixed with application specific code.
- Enhance the framework so that it handles the new framework ("It takes care of everything again"). This is also known as framework evolution. The result is an increase in framework complexity making it more difficult to use, maintain, compose and enhance.
- Use a third-party framework that offers the needed functionality. In order to do this the two frameworks need to be composed to work together in the framework-instance. There are several problems with framework composition (see paragraph). This is especially true if both frameworks are large.
- Make a completely new framework that works nicely together with the existing framework and offers the needed functionality. The new framework can then be deployed without any problems.

Another approach is to put as little functionality in a framework as possible. When a framework-instance is made using such a framework, it is likely that a lot of additional functionality is needed. To add this functionality, the same options apply:

- Implement it the ad-hoc way. This approach will require a lot of coding since the framework does not offer much functionality.

- Enhance the framework to incorporate the needed functionality. Since the framework is small, this should not be difficult. It would, however, violate the principle of keeping frameworks small.
- An existing framework can be used to get the needed functionality. Composition problems are less likely than before because the framework is smaller and offers less functionality that can cause problems.
- Make one or more new frameworks that offer the needed functionality.

The latter approach yields a large number of frameworks (one for each group of functionality). Because the frameworks are small, composition of two such frameworks is probably easier. Because there are a lot of frameworks though, there may still be a lot of composition issues.

If n frameworks are needed in an application, they all have to work together. So possibly a lot of code is needed to make the frameworks compatible before they can be used. If the frameworks are designed to work together, this may not be a big problem.

When setting the boundaries for the framework's domain, a choice has to be made concerning the size of the domain. If the framework has to be able to operate with a lot of other frameworks, one should consider building one or more smaller frameworks (for small domains). If on the other hand it is going to be the only framework to be used in application development, a larger framework (for the entire domain) might be an option. Personally, I think one should be careful not to put in too much functionality that is not directly related to the domain for which the framework is developed. Otherwise, too much dependencies are created between the domain and the extra functionality.

A GUI framework, for instance, has nothing to do with databases. Therefore it should not contain code to access databases. Of course there is a grey area: database aware widgets (for instance a table that reflects part of a database). Such widgets need to have access to databases and need to use the GUI framework.

If I apply my guideline to this situation, a third framework (in addition to the GUI and database framework) would be created that contains the database aware widgets. It would probably depend heavily on the GUI framework but at least the GUI framework will not be polluted by the database code. It would also depend on the database framework. By enforcing this guideline, developers are forced to think about composability (in the example above two frameworks are composed into a new framework). By making clean interfaces, the influence of evolution can be limited (also see 2.2).

3.3 Adapting the behavior of Frameworks

Behavior adaptation of frameworks can be categorized into a few different levels. Not all adaptations are as likely to happen. It should be decided at design time what levels of adapting the framework must be capable to handle.

- Configuration (parametrization)
- Extension (subclassing)
- Adaptation (wrapping)
- Change of OS/language (porting)

Allowing less change, means fewer abstractions are necessary. This greatly eases implementation and possibly allows for a more efficient implementation.

3.3.1 Configuration

At the configuration level, change is achieved by feeding components in the framework different parameters. This requires a black-box framework. Often this kind of customization can be automated. By using a special language or tool, the necessary adjustments are made.

At this level, full reuse of code is achieved. Since the framework should be designed for this kind of reuse, customizing it this way should not be very difficult.

3.3.2 Extension/subclassing

On the other hand customization by configuration may not be sufficient for advanced usage (for instance because the components do not offer all of the needed functionality). To get the needed functionality the framework will have to be extended with new behavior. To do this either existing components can be extended or the frameworks (abstract) base classes can be used for extension.

At this level both design and (part of the) code are reused. The ways in which the framework can be extended should be very clear. The frameworks internal design should not be compromised by the new extensions. Good documentation and language constructs such as 'final' or 'private' in Java, can help protect the design. Those language constructs prevent users from using methods they should not use and from extending classes that should not be extended.

Extending frameworks is not a trivial task. If the framework is not open enough (i.e. the framework does not offer enough hotspots[1] to implement the needed extensions), the programmer will have to work around this. The result will probably be ugly code that is difficult to maintain.

If the framework is too open on the other hand, the programmer might break the frameworks implicit design-rules. If this happens the new extensions will be more difficult to maintain. As the framework evolves, the extensions may not work anymore because the framework's design rules have not been respected. Alternatively the extensions can limit the evolution of the framework.

Usually frameworks start as a group of abstract classes (capturing the design philosophy for the domain). Over time the framework can evolve towards a blackbox framework. This allows for the first level of change.

3.3.3 Adaptation

A special case of framework extension is adaptation. At this level, non-framework code (for instance legacy code) has to be integrated with the framework. If the differences between the framework and the new code are large, it can be very difficult to integrate such code. Possibly a considerable amount of code will have to be written to deal with the differences. This can be eased by designing the framework in such a way that external components can be plugged in.

Providing interfaces (in the form of header files, IDL specifications or Java interfaces) for the key classes of the framework may help. Interfaces make it easier to create wrapper classes or to create alternative implementations without having to modify code in the framework.

3.3.4 Change of OS/Language

When an existing framework has to be moved to a new platform, the entire framework will have to be adapted to work with a new language/OS combination. When the language changes, the whole framework will have to be re-implemented. All that can be reused then, is the design. A change of OS does not necessarily have to be a problem. C++ code for instance can be ported to other platforms when certain standards are respected.

Of course this is not always possible, sometimes OS specific code is needed. In that case the OS specific code should be limited to just a few classes with very clear interfaces in order to keep things portable.

1. Hotspot is a term often used in framework terminology to denote the places where a frameworks can be extended (i.e. abstract classes) or configured (i.e. components).

3.3.5 Framework Use levels.

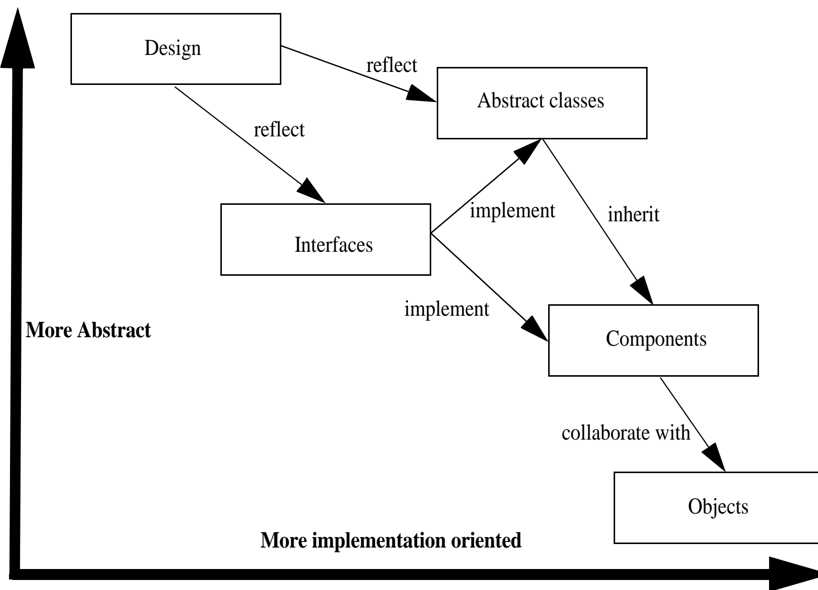

FIGURE 3. **Framework elements.**

Based on the previous analysis we can make a model of the elements in a framework (see Figure 3). The most abstract element in a framework is the design. The whitebox framework (abstract classes and interfaces) reflects this design. Then the blackbox framework uses the whitebox framework to implement components and objects. The difference between objects and components I define as having an interface or not (either in the form of a interface or an abstract class).

Adapting the framework's behavior means changing one or more of these elements. The effect of this adaptation can be described in terms of framework elements:

- *Configuration*. At this level components are used. Their behavior is changed by setting parameters using the components interface. Ideally the full public interface of the component is derived from abstract classes and/or interfaces.
- *Extension/subclassing*. The abstract classes and the interfaces are extended into new components (blackbox framework evolution). Also new objects may be created. A component could for instance use an external (for instance the language framework) hashtable class to store information.
  During this process new interfaces and abstract classes can be created (whitebox framework evolution) to make the future development of components easier.
- *Adaptation*. Components are extended to implement interfaces from other frameworks. If a lot of components need to be adapted, the abstract classes can be extended instead. This way all derived components are adapted.
- *Changing OS/language*. In the most pessimistic scenario the design will have to be changed (to deal with the OS/language specifics). Probably parts of the design will remain the same. That means that also the interfaces probably need few changes. The rest however, is likely to need drastic changes.

3.4 Designing frameworks

Frameworks can be looked upon from different angles. Each of these perspectives comes with its own requirements on quality-attributes. In this paragraph I will analyze a few of those perspectives for a few different quality attributes.

Choices made for specific perspectives can limit the framework. When for instance choosing a component model, one accepts the specifics and limitations of that particular component model. Once the design is finished, it is hard to change the component model.

The quality attributes being considered are:

- Portability (the ability to implement a design on different platforms)
- Composability (how easy is it to use other frameworks/systems in conjunction with the framework)
- Maintainability
- Probability of change (also see previous paragraph)

Different perspectives:

- Domain Model
- Tasks
- Control Flow
- Component Model
- Programming Language
- Language Framework
- Other Frameworks
- Interaction with OS

Intuitively, if a framework scores well on these quality attributes for all perspectives, both evolution and composition of that framework should be relatively painless.

3.4.1 Domain model

A framework implements domain-specific functionality. Probably the reason for building a framework is the need to be able to reuse this functionality. The domain-specific part of the framework models the different entities in the domain. Ideally, those entities should be highly portable. Little dependencies on lower level framework parts should exist.

Composability with similar representations of the same design can be difficult, especially if the other representation is radically different. On the other hand, it is unlikely that the domain will change much over time. So the representation of the domain model can remain stable for quite a long time. Because of this stability, less (perfective) maintenance is required.

3.4.2 Tasks

Concrete operations or tasks on the domain model are a different aspect of the framework. Often tasks are implicitly present in the framework. They can be represented as specific methods in some class. A task is not necessarily restricted to one method or class.

Tasks are not so easy to port to other languages because they are often defined in lower level operations (i.e. system calls, calls to other frameworks). Maintenance of tasks can also be difficult because of the dependencies on the lower level features and the distribution over multiple methods/classes.

Furthermore tasks are more likely to change over time to meet new requirements. The lower level features the tasks depend on can be grouped by functionality:

- Control Flow (i.e. message passing, event-loops)
- Language specific functionality (multiple inheritance, introspection)
- Language specific frameworks
- Other frameworks
- Component model (CORBA, COM, JavaBeans)
- OS specific functionality (green threads on Solaris, user interface, filesystem, etc.)

Those groups are not mutually exclusive. Event-loops are usually specific for a language, but they can also be something specific for an OS. The language framework itself can be OS specific. These dependencies complicate the design.

3.4.3 Control flow

The flow of control in a framework is the order in which methods are called and events are delivered. An important principle in this context is the 'Hollywood principle' ("Don't call us, we'll call you"). Frameworks usually assume to be in control of the program they are used in. This means that the framework can initiate and execute specific tasks.

In a framework instance, several components exist. For communication between those components there are two mechanisms:

- message passing
- event passing

Message passing is used when the message sending component has an explicit reference to the component that is receiving the message.

The message passing mechanism in it-self is very portable. Every language has a notion of message passing. This does not mean that the complete flow of control is portable though. Probably the flow of control is specific for a context (OS, language and component model) that is not portable.

The more complicated the flow of control, the more difficult it is to maintain a program. Probably it is a good idea to prevent that the flow of control gets to complicated. One way of ensuring that, is by using events.

Events are used when a component does not know to whom it is communicating or is communicating to multiple components. Other Components can register itself for a specific event. When that event occurs, they receive it. Events allow for a more loosely coupled network of components, which is good.

Event handling is a more advanced notion of message passing. The component that sends the message does not have to be aware of where that message is going and what will happen to it. This separates message sending from message receiving.

Composability dramatically increases when events are used. New components can be registered to listen to specific events without requiring change to the event-source. On the other hand, the event-handling mechanism is usually language- and/or platform-specific.

For instance in Java, events are specific objects. Components that respond to a specific event have to implement a specific interface. Furthermore, the component that sends the events has to take care of a registration mechanism. So the Java event mechanism causes the introduction of three different kinds of components in the framework design: event sources, events and event listeners. The choice for an event model effectively ties the whole design to a specific language and probably to a specific OS.

Since the event-handling mechanism is specific for a language and/or OS, it is difficult to port a design to another language. Though that language may have a very similar event-handling mechanism, it will probably require changes in the object model. For instance it is very Java specific to develop Listener interfaces and Event subclasses. Those things are probably meaningless in a related language such as C++. Yet they have to be part of the design of the framework.

The event mechanism it-self is not likely to change over time. The relations between components are quite dynamic so it is easy to change them. What's more relevant is whether the roles of the components will change over time. This is in my opinion more likely.

Over time conventional message passing may be changed in event-passing to establish loose coupling or vice versa to increase performance. Components can be changed to listen to specific events. Components will be changed to act as an event source.

3.4.4 Component model

One way to deal with communication between components is using a component model. Component models like CORBA, RMI or DCOM take care of message passing, events and even remote access. It is also possible to provide framework-services through a CORBA or COM bus. Interoperability of different component models further increases the range of platforms and languages that can be used. There is one drawback however. It is not always feasible to use a component model (because of performance, size, etc.).

Component models such as CORBA greatly increase the applicability of a framework. Porting the framework is no longer necessary to get the framework to work with other languages/platforms.

Maintainability decreases because extra code to adapt to the component model will have to be maintained (this can usually be automated though). It depends much on which model is used how much extra maintenance is required. In some cases the maintainability can even increase because of the use of a component model yields better structured programs (JavaBeans is an example).

Component models are designed to make composition easy, so this should be no problem. The Component model is not likely to change. And if it changes, backward compatibility is usually provided.

3.4.5 Programming language

At some point in the design of a framework, a choice will have to be made concerning the implementation language. As we have seen the choice for a language specific feature such as events is reflected in the design of the framework. Other features that are likely to be reflected in the design are abstract classes, multiple inheritance, inner classes (in Java), interfaces, metaclasses, etc.

This greatly influences portability. The choice for a language is difficult to revert. Some languages can be used on multiple platforms though. The other aspects are not influenced (though you might argue that the use of some languages should be avoided to prevent maintainability problems).

One can of course choose to use only a subset of the features in a language, but this may introduce a lot of extra programming. It would be possible not to use inner classes in Java. Inner classes are very useful though so that probably is not such a good idea since a lot of code has to be introduced to replace the inner classes.

3.4.6 Language framework

With a language, a large framework is usually shipped that offers a lot of functionality. This functionality can range from datastructures to database-access to user interfaces. Choosing not to use all this functionality is usually not an option because similar functionality is needed in the framework. But using this functionality creates another dependency between the language and the framework design.

Portability is severely limited by dependencies on a language framework, because the framework may vary from platform to platform even if the language is the same. C++ for example is used in conjunction with MFC on Windows machines. This framework is not available on other platforms so Windows programs are hard to port. As the language framework evolves, the framework may have to evolve too (in order to remain compatibility with the new version of the language).

Using language specific frameworks can also be bad for composability. In effect the framework is being composed with the language framework. Different parts are delegated to the language framework. Any new framework will have to be composed with both the language framework and the framework design. Maintainability increases because the language framework does not have to be maintained.

3.4.7 Other frameworks

In addition to the language framework, features from third party frameworks may be necessary. In general most of the things that apply to language frameworks, apply to third party frameworks. Dependencies are created, thus limiting portability. The third party framework may evolve over time requiring changes in the framework-design. Maintainability problems decrease (at least in the new framework) and composability decreases because of increased dependencies.

3.4.8 Interaction with the OS

Some programming environments go a long way to hide OS specifics. Java is an extreme example of this. The GNU C++ environment is another example. In those cases the type and version of the OS have little impact on the framework. Porting should be easy.

There are other cases though where a certain language environment is only available on a limited number of operating systems or where the environments are not compatible across different platforms. In those cases the choice of OS has an influence on the choice for a specific language and thus is reflected in the framework design.

3.4.9 Other quality attributes

Quality attributes such as performance and size may be reflected in the design. It might for instance be necessary to limit the amount of objects that is created to increase performance. Quality attributes can change over time. The framework may be moved to a faster computer allowing for a more liberal framework design.

3.5 Design principles

In this paragraph I will present some guidelines that attempt to work around some of the problem described in the previous paragraphs.

3.5.1 Abstractions

The keyword in using frameworks and OO programming is abstraction. By hiding implementation details, a piece of code is easier to handle. Also it prevents developers from making assumptions about the code. Ideally an object can only be accessed through its public interface. By doing so the implementation can be changed without affecting code that uses it (that is of course as long as the interface remains the same).

This principle can also be found in frameworks. In fact a framework can be seen as an attempt to abstract from something. A GUI framework for instance is there to assist developers to make user interfaces. It provides abstractions like windows, buttons, drop-down-menu's, etc. to do so.

The framework's domain is not the only thing that needs abstraction. As we have seen in paragraph 3.4, frameworks are developed in a context (language, OS, other frameworks, legacy components, etc.). Because both design and implementation are restricted by the context, it can be hard to adapt the framework to another context.

A solution for this problem is to abstract from the context as far as possible. This way, changes in the context (for instance a new event mechanism or a different OS) have less impact on the framework.

One approach to abstract the context away is by using a layered framework. The OS for instance can and should be abstracted from by a lower level framework. This lower level framework can be part of the language that is used to implement the framework.

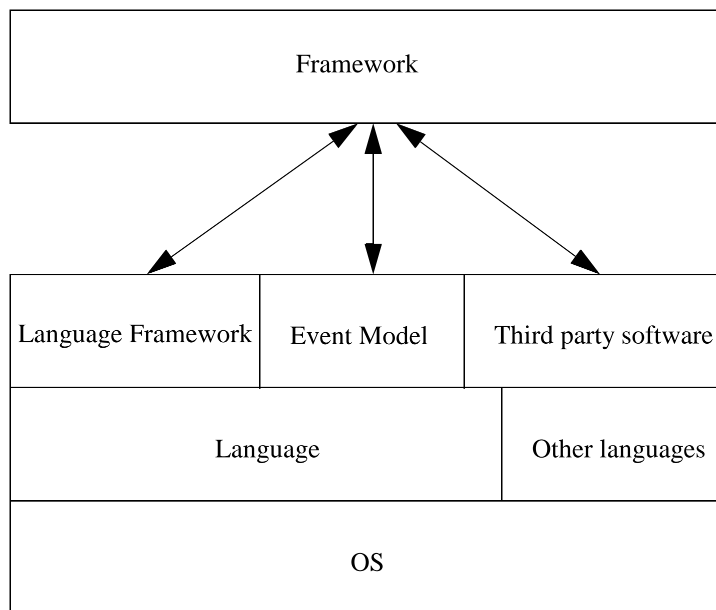

FIGURE 4. **Framework is shielded from the lower layers.**

If the framework has to be ported to another OS, all that has to be changed is the lower level framework. Once that is done, the higher levels of the framework can be ported without much change. As can be seen in Figure 4, the lower layers are abstracted from by the higher layers.

When a framework is build, it is effectively composed with those higher layer frameworks. The same composition problems diagnosed in [Mattsson 96a] manifest it-self. If the solutions to these problems are adhoc, they limit the future evolution and use of the framework. Changes in one of the lower layers (OS update, new version of language framework) may leave the framework obsolete.

Providing proper abstractions wherever possible can prevent this. However, this is not always feasible. Another approach is to rely as much as possible on standard technology. This means using technology that is widely available and is likely to be supported in the future. CORBA is a good example of such technology. It provides the proper abstractions (OS, language independence) and is available on a wide range of platforms.

3.5.2 Framework core

After successful abstraction there are three things left in the framework:

- Objects and classes representing entities in the domain (domain-entities)
- Code to manipulate these entities (tasks).
- Ready made, easy to configure components for common use

The latter only exist in blackbox frameworks.

Domain entities. As we have seen before, the domain-entities are already quite abstract themselves. This means that it is unlikely that they will have to be changed dramatically because of changes in the non-core part of the framework. To preserve this good feature it should be avoided to put functionality in the domain entity classes other than getting/setting data and editing the data structure.

Tasks. Tasks are more vulnerable to changes. Tasks are defined in terms of domain-entities, lower layer framework constructs (events, API calls, etc.) and other tasks. Tasks can be represented in many forms. They can be represented by (multiple) methods in some class. They can be represented as objects themselves. Also the way a task is initiated can vary widely. The execution may start after an event. A framework user can also explicitly start it.

The more complex a task is, the more it will be tied to the lower layers. If a task becomes too complicated to handle, it may be better to break it up into smaller tasks. It probably is a wise thing to choose one way of implementing tasks and avoid the use of multiple types of task implementations. This simplifies the framework and makes it easier to document and understand.

4 The guidelines

Throughout this chapter I mentioned solutions for common problems and do’s and don’ts in building frameworks. In this chapter I will summarize these recommendations. The recommendations can be grouped into a few categories:

- *Process*. These guidelines focus on the process of building a framework.
- *Design*. These guidelines try to improve framework design
- *Component*. These guidelines attempt to improve blackbox behavior of frameworks
- *Implementation*. These guidelines are deal with implementation details.

4.1 Process

Guideline 1 Use a trial period to finetune the framework before releasing it.

The development cycle of OO Frameworks differs from the development-cycle of normal OO systems in one crucial way: Programs created with the framework become dependent on the framework and thus the framework can’t be changed without affecting those applications.

Guideline 2 Predetermine the context in which the framework will run

The context in which the framework is going to run is going to influence its design. Developers should ask themselves questions about where the framework is going to be used (within an application, within a company (with a set of related applications), within a group of related companies (that share some software), the whole world. About what technology is going to be used. Choice of target-platform and implementation language influence the design. About future developments. Perhaps another department is implementing something that may have to be integrated into the framework later.

4.2 Design

Guideline 3 *Reduce the chance that APIs need to be changed*

API changes are bad for all applications that use the framework since they will all have to be made compatible with the changes. Sometimes it can't be avoided to change the API though. API extensions dont't have to be a problem but API enhancements are bad.

Guideline 4 *Keep the framework strictly limited to its domain.*

It is very tempting to add all sorts of things to the framework. So over time things may be added which in principle are independent of the domain. Yet dependencies are created between those things and the framework.

Guideline 5 *Specialize the framework for a very specific domain.*

Use the unix philosophy to do only one thing but do it extremely well rather than putting all sorts of functionality into the framework.

Guideline 6 *Split the framework and the domain if the domain gets too large.*

If the framework gets too large, it might be necessary to create an additional framework. The resulting frameworks will be smaller and have cleaner designs. This makes them more flexible and easier to maintain.

Guideline 7 *Try to accomplish loose coupling by using events for instance, rather than hardwiring relations between components*

Interested components can register themselves at the broadcasting component (like in Java). This makes it easier to re-arrange components in a system and allows for greater flexibility. This flexibility is especially useful when composing the framework with other frameworks.

Guideline 8 *Favor delegation above inheritance because that allows for blackbox configuration*

For the same reason, delegation should be favored above inheritance. Though inheritance can be useful at some times it effectively hardwires relations between classes in the code.

Guideline 9 *Break large tasks up into smaller ones. This makes the tasks more reusable and makes it simpler to make new tasks based on the tasks already present in the framework.*

Smaller tasks are more easy to handle. Also the chance that the individual parts can be reused is greater than the chance that a large task can be reused.

4.3 Component

Guideline 10 *Provide access to the frameworks blackbox behavior through as few classes as possible.*

This simplifies the interaction with the framework because now developers only have to know details about a few classes. Also see fine grained-classes in [Johnson].

Guideline 11 *Make configuration of the components in a framework easy by providing convenience methods, default behavior or even a configuration language/tool*

The easier it is to configure a framework, the more likely it is that people will use it. The use of tools/configuration languages has the additional advantage that the input for these programs can be reused if the framework changes (provided that those tools are ported).

Guideline 12 *Use interfaces/headerfiles to abstract from components. Each components public interface should be available in the form of either an abstract class or*

an Interface file (Java). Also other components should refer to these interfaces instead of the component classes.

This way it is possible to re-implement components without touching other components. Also it is possible to have multiple implementations for an interface.

4.4 Implementation

Guideline 13 *Hide internal implementation of classes and components by using language features as final or private.*

This prevents users from touching things that should not be touched. It also clarifies the way in which the framework should be used.

Guideline 14 *Document the code.*

Often the source code is the documentation. So documenting it well may help people to understand the framework. In Java the use of the JavaDoc sourcecode documentation generator is strongly recommended.

Guideline 15 *Limit the amount of platform specific code in order to increase portability.*

This may not always be possible, but in general it is a good idea not to tie the framework to the platform it runs on. This way it is easier to adapt the framework to new platforms in the future.

Guideline 16 *Use standard technology rather than adhoc solutions.*

Standard technology is less likely too change over time and is more likely too be supported for longer periods of time. It also increases portability.

Guideline 17 *Make domain entities independent of the underlying system as much as possible.*

Keep domain entity classes clean of implementation details. The domain entity classes are not likely to change much.

Guideline 18 *Don't make the control flow too complicated. Avoid using large if statements. Case statements should not be used at all.*

Programs with a complicated control flow are hard to maintain and also hard to understand. Those two thinks are bad to have in a framework so it is best to avoid complicated a controlflow.

Section 3 *Communication Protocols*

As a domain to test my guidelines concerning the design and implementation of frameworks, I have chosen communication protocol frameworks. Communication protocol frameworks are used to implement protocol-stacks. The initial reason for choosing this domain was that it allowed me to work together with Axis and Ericsson (who are both active in this domain).

Communication protocols (for example IP or TCP) generally work together in something called a protocol stack. A protocol stack is a layered structure. Each layer may contain several protocols. The layers communicate by sending each other messages and data packets. Each layer adds a header to the data when sending it down the stack. The layer's header is removed when the packet moves up the stack.[1]

The OSI model [Tanenbaum] is an abstraction of how a stack should be organized. In this model seven layers exist. Each layer represents a group of functionality that should be implemented by protocols in that layer. Most existing protocol stacks do not adhere fully to this model (for instance because layers are combined) but it's still a nice model for protocol stacks.

5 Introduction

For various reasons the application of software frameworks in the domain of communication protocols has only been partly successful. While existing frameworks do encourage the reuse of a particular design in communication protocol software, they fail when it comes to code reuse. Typically a communication protocol framework only accounts for a fraction of the LOC in an actual protocol implementation (see Appendix A).

Before looking at some frameworks I will briefly introduce a few common protocols that can be implemented using communication protocol frameworks.

5.1 IP

The purpose of IP (Internet Protocol) is to move datagrams through an interconnected set of networks [rfc791]. This service is not guarantied to be reliable (packets may be lost, data may be changed). Reliability (i.e. the safe delivery of data) is a service that must be provided by higher level protocols (such as TCP). The minimum requirements for an IP module are specified in [rfc791]. A description of the IP header and its fields can also be found there.

The model of operation is that IP datagrams travel through the network passing so called Internet Modules. An Internet Module is a piece of hardware that understands the Internet Protocol. When some computer wishes to send a datagram to some other computer on the Internet, the datagram's header is provided with the destination address. The datagram is then send (using the

1. This is only true for lower level layers. In higher level layers protocols like http don't send datapackets but use a datastream provided by a lower level protocol (TCP).

system's network interface such as Ethernet) to the destination computer (if it's on the local network) or otherwise to the local Internet gateway, which understands IP. From there it is send to through the Internet towards the destination computer. On its way it may pass several gateways. Each of those gateways will forward the datagram in the right direction using the information in the IP header. If necessary (for instance because the network can only handle packets of a certain size), the datagram is fragmented into smaller ones. The receiving Internet Module takes care of putting the pieces back together.

So IP provides two mechanisms for sending datagrams: addressing and fragmenting. IP addresses are 32 bit numbers that uniquely identify a computer on the Internet. There are three coding schemes for IP addresses (each 32 bit). They are called class a, b and c addresses respectively.

The fragmenting or multiplexing of datagrams requires some changes in some fields in the IP header. Those fields are used when the fragments have to be put together again.

5.2 TCP

TCP (Transport Control Protocol) can work on top of any other lower level protocol but is usually used in conjunction with IP. It provides a reliable host to host connection across the local network or a set of interconnected networks. This means that all information sent over the connection will either be delivered or TCP will detect it was not delivered (and re-transmit it).

The minimum requirements for a TCP implementing system are specified in [rfc793]. A description of the TCP header and its field can also be found there.

The TCP protocol forms the basis for a number of other protocols like FTP, Internet Mail, Telnet and HTTP. It forms the backbone of the internet together with IP.

The TCP protocol was designed to be extremely reliable: "As long as the TCPs continue to function properly and the Internet system does not become completely partitioned, no transmission errors will affect the correct delivery of data. TCP recovers from communication system errors." TCP specification [rfc793], page 4.

To provide reliability each byte that is sent has a sequence number and acknowledgments for each correctly received datagram is sent to the originating host. If an acknowledgment (ACK) is not received within a period of time, the entire datagram is sent again.

The ACK also contains the sequence number of the byte that is expected next. This way the sender knows that a datagram has been received correctly by the receiving TCP. This receiving TCP is then responsible for delivering the datagram to the addressee (if it's not the addressee).

The receiving host knows when it has received the last datagram of a transmission when a special field in its TCP header is marked. By looking at the sequence numbers the TCP module can also determine whether it has received all of the datagrams.

A TCP connection between host A and B is defined by the IP address of host A, a port on host A, the IP address of host B and a port on host B. A port is 16 bit number. Ports are used to allow multiple connections to a single host at one time. The combination of a host's address and a port number is called a socket.

Special messages are used for the communication between the application and the local TCP module.

An application can request the TCP module to establish a connection with another socket. What happens then is called an active OPEN. The module sends a message to the addressee. The addressee answers with another message, thus confirming it received the request for a connection. Then a message stating that the connection is established is sent to the

addressee. This procedure is also known as the three-way handshake. To close a connection similar approach is used.

An application can also wait for another host to connect to its host. This is called a passive OPEN. When a host performs a passive OPEN, it will wait until another host tries to connect to the specified socket. By leaving the port number unspecified (all the bits zero), the host will accept connections on every port. If specific port is specified, only connections to that port will be accepted.

Once a connection is established, the processes on both ends of the connection can communicate (by using the SEND message). The TCP waits until enough data has been collected to perform the actual sending of the data. An immediate send can be enforced with the PUSH message. If such a message is passed to the TCP module, it will start transmitting all the data that has not yet been sent.

5.3 FTP

FTP (File Transfer Protocol) is a protocol that has been used for a long time on the internet. It is used to transfer files over the internet.

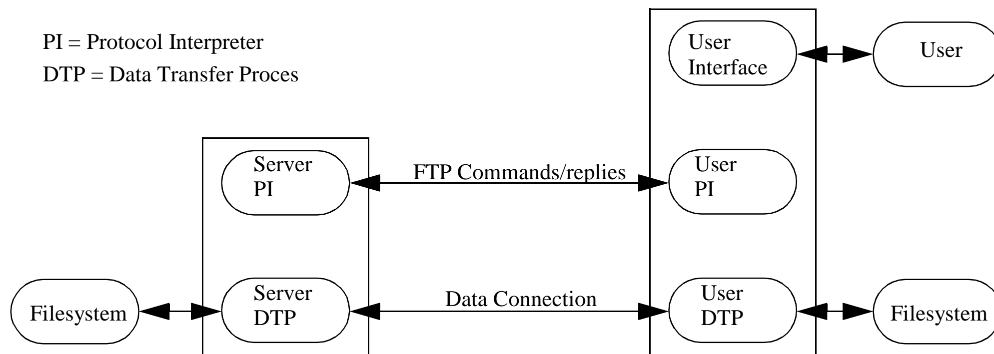

FIGURE 5. FTP usage model.

The diagram in Figure 5, was borrowed from [rfc765]. It displays how FTP is used. FTP has a client side and a server side. FTP commands are sent between two protocol interpreters over a control connection. This control connection is established using the Telnet protocol.

The control connection is used to set the parameters for the dataconnection (i.e. datatype, filename, directory, port etc.). In a DTP there is a passive and an active side. The These can be either the server or the user side. If the user is passive, the server will initiate the data transfer (download) if the server is passive, the user will send data (upload).

Besides transmitting data, FTP also can perform some translations on the data (i.e. EBCDIC to ASCII). There are three transmission modes:

- one that formats the data
- one that compresses the data
- one that just passes the data

The transmission mode is set by sending commands over the control connection (for a list of commands see [rfc765]).

5.4 HTTP/1.0

Unlike FTP, HTTP is not connection oriented. It usually runs on top of TCP. But nothing in the HTTP specification prevents it from being implemented on another protocol. HTTP just assumes the connection is reliable.

HTTP works on a request/response basis. This means that a client connects to a server, sends a request, receives an answer and then disconnects. If a connection is aborted before a request is finished, the request is just canceled.

In the simplest form there is a direct connection between client and server. But it is also possible that there are intermediary servers (for instance proxy servers or caches). Requests can be handled in any of the intermediate servers. A proxy-server may for instance handle a request by giving something back from it's cache.

A request contains:

- A request method
- URI (uniform resource identifier)
- protocol version
- MIME like message containing the request modifiers, client information, and some optional data

The server responds to the request by sending a status line (protocol version, error codes/ success code) followed by a MIME message with data.

6 Communication Protocol Frameworks

In this paragraph I will examine three existing frameworks for communication protocols. All of them find their origin in the academic world.

6.1 The X-Kernel

In the x-kernel [Hutchinson], an attempt is made to build a framework of services that make it easier to build network protocols. As it is written in C, the design is not object oriented. Yet it does provide a few useful abstractions.

6.1.1 Abstractions

The main abstractions in the x-kernel are protocols and sessions [xkerneltutorial]. A protocol is an object[2] that implements a protocol specific interface and a so called peer to peer interface. The latter can be used to set up a connection to a peer. In the case of TCP this would be the TCP layer on the machine on the other side of the connection.

2. Though the x-kernel is a C program I will use the word object here since the program is object based. Note that the usual OO concepts like instantiation, inheritance and polymorphism are not applicable here.

The protocol module also support de-multiplexing. This means that messages are dispatched to a session.

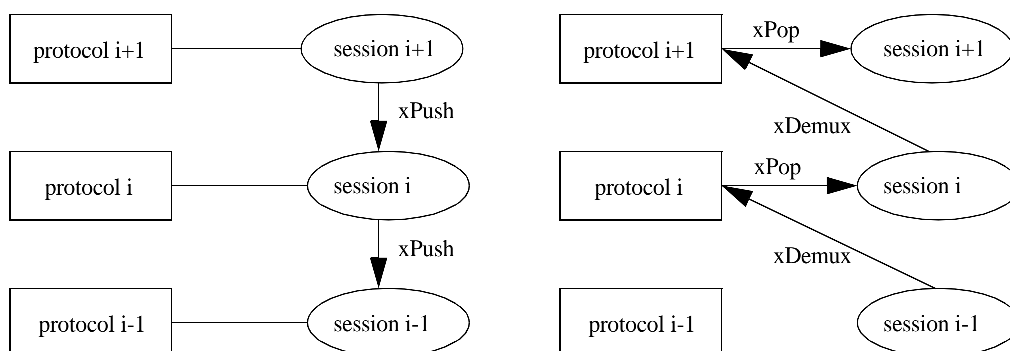

FIGURE 6. xPush, xPop and xDemux.

In Figure 6 two situations are shown. In the first data is pushed onto a connection. Session i+1 represents a dataconnection of protocol i+1. At one moment there can be several active sessions for one protocol. The dataconnection is virtual. That means that the session is not communicating directly with its peer on the other machine but instead it uses lower level protocols to communicate to the other side.

These lower level protocols also have session objects of course. So if session i+1 wants to talk to its peer, this results in one or more calls to the session below. This is done by calling a special method called xPush.

On the other hand if data moves up the stack, there may be more than one session available. For instance a IP session would have to choose the correct TCP session based on the port information in the TCP header. To do this a xDemux method is provided. This method (implemented in the protocol object) picks the right session object to deliver the data to.

So when session i-1 wants to deliver something to protocol i, it just calls the protocol's xDemux method. This method then selects the correct session and calls that session's xPop method to indicate that information was received from below.

The xPush and xPop methods are used to move packets of data up and down the stack. The xPush method interprets and strips a header of the packet. If for instance a HTTP session would push a TCP packet down the stack. The TCP session would look at the header, do some things (setting a timer for instance, setting some information in the header) and than would add a IP header and push it down to the IP session.

If on the other hand an IP packet is popped to a TCP session (using the TCP protocol's xDemux), the TCP session would remove the IP header, read the TCP header which was added by the peer protocol and perform some actions. After that it would pop it further up the stack (where it would be xDemuxed to the HTTP session). So xPush and xPop together implement a protocol.

The xKernel defines several other methods like for instance creating and destroying connections (xOpen and xClose). Two layers in the stack communicate through a uniform protocol interface.

6.1.2 Scheduling

There are essentially two ways to apply multit-hreading in a protocolstack [xkerneltutorial]:

- Thread per protocol
- Thread per message

In the first approach each protocol runs in its own thread. When a message travels through the stack, a so called context switch takes place when the data leaves the thread of the protocol it is currently at and enters the thread of the protocol in the next layer. These context switches are generally expensive.

In the second approach a thread is assigned to each message that travels through the stack. This means that when a message enters the protocol stack, a thread is launched. Then the message is moved through the stack (interacting with protocols and sessions on its way bay this thread. This way of threading is more efficient than the first approach because less context switches take place [xkerneltutorial].

6.2 Conduits

An example of an OO Framework for communications protocol is the Conduits+ framework [Hüni]. Conduits+ tries to make building protocols and protocol stacks easier by providing an object representations of a protocol stack. Traditionally object orientation has been avoided in protocol implementations because of its notoriously bad performance. Zweig et al however, believed that this should not be a problem and designed the Conduits framework [Zweig](a predecessor of Conduits+).

The Conduits framework can be used to implement message based communication protocols. The framework takes care modeling protocol stacks and transporting messages and data through the protocol stack. In this paragraph I will discuss the design of this framework. and a successor of this framework: Java Conduits[Nikander].

6.2.1 What is Conduits

The Conduits Framework was originally developed at the University of Illinois as part of the Choices operating system [Zweig]. It was a framework that was used to implement protocols like TCP/IP. Later Herman Hüni et. al. [Hüni] developed the Conduits+ framework which improved on the concept and made the framework more flexible.

Conduits+ is a so called blackbox-framework. The framework provides components called Conduits that can be connected together into a Conduit graph. A conduit processes and passes data to other conduits. Four types of conduits are defined in the Conduits+ framework:

- *Protocol Conduit*. This conduit contains a finite state machine[3] for a specific protocol. The FSM processes messages that are delivered to it by conduits on either side.
- *Mux Conduit*. This protocol functions as a connection between two other conduits. It has two sides. On one side A, only one conduit is connected. On side B, multiple conduits can be connected. A Mux has an associated Accessor object that decides to which protocol a piece of data is going to be dispatched.
- *Adaptor Conduit*. this conduit adapts to some specific interface. In a network protocol, this might be used to wrap the hardware interface. The Adaptor converts data and API calls from outside to objects in a format that is understood by the other conduits.
- *Conduit Factory*. This conduit is associated with a Mux. If a conduit has to be added to a Mux, this factory is used to create it and dynamically install it into the mux (for more details on this procedure see paragraph 6.2.2).

3. I will discuss implementing individual protocols in more detail in Section 4.

These four conduits are implemented as blackbox components. This means, that the components can be used by providing parameters. Hüni describes the framework in detail, using design patterns, in his article [Hüni].

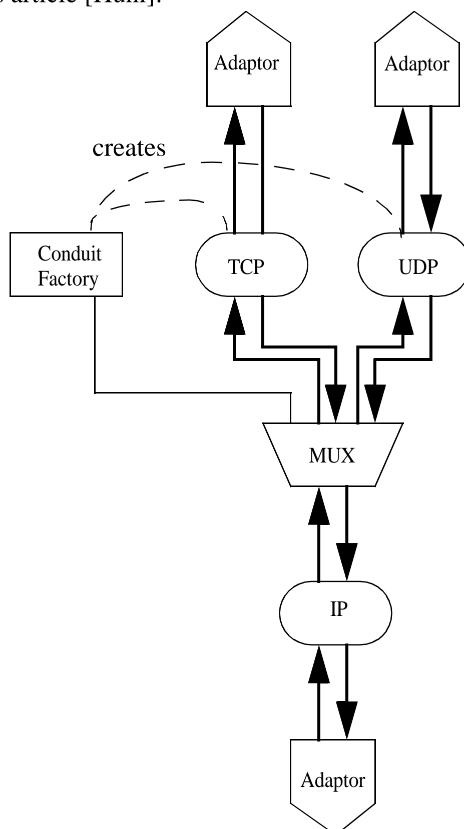

FIGURE 7. **An example conduit graph.**

In Figure 7 an example conduit graph is drawn. It models a stack with the IP protocol and on top of that the TCP and UDP protocol. On the top and on the bottom of the stack are adaptor components that interface with the 'outside' world (for instance a network driver at the bottom and a web-browser at the top). The arrows indicate the direction in which the data flows.

A small example: IP packets are put into the graph at the adaptor at the bottom. This conduit converts them to Messages for the IP conduit and passes them to the IP Conduit. The IP conduit collects the packets and after a while can make a TCP packet out of the received packets. This new TCP packet is passed on to the Mux which now uses its Accessor to determine to which of the conduits the packet is to be sent. The TCP conduit is chosen and the Mux gives the packet to the TCP Conduit. This conduit extracts data from the packet and passes it on to the adaptor component above which gives the data to some application.[4]

The Conduits+ framework is designed in such a way that all of the Conduits can be blackbox components. This means that they have fixed API's which is useful if they need to be connected together automatically. This also means that other objects will have to do the protocol specific things.

The protocol conduit for instance delegates behavior to State objects (following the State Pattern [Gamma])[5]. The Mux uses an Accessor object to determine how to dispatch incoming data.

4. This is a very much simplified example. In reality the ConduitFactory can be used for session management (more about this in paragraph 7.2).

5. The State pattern and other finite state machine implementations are discussed in Section 4.

The Accessor is also used by the ConduitFactory to create new protocol conduits. Based on the Accessor's output a prototype of a protocol conduit is chosen, cloned and installed on the Mux.

6.2.2 The Conduits+ design

In this chapter I briefly describe the design of the conduits+ framework. By using event-traces I'll try to explain how conduits works. The figures presented are derived from the description presented in Hüni's article [Hüni].

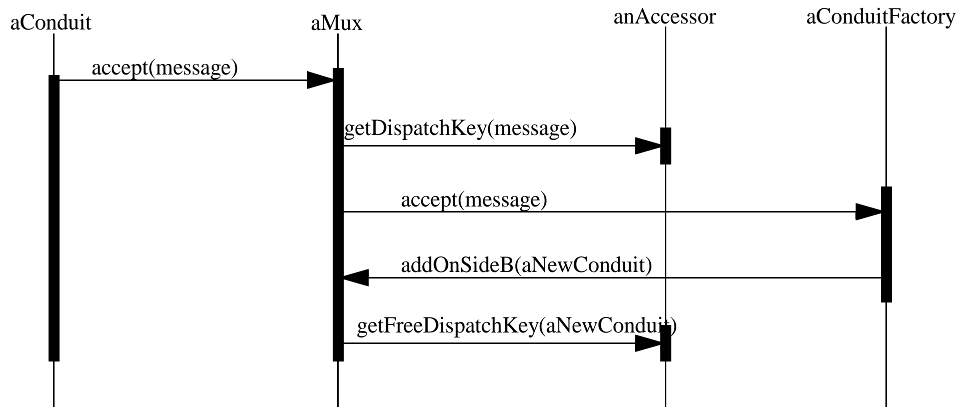

FIGURE 8. Adding a Conduit to a Mux.

Adding a Conduit to a Mux. The sequence of events starts when a message arrives at a Mux. The Mux uses it's Accessor a to get a dispatchkey. Since no Conduit is installed for this particular message, the key for the default conduit is returned (this is of course the ConduitFactory). The mux then passes the message to the ConduitFactory on b0, which then adds an appropriate Conduit to the mux.

The ConduitFactory uses the prototype pattern to create the new Conduit. This means that it has a prototype of every Conduit that might be requested. The ConduitFactory determines the type of the new Conduit by looking at the informationchunk. It then returns a clone of the requested type of Conduit.

The mux requests a free slot for the new Conduit and uses it to install the new Conduit. The mux then informs the ConduitFactory that the add-operation was successful.

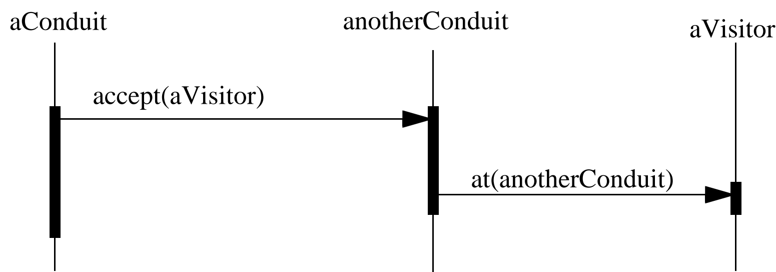

FIGURE 9. Visitor arrives at a Conduit.

Visitor arrives at a Conduit. What happens when a piece of data arrives at a Conduit depends on both the type of Conduit and the type of data. To implement this, the Visitor Pattern [Gamma] is used. Each informationchunk is wrapped in a Visitor object.

Each conduit has an accept method[6] that calls back the visitor (using the at method). Whenever a visitor arrives at a conduit, the conduit calls the vistors at method. This polymorphic method selects on the type of conduit that is given as a parameter. So a Vistor has four at() methods (one for each conduit) that can be used to specify different behavior for each type of conduit.

To do something specific at a certain conduit, specific subclasses of the visitor have to be created. Examples of visitors that are likely to exist are visitors that deliver a packet to a specific protocol conduit, visitors that are used to configure an adaptor and visitors that are used to install a newly created protocol conduit on a Mux.

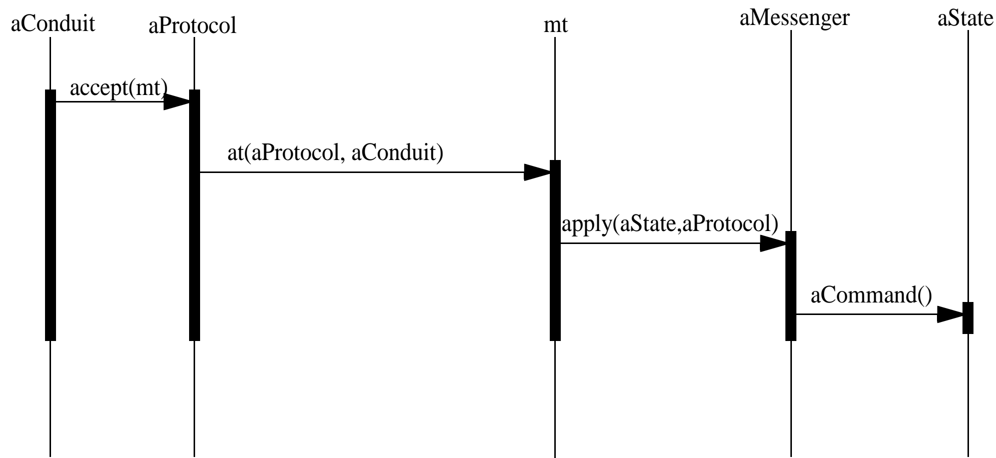

FIGURE 10. Message arrives at protocol.

Message arrives at protocol. When a datachunk arrives at a protocol, several things can happen:

- The datachunk contains a packet that needs to be processed by the protocol
- The datachunk carries some message to change the Conduit (for instance a request to add a certain Conduit to a Mux)
- Etc.

To distinguish these cases, the Visitor pattern is used (see also the previous event trace). In Figure 10, a subclass from Visitor is used (MessageTransporter) to deliver a message to a protocol. To do that, it uses the encapsulated subclass of Messenger, which knows how to talk to the protocol specific state objects.

Messenger objects are used to model the different messages a protocol understands. A messenger carries a message, generally a data packet for a protocol and behavior. The behavior part of the Messenger is defined in the polymorphic apply method.

This method is called by the Visitor's at method. The at method first retrieves the protocol conduit's current state and then calls the apply method. The apply method's behavior is selected based on the State (using polymorphism). This way a case statement is avoided to select the behavior.

The selected apply method uses methods in the state object to perform a state transition. As a result new Messenger objects may be created to be put into the conduit graph. Also the protocol conduit may change state (one of the other state objects is set as the current state in the protocol conduit).

6. I left out the parameter that indicates from which direction in the stack the data arrived.

Objects in Conduits+. There are several objects in the framework that need to be customized to implement a protocol. So far four types of Conduits have been presented:

- Protocol Conduit
- Adaptor Conduit
- Mux
- ConduitFactory

Other classes in the framework are:

- Visitor
- Messenger
- State
- Accessor

Those classes can be specialized to add functionality. For instance a protocol can be represented as a collection of subclasses of State and a set of associated subclasses of Messenger. The Messenger objects are closely tied to the state objects of a particular protocol. In addition to that Accessor subclasses are needed for each Mux in the graph.

To traverse the conduit graph, subclasses of the Visitor class are used. Each Visitor has a specific task for a each type of Conduit (the four at<SomeConduit> methods). One of the tasks might be delivering Messenger-objects at the right protocol. Other tasks might involve installing/deleting conduits from a specific Mux.

In Figure 11 an example is given of the delivery of a message to a protocol. Though the example is fictive, I have used names that suggest it could be part of a TCP implementation.

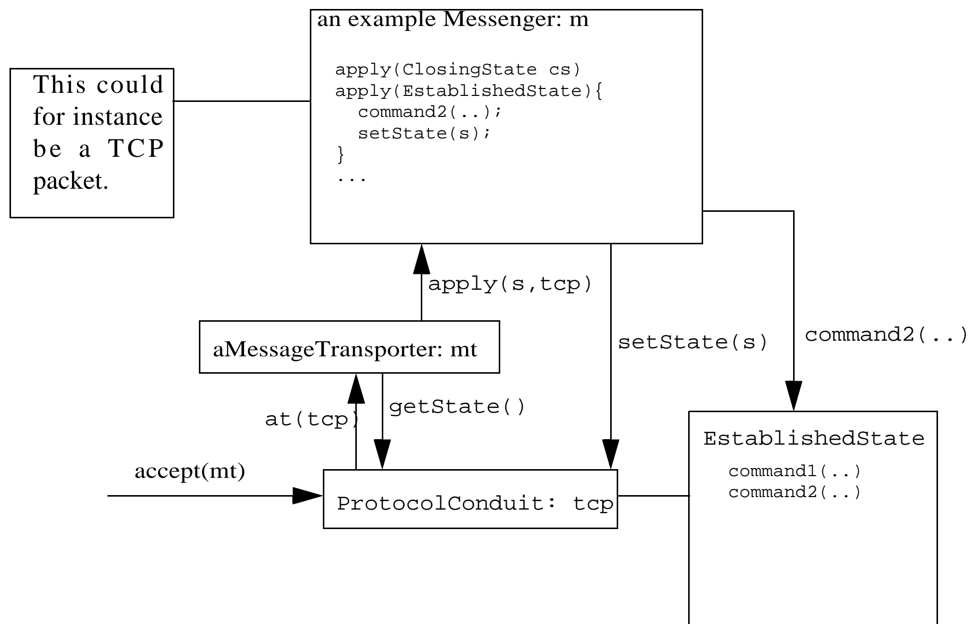

FIGURE 11. **Example of message delivery.**

In the example a Message Transporter mt arrives at a Protocol Conduit called TCP. The MessageTransporter uses it's embedded Messenger to do whatever the Messenger needed to do at this protocol. In this case a method is called in EstablishedState (which is the current state of TCP). In addition to that a different state is set as the new current state.

7 Java Conduits

The Java Conduits framework improves on the concepts presented in Conduits+. It is currently being developed in a research project at the University of Helsinki. The two most important differences with the Conduits+ framework [Nikander] are:

- The Conduit+ conduit protocol has been renamed to Session. The motivation for this is that a Session does not represent the whole protocol but only a single active connection or socket.
- A new conduit has been introduced: the Protocol (this is not the same as in Conduits+). It is also referred to as MetaConduit. It is used to group the components in one layer together.

The main contribution of Java Conduits is that it further reduces complexity of protocol implementations by splitting them up in multiple sub protocols. Those sub-protocols can be grouped together using the Protocol conduit.

One thing that is not explicitly mentioned in Hüni's article [Hüni] is scheduling. Scheduling is needed to take advantage of Threads. Java Conduits uses threads for greater scalability (how this is done is explained in paragraph 7.2).

The Java Conduit framework takes care of the plumbing needed to fit the different parts in a protocol stack together. In that it does a good job. The framework makes extensive use of Java language features like garbage collection, Java Beans, multi-threading and security. By using these features some of the traditional implementation overhead like fixing memory leaks, is eliminated.

Also it is very easy to connect the conduits together to form a protocol stack (because they are JavaBeans). In fact it should be possible to make a graphical tool for building protocol stacks from ready made components [Nikander].

7.1 How to use Java Conduits

Java Conduits has also been used to implements TCP/IP. An alpha version is available for downloading [jacob]. Because it contains native code (probably in the adaptors) for Solaris and because the implementation was based on an earlier version of Java Conduits (no longer available), I failed to get it to work.

The implementation contained several protocols (TCP, IP, UDP, ICMP, Ethernet). Also a few example programs were provided. Those applications revealed how simple it was to construct a conduit graph. To implement a protocol using Java Conduits the following things have to be done:

- The protocol has to be split up in subprotocols representing the separate steps in processing an information chunk. The IP protocol for instance fragments bigger chunks into smaller ones when sending and puts them to getter when receiving. This can be implemented as a subprotocol. The IP protocol would be implemented as a stack of subprotocols.
- When the subprotocols have been identified, for each sub protocol a FSM has to be designed.
- For each state, a subclass of State should be made. State objects are stateless which means that they are passive.
- For each state-transition a messenger object has to be made that implements the state-transition. The implementation can use global services and protocol specific services to implement its behavior. Furthermore behavior can be shared with other messengers by using inheritance. A typical example for this would be the ability to read fields in an IP header. Since most messengers will have to use this feature, it can be useful to abstract this in an abstract IPMessenger.
- When all the sub-protocols have been defined, they can be composed with other (sub) protocols. This process has to be done manually at this time. A commercial version of this framework would probably have a graphical tool or a scripting facility to do this.

- If the protocol is at the bottom or the top of a protocol stack, it has to have an interface to other applications or hardware. To do that, Adaptor conduits have to be made.
- If the resulting conduit graph is complex, it might be useful to encapsulate it in a Protocol Conduit so that it can be reused in other protocol stacks.
- Accessor classes need to be created for the muxes in order to be able to do multiplexing.

The first two steps are usually quite straight forward because protocols are typically well specified. The third step can be more difficult depending on the availability and quality of previous implementations of the protocol.

7.2 Session Conduits

One reason the Protocol Conduit (in Conduits+ changed name to Session Conduit was to reflect that a protocol can be used multiple times in a system. TCP for instance allows multiple connections on a machine (up to 30000). Java Conduits would create a Session Conduit for each of those connections.

For each connection a session conduit is created. The session conduit can also be used for connection-less protocols. In that case only one session object is created. Session objects can created dynamically from a prototype by the ConduitFactory. The Mux to which this factory is connected takes care that the data is delivered to the correct Session Conduits.

The word Session suggests that threads are involved. This however not true. There are two ways to schedule threads in Conduits.

- Conduit Oriented Scheduling. This the way threads are scheduled in Conduits+ [Nikander]. All Conduits run in their own thread. When Conduits exchange visitors a so called context-switch takes place. This way of scheduling causes many threads to be launched. Also it causes many context-switches which is very bad for performance
- Visitor Oriented Scheduling. Java Conduits uses a different strategy. In Java Conduits each adapter conduit has it's own thread. This thread carries Visitors through the Conduit graph. All the other conduits are not threads. The methods in the Adaptor Conduit are synchronized (using the Java synchronized keyword) to prevent synchronization problems. This way of scheduling reduces the number of threads and context-switches. So Session Conduits do not run in their own thread (as the Protocol Conduits in Conduits+).

These ways of threading are very similar to the way the x-kernel [xkerneltutorial] handles threading. Note that the option chosen for Java Conduits is slightly different than the x-kernel's way of threading.While the x-kernel launches a thread for each incoming message, Java Conduits has a thread running for each adaptor conduit. The consequence of this approach is that less threads (and less context-switches) are needed. The only limitation is that while a message is moving through the stack, the adaptor that it originated from cannot insert new messages. So in a system with many layers. A message at adaptor X must wait until the previous message at that adapter has moved through all the layers.

7.3 Blackbox configuration

Java Conduits was designed to be used in both a whitebox way (implementing protocols) and a blackbox way (creating protocols stacks). Creating protocol stacks in a blackbox way is an improvement over the traditional way of hardwiring the relations between protocols.

This way protocol stacks can be customized for different situations using the same code base. One could imagine building a protocol stack for TCP/IP. Than later when a UDP protocol is implemented, an UDP/IP protocol stack could be made (using the old IP implementation) by simply parametrizing the framework in a different way.

This parametrization could be done by a tool. This would make the process of creating protocol stacks even more easy. The creators of Java Conduits actually stated that such a tool could be created for their framework [Nikander].

8 Analysis of Conduits

Conduits+ and its descendant Java Conduits are the most advanced OO frameworks I found for the domain of Communication Frameworks. In this paragraph I will use the guidelines from Section 2 to analyze these frameworks. I won't consider the x-kernel here because its not object oriented and because the conduits architecture is very similar.

8.1 Limitations of Conduits

Though the framework takes care of the 'plumbing' (managing the communication between the protocols in a protocol stack and putting the stack together), very little support is offered for implementing individual protocols. There are so called hotspots available to add such functionality (e.g. subclassing the Session protocol to add context to protocols and subclassing State to create protocol States) but no ready made components are available.

The design focuses on providing a way to implement protocol stacks. Naturally, part of developing a protocol stack is implementing the specific protocols. Though there is an abstraction of what a protocol is (a FSM with state-transitions), there are only some very basic abstract classes and interfaces.

8.2 Composition-issues in Java Conduits

The Conduits framework has a long history. It started out as a C++ framework. Then it was ported to Java (with the addition of more features). At the moment it is still being developed at the University of Helsinki. Like many older frameworks, this framework has been developed as a monolithical framework [Mattsson 96a], so it can be difficult to reuse existing protocol software.

For instance, it is not obvious how a third party protocol can be inserted in the stack. A solution would be to wrap the legacy protocol in adapters. It is not clear though what consequences this has on performance. A problem could be that Java Conduits relies heavily on threads for session management. The inserted protocol would probably have to run in a separate thread, the messenger threads Java Conduit creates would stop at the adaptor to the embedded protocol and another thread would have to take over on the other end (this can be a big performance hit).

Another problem is that a lot of conversion has to take place. Converting data takes time. It is probably not possible to simply pass a pointer to the legacy protocol or vice versa. Attaching a protocol to one of the ends of the protocol stack is easier because of the Adapter Conduit that is specifically designed to allow third party software to communicate with the framework.

The framework makes extensive use of the 'Hollywood Principle'. All that has to be done is provide a set of subclasses of the 'hot-spots' in the framework. After that the conduit-graph has to be composed. The designers of the framework claim that a graphical tool could do the composing [Nikander].

This indicates that individual conduit-graphs are no likely candidates for reuse. An exception to this is the Protocol Conduit (which is a container for a graph in Java Conduits). This allows reuse of a part of a protocol-stack. More likely candidates for reuse are adapters for specific application or hardware interfaces and State and Messenger subclasses.

8.3 Evolution issues

The Java Conduits framework was designed to make evolution of protocols easier. This is done by separating protocol specific code from the rest of the framework in subclasses of State and Messenger. If the framework itself evolves, this should have not many consequences for individual protocols as long as the interface to State and Messenger remains the same.

The way the protocols are composed together in a conduit-graph is more likely to change. This may render existing graphs obsolete. A tool that automates the creation of the conduit graph can solve this problem.

Case studies would be necessary to reveal other problems. Students at the University of Helsinki have implemented a few protocols. These small projects revealed that it was still a lot of work to implement a protocol. I've found no reports on maintainability or evolution of these implementations.

8.4 Other quality attributes

Protocol implementations have to run on a wide variety of systems. The Java Conduits framework addresses this issue by being able to run on any platform that can run Java. This may exclude the smaller systems (at least at this moment) as well as systems where performance is very important (Java is slower than native code). Since I have not seen any reports on the performance of either Conduits+ or Java Conduits, the following analysis is based on knowledge of the Java environment and the analysis of the Conduits design.

Performance. As mentioned, Java is usually slower than native code, so this may be a problem for Java Conduits. On the other hand it is usually not the cpu horsepower that limits network traffic (one of the reasons why Java is popular on servers despite the performance gap). Scalable Java virtual machines for several server platforms already exist. Also the Java platform is still being optimized. JIT compilers and Suns dynamic compiler will probably increase performance further. On those platforms Java Conduits will probably have a good performance.

Java Conduits avoids one of the things that is known to have a big influence on performance of object oriented frameworks: object creation. The State objects that are used in a connection, are being reused. Messengers pass information through the graph, so there is no need to copy information once it is wrapped in a messenger.
Conduits+ on the other hand is normal C++ code. C++ is known to perform good so that is not an issue. Further more the Conduits+ design has the same advantages as Java Conduits when it comes to the overhead of object creation.

Memory. Java is infamous for its memory usage. Though the binary code of Java programs is much smaller than native code, the run-time environment can require a lot of memory (depending on how much objects need to be created).

Luckily Java Conduits is very conservative in creating objects (for performance reasons). Several parts of the framework can be implemented as Singleton objects[7] (Messengers, State objects).

Scalability. Java Conduits has good scalability (depending on the layout of the conduit graph) because of the use of threads. All Adaptors run in their own thread. If there are only a few Adaptors, there are only a few threads. The advantage of this is that few context-

7. The Singleton Pattern is explained in[Gamma]. It can be used to ensure a class is instantiated only once in a system. Effectively that single instance is shared by other objects.

switches are needed. The disadvantage is that if there are few Adaptors and many CPU's, processor usage might not be optimal. On smaller systems this is not an issue though.

8.5 Guidelines

In Section 2, I presented a list of guidelines that help developers to make better frameworks. Those guidelines can also be used to analyze frameworks. Though clearly not all of the guidelines are suitable for such an analysis (the ones about the development process for instance), some of them are useful.

8.5.1 Guidelines that were followed

Guideline 7 (loose coupling is good). Obviously since Java Conduits can be configured using a tool, this is a guideline that was followed. Java Conduit's components are all JavaBeans and can be put together using a tool. JavaBeans can be customized by setting properties and using events.

Guideline 8 (favor delegation over inheritance). Java Conduits offers both whitebox and blackbox behavior. Especially well in Conduits+ is that all the Conduits components are blackbox. The protocol specific behavior is delegated to other components. Though most of this design philosophy is intact in Java Conduits, the Session Conduit is not blackbox.

The reason for this is that the state machine contained in the Session object needs to store data somewhere. The designers of Java Conduits chose to use the Session object for this. In the next chapter I will show a different solution for this problem. To customize the type of data that can be stored in a Session object, subclasses of the Session class that contain the variables and access methods need to be created.

Guideline 11 (make configuration easy by providing a tool). Though the tool has not been implemented the developers of Java Conduit claim that it is possible to make one [Nikander]. Such a tool would make the protocol stack layout independent of the implementation. Developers can focus on implementing individual protocols instead.

Guideline 12 (use interfaces to abstract from components). Java Interfaces make it possible to separate API and implementation. By using interfaces as object types rather than classes, it is possible to plug in different classes (implementing the same interface). This was used in Java Conduits to provide a Conduit interface. Several other interfaces exist.

This gives developers greater flexibility than they would have had if they mixed interfaces and implementations because now they can choose to implement the interfaces in a different way if they don't like the default implementation, without affecting software that uses objects with that interface.

Guideline 13 (Use language features to protect framework design). Some classes were clearly not designed to be extended. Java provides the keyword final to prevent extension of certain classes. The conduit classes (except Session) are all final to prevent developers from implementing things in those classes instead of using the State classes and Messenger classes. Also the keywords private and protected are used to prevent developers from calling certain methods.

Guideline 15 (Limit the amount of platform specific code). Since Java is platform independent, the code is not tied to any platform at all. Implementations will have to interface with the hardware though (possible through the use of native methods). Of course one could say that Java is a platform on it self. But it's a very portable platform.

8.5.2 Guidelines that have not been followed

Guideline 5 (Specialize the framework rather than to generalize it). The goal of this guideline is to keep frameworks small and simple. Java Conduits addresses two things in the domain of communication protocols:

- Protocol Stack construction and management. The Framework makes it easy to put together different protocols in a stack. At run-time the framework takes care of data transport through the stack.
- Protocol Implementation. Protocols are assumed to be FSMs. That's why hotspots for implementing the State pattern are provided (An abstract State class).

The problem here is that while the protocol stack part of the framework clearly is an improvement over the old way of implementing a protocol stack (hardwiring relations between layers), the protocol implementations could be improved (see next chapter).

Guideline 6 Split if necessary. The consequence of applying Guideline 5 would have been to split the framework in two parts. One for constructing protocol stacks and one for implementing protocols.

Guideline 10 Provide access to the framework through as few classes as possible. The framework is quite complex to use. This could be overcome by providing utility classes and methods to compose stacks. Also it is not immediately clear which classes need to be extended to build a protocol. An excuse for this is that the framework was an alpha version when I downloaded it.

Guideline 14 Document the code. Probably for the same reason documentation was missing. The code had hardly been documented which made it quite hard to understand how everything worked.

Guideline 18 Don't make the control flow too complex. The downside of the 'Hollywood principle' is that the control flow is not entirely clear. This can be difficult for developers if they have to implement methods in such a way that they fit into the control flow. This is also true for Java Conduits. Even though I managed to find out what classes needed to be extended to implement a protocol, I did not manage to find out what each method was supposed to do. This can partly be prevented by giving examples and by documenting things well.

8.6 Conclusion about the Conduits Design

Conduits tries to solve two problems. Protocol stack configuration and management on one side and on the other side protocol implementation. Those are two independent domains and they should not be combined in one framework.

Java Conduits does a great job at implementing the first domain but has does not provide much functionality for the second domain. Separating the domains into two frameworks would allow multiple implementations of both.

One could imagine several versions of the protocol stack framework (optimized for different platforms for instance by applying different scheduling techniques). Also one could imagine having different frameworks for implementing protocols (for instance one for lower level protocols and one for higher level protocols).

9 Axis Network Protocol Framework

Axis Communications in Lund is involved in making devices for connecting things like printers and CD ROMs to a network (without using a PC). The system software needed for these devices is developed in Lund.

To ease the work of implementing the embedded software for their products they use a number of frameworks. One of these frameworks is the so called Network Protocol Framework that aims to ease implementation of protocols and protocolstacks. It was designed to work with their other frameworks. Yet it also implements some of the ideas in Conduits and the x-kernel.

Since Axis makes networked devices, the frameworks are widely used and any improvement in one of the frameworks immediately translates into better profits. One of the reasons they use proprietary software rather than buying it from a third party vendor is that their own implementations are more efficient and work better with their software.

9.1 Design

The framework is based on the OSI model for network protocols [Tanenbaum]. This model splits protocol stacks up in layers. The framework assumes that each layer takes a datagram from the layer above, adds a header and communicates it to the lower layer.

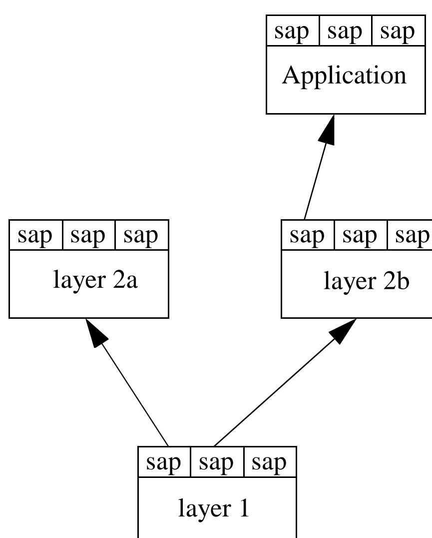

FIGURE 12. An example protocol stack.

Each layer is responsible for multiplexing data to the next layer. Each layer has a number of "Service Access Points" that other layers can plug into (also see Figure 12). The arrows between the layers are IO streams. These streams are used to transport the datagram.

Every layer supports up to five types of components (also see Figure 13):

- Protocol: a container for the other components
- Dispatcher: responsible for finding the correct SocketProcessor in the layer above
- Socket: connects to a SocketProcessor in the layer above (using the dispatcher) and talks to a socket processor in its own layer.
- SocketProcessor: processes the data flow.

- Router: helps finding the correct socket in the lower layer

Not all of these component types have to be present in a layer.

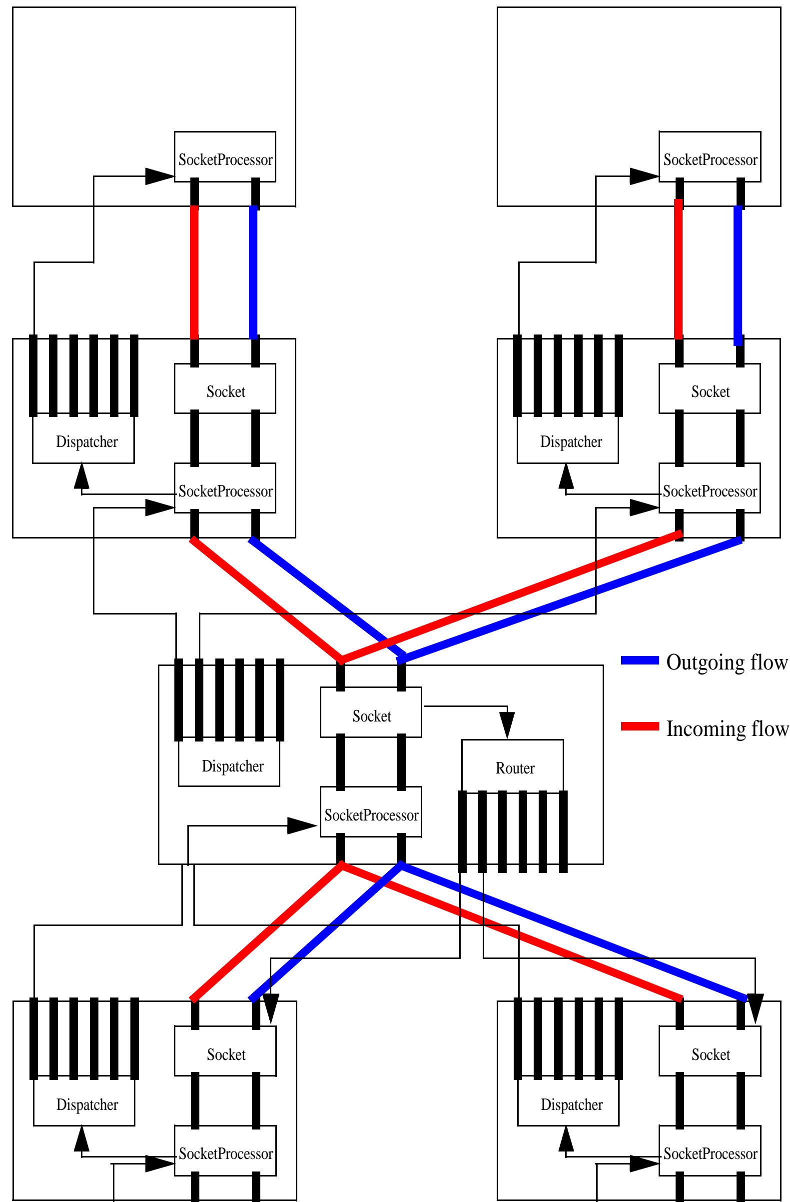

FIGURE 13. **Components in different layers.**

9.2 Conclusions

The impression I got is that Axis' Network Protocol Framework is the result of several evolution cycles on the original framework. It appears to be a clever design. It is optimized for both performance and memory usage. Though protocol stacks are configured manually, this is in principle something that could be automated. The reason that this is not done is probably that configuring things manually gives greater flexibility to developers.

An interesting point of view taken by Per Flock (the network protocol framework responsible, see Appendix A) is that the framework stimulates design reuse rather than code reuse. This point of view is very realistic since the amount of framework code in an appli-

cation is very low (3000:100000 LOC). Yet, the design is reused. This gives the code more structure and makes it easier to understand (once the framework specifics are understood).

Section 4 *Building Communication Protocol Frameworks*

In the previous chapter we examined a framework from the academic world called Conduits. In this chapter we will discuss the shortcomings of this framework when it comes to individual protocol implementations. Also we will look at a new way to implement FSMs.

10 Building Communication Protocol Frameworks

Conduits offers functionality in two areas:

- Protocol stack structure. Given a set of parts, it is be easy to build a stack. The framework should take care of the plumbing (i.e. delivery of data and events to the parts, routing the data through the stack).
- Finite state machines (FSMs). What happens inside a protocol component depends on the type of the protocol. Some protocols are simple and can be represented by for instance filters. More complex protocols can be represented by FSMs. Conduits assumes all protocols to be FSMs.

One of the guidelines I presented in Section 2 was that frameworks should be small and highly specialized (Guideline 5 and Guideline 6). When this is applied to the situation of communication protocols, we should conclude that there are actually two frameworks.

One framework to deal with the individual protocol implementations. Such a framework could should provide a nice mechanism to implement finite state machines. But other frameworks could be useful as well here (for instance a framework that models protocols as filters). And one framework to deal with the communication between the individual protocols.

Nevertheless, the two frameworks seem to be related in some way. The data that is pushed around through the protocolstack, is also used in the protocols. So there has to be some way to pass information between the two frameworks. Furthermore, one of the components that is assembled into a protocol stack is the protocol. So the framework has to be compatible with the protocol stack framework in order to fit protocols into the stack.

This means there has to be some interface between the two. By keeping this interface small (Guideline 10), unnecessary coupling can be avoided. This way both frameworks can evolve independently.

10.1 Building a protocol stack

The number of OO frameworks for communication protocols is fairly limited. Most frameworks seem to be related to either the x-kernel [Hutchinson] or conduits [Hüni]. Both approaches are quite similar and aim to make the composition of a protocol stack easier. In Java Conduits this is perfected to the point where a graphical tool could be used to do so.

This makes it easy to reuse individual protocols in a different context. A TCP/IP stack could be plugged into an ethernet adapter for a specific platform. If a new version of IP is introduced, it can be plugged into the existing stack (replacing the old version). In general those frameworks model a protocol stack in layers, as in the OSI model [Tanenbaum]. Each layer receives information form the layer above or below. Usually the information is passed in the form of frames or streams.

In the ISO-OSI model there are seven layers. The physical layer represents the actual physical connection (i.e. a copper wire or a glass fiber). This layer transports the bits from A to B. All the other layers rely on this layer for bit transport (the solid arrow in Figure 14).

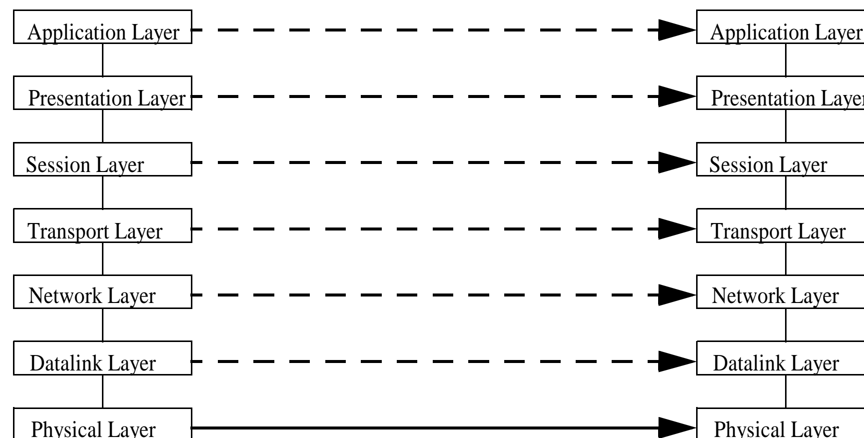

FIGURE 14. **OSI Model.**

Each layer is an abstraction for the lower layers. It's API hides all the details from the lower layer for the layer above. For layers above it appears as if the layer directly below is connected to a similar layer on the other host machine. These virtual connections are represented as dashed lines in Figure 14.

Procedural languages. Traditionally the layers are implemented as modules in a procedural language (like in the x-kernel). The procedural paradigm is very well suited for making layered software. The modules simply call functions in the lower layer module. This way information can be passes (through parameters and return values). Modules are ideal for implementing layering. If a module A needs to be put on top of module B, this can be implemented by letting module A's implementation use module B's API.

OO Approach. In a OO language there's no such thing as a module, so a different approach is needed. The layers could be modeled by objects with layer specific APIs of course but the OO paradigm offers more possibilities.

One of the things that could be done is to create a standard API for moving data between the layers. This could be done in the form of an interface or an abstract class. Instead of layer specific API calls, a standardized way of passing data could be provided. This has the advantage that it is easier to compose stacks of layers that were developed independently.

This is the approach that was followed in Conduits. Conduits improves on the concept of layers by using components instead. These components (the conduits) are connected together at run-time which makes the stack dynamically configurable, something that is difficult to achieve with a procedural language.

10.2 Building individual protocols

Though the conduits framework acts as a blackbox framework when the protocol stack is built, building the individual protocols still happens in a whitebox way. This means that classes have to be extended to implement a protocol. Once a protocol is implemented, it can be layered with other protocols in a blackbox style using the rest of the framework.

In Conduits it is assumed that the protocol is a finite state machine. So it uses the State pattern [Gamma] to provide implementation support in the framework. The conduits framework provides a state-base class and a mechanism to deliver messages to the state machine. Implementing this state machine requires the developers to extend both the state and messenger classes (also see the paragraph on Conduits).

From analyzing both academic and industrial frameworks (Axis), I found out that the larger part of the implementation effort goes into implementing the individual protocols. This suggests that the way in which existing frameworks support the development of individual protocols is insufficient and that more reuse could be achieved by improving support in this area.

Most protocols can be represented by Finite State Machines. Whether this is a good approach or not remains to be seen. In general there are two kinds of protocols [Tanenbaum]:

- connection oriented. These kind of protocols work as follows: first a connection is established, after that it is possible to communicate with the protocol, finally the connection is closed.
- conectionless. These kind of protocols do not require any connection to be set up. Instead each time there is communication, address information is sent along with the regular data.

Clearly the connection oriented protocols are more complex than the connection-less protocols. The reason for this is that connection oriented protocols need to store information about the connection between messages. So it makes sense to express these protocols as FSMs. For connection-less protocols this is not so obvious. Connection-less protocols don’t have to store information between messages.

Examples of connection oriented protocols are TCP [rfc793] (transmission control protocol) and FTP [rfc765]. TCP is used to establish connections to other computers. To do so it uses the connection-less protocol IP. FTP (file transfer protocol) is built on top of TCP. Though it relies for connections on TCP, it has a login mechanism to guarantee security. A user has to login to get access to the files on the FTP server. The server responds differently depending on whether the user has logged in successfully.

An example of a connection-less protocol is IP [rfc791]. IP is used to deliver datagrams to a specified internet address. Once the datagram has been sent away the protocol can forget all about it. On the receiving end, there is some administration work though. IP can split large datagrams up in smaller parts. On the receiving end they have to be put together again. This process can be expressed as a state-machine.

Another connectionless protocol is HTTP [rfc1945] (at least the 1.0 version). HTTP is in a way a very inefficient protocol. For each request to the server, a connection is opened. After the server sent back its data, the connection is closed again. For a complex web page this can mean opening and closing dozens of connection as for each image a separate request has to be sent. The advantage of this is that no information needs to be stored between requests. This way if a computer (either server or client) stops responding this has no consequences for the protocol. If a client stops responding the server will just continue running and provide its services to other clients. If a server stops responding, the client won’t receive any response to it’s request and can display some error message.

A distinction can also be made between low level protocols and high level protocols. Low level protocols typically process data packets with a header in some binary format. based on information in the header the protocol acts differently. Examples of such protocols are IP and TCP. In higher level protocols like HTTP and TELNET on the other hand, messages are typically defined in some abstract syntax.

So there are four groups of protocols:

- *Low Level Connection oriented*. Examples of such protocols are TCP and UDP.
- *Low Level Connectionless*. Examples of such protocols are IP and ATM.
- *High Level Connection oriented*. Examples of such protocols are Telnet and FTP.
- *High Level Connectionless*. An Example of such a protocol is HTTP.

Low level protocols differ from high level protocols in the way they receive messages. Messages are often encoded in a datapackage in low level protocols. To retrieve the message, a protocol first has to decode the data package. High level protocols often have a special syntax for doing so. HTTP for instance parses the input stream from the network as lines of strings.

Low level protocols can be hard to implement in a conduit like system (because there are hardly any explicit messages). To solve this problem they use filters that translate data headers into messages and vice versa. By inserting those filters on both sides of a low level conduit in the conduit graph, the low level conduit can be implemented as if it were a high level conduit (i.e. a FSM can be used).

Let's look at a few different approaches to implement FSMs.

10.2.1 Procedural languages

In procedural languages FSMs can be implemented using a double case statement. The outer case statement selects on the current state of the FSM. Then the inner-case statement selects the appropriate behavior for the current state given the type of message that was sent to the FSM (also see Figure 15). Of course it can also be done the other way around.

```
state A
    Message 1
    Message 2
    ...
    Message n
state B
    Message 1
    Message 2
    ...
    Message n
```

FIGURE 15. A nested case statement.

If the FSM is nested, the case statement nests even further. Each level of nesting causes the number of cases to be multiplied.

In this approach a lot of code is duplicated and reuse of code is nearly impossible. Maintenance is also very hard because adding a new type of message means implementing a case statement for all the states in the protocol. If on the other hand a state is added to the state machine, for each message the accompanying case statement (for the state) has to be edited.

In addition to this the duplicated code can make maintenance very tedious:

- bugs have to be fixed more than once
- it is easy to forget a piece of code
- the code becomes very complex.

All this makes the procedural paradigm a very unattractive way to implement a FSM. The main reason it is used anyway is that it gives the best performance and that in some cases the implementations are going to be used over long periods of time. This justifies investing the time and effort to build an implementation in a procedural language.

Also code size can be an issue. In embedded machines for instance, memory size and processor capacity are limited. In this kind of devices it might be more profitable to use a small and fast procedural implementation than a bigger and slower OO implementation.

10.2.2 The OO approach

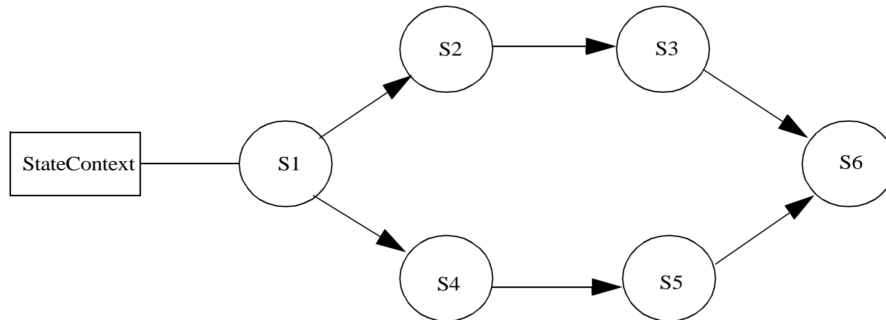

FIGURE 16. **The state-pattern.**

By using object orientation, the use of case-statements can be avoided. Usually some form of the State pattern is used to model a finite state machine (FSM) [Gamma]. Each state is represented as a separate class. All those state-classes inherit from a State class. Figure 15 shows this situation. The round shapes represent the different states in the FSM, The arrows represent possible state-transitions between those states.

The StateContext delegates its behavior to the state objects rather than selecting it with a case statement. The individual state objects contain methods that define this behavior. The result of the execution of such a method may be a state change. The next time the StateContext delegates its behavior to the new active state.

Gamma motivates using the State Pattern as follows: “An object ‘s behavior depends on its state, and it must change its behavior at run-time depending on that state” or “Operations have large, multipart conditional statements that depend on the object’s state” [Gamma].

In other words, each time case statements are used in a procedural language, the state pattern can be used to solve the same problem in a OO language. Each case becomes a state class and the correct case is selected by looking at the current state.

The StateContext is the object that ‘owns’ the FSM. It simply holds a reference to the current state. It also functions as an access-point for common data, hence the name FSM context. State transitions are implemented as methods in the state objects.

10.3 The state pattern does not solve everything

Though the statepattern is certainly an improvement over the case statement approach it has a few shortcomings when it comes to implementing Finite State Machines (FSMs). FSM-diagrams can be used to model large and complex systems (reactive systems as they are called in [Harel 86]). Protocols are a good example of such systems. By using a graphical notation it becomes easier to understand the design of the system.

Finite state machines are described as a set of:

- States.
- Input events. This could for instance be the arrival of some data at the FSM or it could be a Timer that runs out. It could also be the entry into a new state (after a statechange) or the exit of a state (before a state change).

- Output events. This is some action that takes place after an input event arrives at a state-machine. It could be part of a transition (see below) or it could be part of a State (for instance a state-exit-output event triggered by a state-exit input event.
- Transitions. A transition has a source state and a target state. Transitions are triggered by an input-event. The triggering may also cause the launching of one or more output-events

Of all these concepts the only thing represented in the state pattern are the states and the output events. Worse they are represented in a single class.

Output events are represented as methods in state classes. By doing so those output events are hard to reuse in other states. By putting states in a state class hierarchy, it is possible to let related states share output events by putting them in a common superclass. But this way, output events are still tied to the state machine. It is not possible to use the output events in a second FSM (with different states).

The other FSM elements (input events, transitions) are represented implicitly. Input events are simulated by letting the FSM context call methods in the current state object. Transitions are done by letting those method change the state after they are finished.

The disadvantage of not having explicit representations is that it makes translation between (conceptual) design and actual implementation more difficult. Also when the design changes it is more difficult to synchronize the implementation with the design.

Advanced state machines. David Harel extended the idea of FSMs (not supported by the state pattern of course) into something what he called statecharts [Harel 86]. Basically State-charts is State Diagrams + nesting + orthogonality + broadcast communications. Lets take a look at these new concepts

- Nesting. State Diagrams can become very complex. To reduce complexity it is possible to encapsulate sub FSMs in a state.
- Orthogonal states. State Diagrams model a system in a sequential style. In multi-threading systems a system can be in multiple states. This can be modeled using orthogonality. Orthogonality is an AND relation between multiple states.
- Broadcast communications. This allows one event to cause transitions in more than one state. This can be very handy in combination with orthogonality.

Nested frameworks can be an indication of layers within a protocol. The developers of Java Conduits saw this and changed their framework accordingly [Nikander]. Instead of viewing a protocol conduit as a single layer in the OSI model, a protocol is a sub-graph of conduits (also see paragraph 6.2). In other words they have more fine grained layering.

This allows for the individual layers to be simpler and since putting the stack together can be automated to a large extent already, it also eases the implementation. We will look at this type of FSMs in more detail later on in this Chapter.

FSM instantiation. In the TCP protocol, up to approximately 30000 connections can exist on one system. Each of these connections has its own FSM. Of course it would be inefficient to just clone the entire state machine (all the state objects) each time a connection is opened. The number of objects would explode. Suppose the TCP FSM has 25 states, with 30.000 active connections the system would have 30.000 X 25 = 615.000 objects. The memory usage of such a system would be very large.

To contrast the memory usage: a system implemented in a procedural language would implement this as a module with a case statement. This module would be loaded into memory only once and all connections would share this implementation.

Also it would not perform very well because object creation is an expensive operation. In the TCP example, creating a connection would require the creation of 25 objects, each with their own constructor.

So a mechanism is needed to use FSM's without duplicating all the state objects. The State pattern does not support this. In conduits this is solved by combining the State pattern with the Singleton pattern [Gamma]. State objects are made singleton to make sure they are instantiated only once.

The Singleton Pattern ensures that classes are only instantiated once in a system. The methods in the state objects store their variables in the FSM context instead of in their own class. One advantage of this is that these variables are still accessible after a state change.

I will use the term FSM-instantiation for the process of creating a context for a FSM. Then a context can also be called a FSM instance. Multiple instances of a FSM can exist in a system.

Global variables. Another problem is data storage. The actions in the transitions of a statemachine perform operations on data in the system. Storing data in a central place in general is not a good idea in OO programming. Yet it is the only way to make sure all operations in the FSM have access to the same data. That's why a context object is used to store data in the State-pattern.

The problem with this context object is that to make a FSM, one must also create a context class (containing all the variables and data used in the FSM). The actions in the FSM are thus tied to this context subclass because without the context they can't access their data. The context acts as a container for global variables.

This makes those actions very hard to reuse. This explains partly why implementing protocols is so much work. A lot of actions could be reused if they would not use the context.

10.3.1 Delegation based languages could help

A problem in the state pattern is that state transitions do things (lets call these 'things' operations) that could happen in another context (for instance a different FSM or even a different application) but those things are tied to the FSM because of the way they are implemented. The actions are implemented as methods in State objects. Those state objects have little or no meaning outside the FSM.

An simple example of such an action is setting a timer. This is something that is done in many protocols. Assuming the timer is a normal object, the context will probably have a reference to a timer variable. A setTimer action would access this variable (hardwiring the call to the variable) to start the timer.

The transition-actions change things in the context and are generally implemented in a State object. This makes operations reusable only within the FSM. Other states within the FSM can inherit from a state that supports a set of operations. It's usually not feasible to inherit from states in a different state machine so the best method for reusing this code is 'copy and paste'.

The main reason it is organized this way is that operations need access to the context (to store, change and update variables). This means that the access to the context is centralized in one object. If the FSM would have been implemented in a delegation based language[1] (for instance

1. This is also recognized in [Zweig].

Self [Ungar & Smith]) instead of a inheritance based language, it would be possible to decentralize the context.

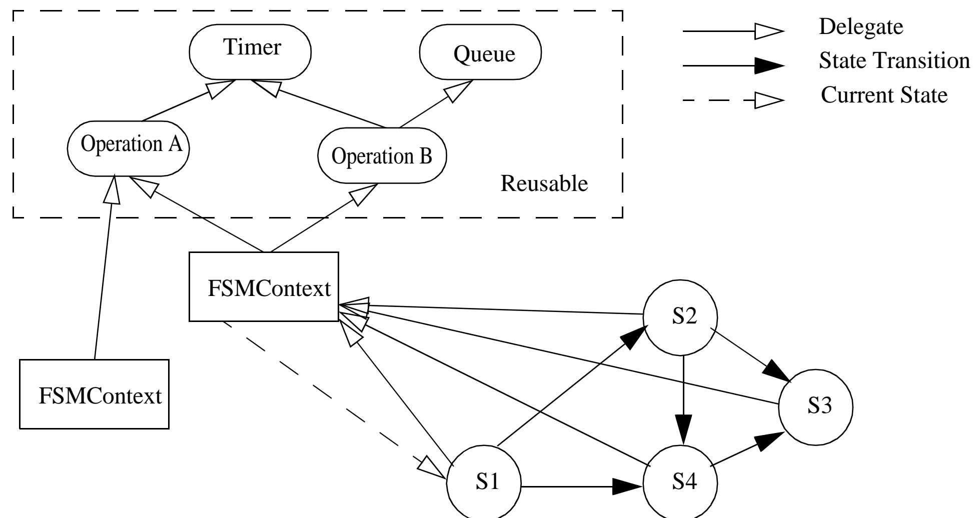

FIGURE 17. Delegated state pattern.

In Figure 17 is demonstrated how a delegated model would solve this. Instead of defining the operations in the state objects (which would automatically make them FSM specific), the state objects delegate everything to their context. The context delegates it further to reusable objects. If a certain object could be used in another FSM, it can just be plugged into the accompanying context.

Notice that with delegation both state (object-state, not FSM state) and behavior are ‘inherited’. It is of course not class inheritance but only object inheritance. At first this seems bad because it forces is to duplicate all the command objects for every FSM instantiation. But there is a solution. Delegated languages such as Self support incomplete objects. These objects can be compared to abstract classes. The incomplete object uses methods and/or variables that need to be provided by objects that delegate to it. Using this mechanism, Command objects can move their data to the context object.

However, using a delegation based language still doesn’t solve the problem of FSM elements not being represented explicitly in the system. Worse the number of dependencies between the objects has increased.

11 Enhancing the State pattern

Delegation only partly solves the problems mentioned in the previous paragraphs. By enhancing the state pattern several of the problems can be solved. The main problems mentioned there were:

- The existing state pattern does not provide explicit representations for all the FSM concepts. This makes maintenance hard because it is not obvious how to apply a design change to the implementation.
- The state pattern is not blackbox. Building a protocol requires the developer to extend classes rather than to configure them.
- Implementations are complex.
- The state pattern prevents reuse of behavior by tying behavior to states.

Representing FSM concepts as objects would cause the number of object instantiations to rise during run-time. Often this is not desirable, so a better solution is necessary. As we saw, the state pattern puts actions into state objects. This gives them access to the state's context and allows them to store/edit values in this context. This way the only thing that needs to be created when the FSM is instantiated, is the context. Once it is created the initial state is set in the context and the rest of the objects are shared with other contexts. So State objects can only contain behavior but no data.

I would like to propose a different approach to modeling FSM's in OO languages.

11.1 The new pattern[2]

In the state pattern, actions are simply methods in some class. This ties them to that class and the only way to reuse them is to extend that class. So where else could we put these methods? We could put them in the context, but then we would get one huge context (which was the reason they were put in separate state objects). We could also put them in separate objects (following the command pattern). These objects could act as Components that can be associated with a transition.

Then what exactly is a transition. A transition is a state change triggered by some event. As a side effect an action can be performed. A transition has a source state, a target state, an event (often called the trigger) and a action that is executed when the transition is triggered.

The context distributes all events to the current state. This state acts as an event dispatcher and triggers the transaction that is registered for the incoming event.

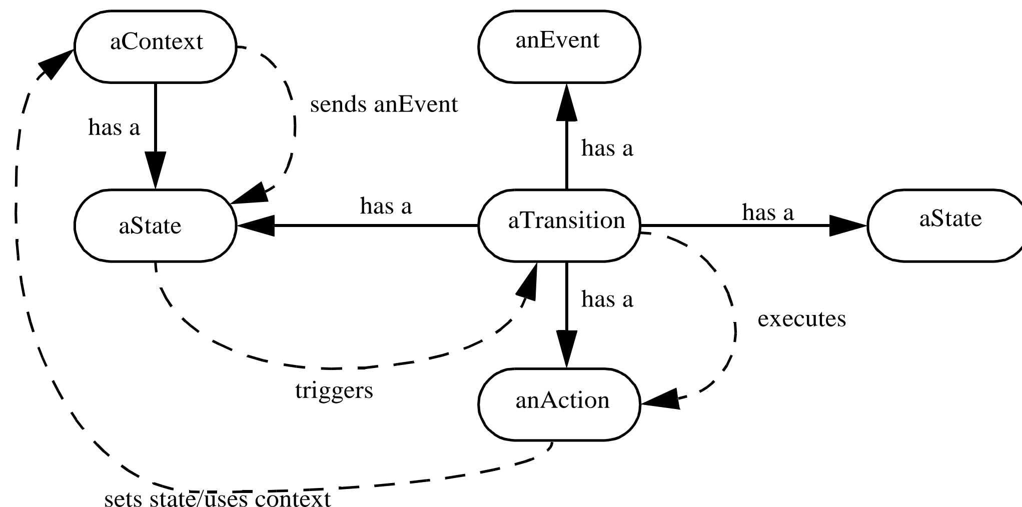

FIGURE 18. The enhanced state-pattern.

In Figure 18 this idea is represented in a small model. In this model all the states, events and commands can be singleton since they carry data nor behavior. The data is stored in the context. Commands are executed in a context. Events are dispatched by states with a reference to a context.

Whenever a event is dispatched from the context to the current state, a reference to the context is passed along. This way transitions and actions can access the data in the context during execution. Also this makes it possible to have multiple contexts (multiple FSM instances).

2. I use the term pattern to contrast my design with the existing State pattern.

A possible scenario. A timer in the context times out. This event is given to the current state's dispatch method together with reference to the context the timer was running in. The state finds a transition that has the same event as a trigger and executes it. The transition executes its associated action and after that sets the new state in the context.

Conditional Transitions. There is one thing that is not yet modeled by this approach: conditional transitions. Conditional transitions are used to specify transitions that only occur if the trigger event occurs and the condition holds true. Though this clearly is a powerful concept, it is hard to express it in a OO way. A possibility could be to use the Command pattern again to create condition objects with a boolean method but that would tie the conditions to the implementation (meaning that if the implementation changes, the conditions also have to be re-implemented even though the design did not change). Another possibility is to have more events and simply check whether conditions hold true before launching the event. The implementation in paragraph 11.2 follows the last approach, mainly to keep things simple.

Benefits. This pattern solves the problem of actions not being reusable. Also it provides more explicit representations of FSM concepts. Now there are state objects, event objects, transition objects and action objects. And in addition to that, most of these objects can be configured in a blackbox way. Since the state objects no longer contain FSM specific behavior, there is no need to create subclasses of a State super class for each state. Instead we can use a single state class that has a property that specifies the name of the class. All that needs to be done to create a new state is:

- Create a new instance of the state class
- Set the name property to the name of the new state.

In the same way, events can be created. Also transitions can be created in a blackbox style by setting source and target states, an action and a trigger event. The only object that is FSM specific is the action that is executed during a transition. Actions have to be created in a white box way.

Actions store and edit variables in the context. This creates a dependency between the context and the action. It might be useful to turn the context into some sort of repository where variables can be added and removed by actions instead of just making context subclasses with all the necessary variables.

This would greatly simplify the context because all FSMs could use the same context class. The responsibility for making sure variables are available would shift to actions. Initialization can be handled by a special initialization action that initializes the context when a new context is created for the FSM (the instantiation of a FSM).

11.2 An implementation of the enhanced State Pattern

I implemented[3] this pattern as a small framework in Java. In this paragraph I present some details of this implementation. While implementing I tried to follow the guidelines presented in Section 2 in order to make the framework composable (with for instance conduits) and to allow for future extensions.

11.2.1 Classes

The core framework consists of only six classes and interfaces. In addition to that I created a few helper classes. In Figure 19 a collaboration is given of the frameworks core classes. The

3. A zipfile with all the source code and some demos can be found at `http://www.student.hk-r.se/~jvg/MasterThesisHomepage/`

source code for the core framework is given in Appendix B. One class (FSM) is missing in this diagram, in the explanations below is explained what this class is used for.

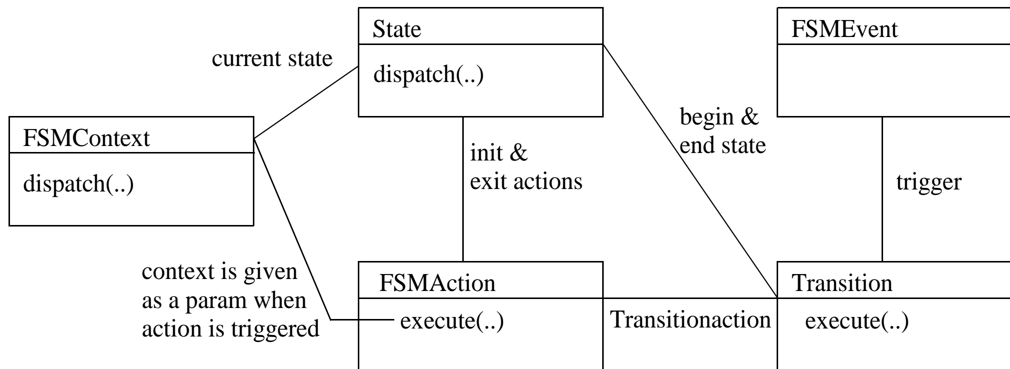

FIGURE 19. Enhanced State Pattern Collaboration diagram.

State. This class models a state. A state has a name and is associated with an state-entry action and a state exit-action. It also maintains outgoing arrows (transitions). Transitions can be added by calling the method addTransition(..). This method takes as parameters the target state, an event and a FSMAction.

By putting the creation of transitions in the state object, the state object knows which transitions there are and can also host a dispatching mechanism. In my implementation, events are mapped to transitions using a Hashtable (java.util.Hashtable). When the dispatch method is called (somestate.dispatch(FSMEvent trigger, Object data, FSMContext fsmc)), the correct transition is looked up in the hashtable and then executed. The implementation is not very fault tolerant at this moment (mainly due to lack of time to make a proper implementation) so incorrect input (like events that cannot be handled) result in exceptions.

FSMContext. The context holds a reference to the current state. Incoming events are dispatched to this state. The context also functions as a repository for objects. This way the FSMActions can use it to store and retrieve FSM specific data (timers, queues, etc.). In my implementation the context simply extends from java.util.hashtable. This is a reasonably fast dictionary implementation. It can be used in the following way:

```
...
Timer t = new Timer(20).start();
aFSMContext.put("timer no1", t);
...
```

This way the timer will survive the state change and another action can later do this to retrieve a reference to the timer:

```
...
Timer t = (Timer)aFSMContext.get("timer no1");
t.stop();
...
```

FSMAction. This is an interface with the following method declaration:

```
public void execute(FSMContext fsmc, Object data);
```

It can be used to implement some specific action. To be able to store information globally (for the whole state machine instantiation), the context is given as a parameter. This way the FSMAction can be singleton and still behave in a FSM instance specific way.

The data parameter allows the FSMContext to pass data to the action. An example of such data could be a data-packet traveling through a protocol stack. This creates a dependency between the actions and the protocol stack framework because both need to know what the format of the

data being passed is. The FSMContext is not affected because it just passes a reference to an Object.

FSMEvent. The event class is very simple in my implementation. It contains a string with the name of the event. It is completely passive and is just used to distinguish between the different events. It might be useful to replace this class by ordinary strings since this removes some complexity. On the other hand that would tie the design more to Java.

Transition. The Transition class offers only one method:

```
public void execute(FSMContext fsmc, Object data);
```

When it is called, it simply passes the parameters on to it's associated FSMAction. When a Transition object is created, it gets a reference to the source and target state object and an Object that implements FSMAction.

FSM. This class contains the entire FSM and offers a method to create new contexts for this FSM. The entire FSM can be created by calling the methods of this class. This way it is possible to radically change the implementation of the FSM without breaking existing applications.

This class also offers methods called addState(..), addEvent(..) and addTransition(..) as an easy way to configure the FSM. By using this class, developers can avoid creating State objects, Event objects and Transition objects directly.

11.2.2 Event trace

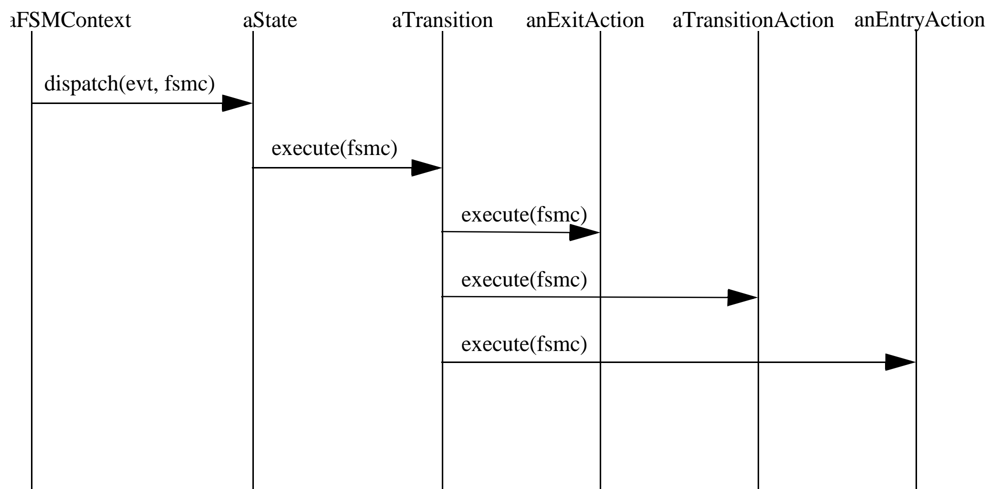

FIGURE 20. Event trace for the FSM Framework.

In Figure 20 an event-trace is shown that represents the triggering of a transition. First the FSMContext receives an event (not in the diagram). Then this event is given to the state together with a reference to the context by a call to the dispatch method in state. Using the event the state object finds the correct transition and executes it. The Transition first launches the current state's exit action, then the transitions action is executed. After that the target state is set as the current state in the context. Then the target state's entry action is executed.

11.2.3 The helper classes

To test the framework, I created a class FSMController which provides a simple user-interface to the FSM. It can be used to check whether the correct transitions are triggered by events. It shows the current states and all the events for the outgoing arrows in that state. Selecting one and pressing the “dispatch event” button causes the event to be dispatched (also see Figure 21).

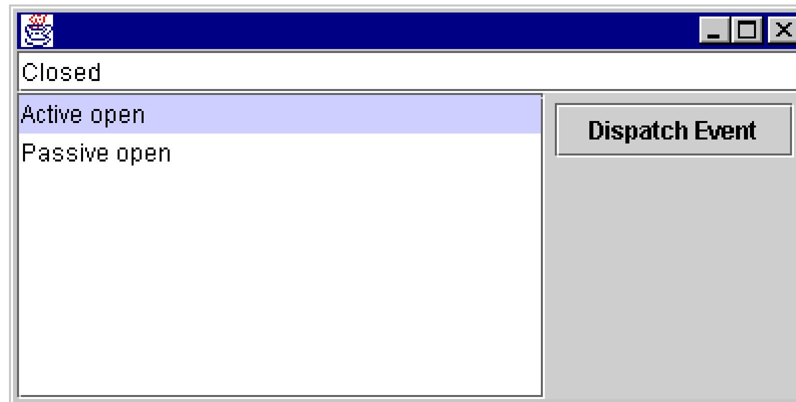

FIGURE 21. **The FSM Controller.**

Other helper classes I created are

- *SkipAction*. This is an implementation of the FSMAction interface (Appendix B.2). The execute method has an empty body. This class can be used as a dummy action.
- *WriteLogAction*. This is also an implementation of the FSMAction interface. It prints a string on standard out when the execute() method is called. Like SkipAction this class is ideal for debugging.
- *FSMActionList*. This is a convenience class that makes it possible to put multiple FSMAction objects into one container. The FSMActionList itself is of course also an FSMAction and when the execute() method is called, all FSMActions in the container are executed in the order they were put into the list.
  FSMActionList is a nice addition to the Command pattern. It allows developers to make composed commands. Several variations might be useful. For instance, threaded commands where the execute method triggers the thread’s run method which in it’s turn uses the execute method of a normal command.
  This type of behavior extensions is also known as the Decorator pattern [Gamma].
- *FSMGenerator*. This class generates a FSM object from a XML file. More about this class can be found in paragraph 11.3.
- *Serializer*. This class serializes any object (that implements java.io.Serializable) to a file. It can also deserialize objects from a file. It can be used to serialize FSMActions. This class is used by the FSMGenerator.

11.3 FSMGenerator

The Conduits framework allows stack configuration by using a tool. The reason this is possible is that large parts of the framework are blackbox. In the implementation of the FSM state pattern, most of the classes are blackbox too. So it should be possible to create a graphical tool to create FSM’s.

Instead of building a graphical tool (which would be time-consuming), I decided to make a language from which FSMs can be generated. Such a language could also be used as a fileformat if a graphical tool was ever made.

Instead of just making some BNF grammar for my language, I decided to use XML [xml]. The advantage of XML is that parsers are widely available (at least for Java) and that it appears to become a standard language for structured data.

11.3.1 XML

XML stands for eXtensible Markup Language. It is very similar to HTML. Just like HTML it is an application of SGML (Standard Generalized Markup Language). In fact XML offers a subset of the features of SGML. XML can be used for many purposes. The basic construct of XML is a tag. A tag has the following form:

```
<tagname attribute1="value" attribute2= ..> text </tagname>
```

Unlike HTML which comes with a predefined set of tags, XML allows developers to define their own tags. This makes XML a very powerful language. In general XML can be used for most types of structured data.

Part of the XML specification is the so called DTD (Document Type Definition). A DTD can be used by developers to specify requirements for a XML document. A parser can use the DTD to check whether an XML file contains the correct tags and whether those tags are in the right places.

11.3.2 FSMs in XML

Below is an example of an XML file that can be used to create a FSM. In this file three states, two events and three transactions are defined.

```
<?xml version="1.0"?>
<fsm firststate="A" initaction="Helloworld.ser">
        <states>
                <state name="A" initaction="enter.ser" exitaction="leave.ser"/>
                <state name="B" initaction="enter.ser" exitaction="leave.ser"/>
                <state name="C" initaction="enter.ser"
                                exitaction="leave.ser"/>
        </states>
        <events>
                <event name="E1"/>
                <event name="E2"/>
        </events>
        <transitions>
                <transition sourcestate="A"
                            targetstate="B"
                            event="E1"
                            action="Helloworld.ser"/>
                <transition sourcestate="B"
                            targetstate="C"
                            event="E2"
                            action="Helloworld.ser"/>
                <transition sourcestate="C"
                            targetstate="A"
                            event="E1"
                            action="Helloworld.ser"/>
        </transitions>
</fsm>
```

In this file several tags are defined. The structure of the file is very simple and is just intended as an example to show that it is possible to create an FSM using such a simple language. If this approach would be used in real life it would probably be necessary to create a proper DTD (document type definition) that defines exactly how the XML code should be structured.

The FSM that is specified here looks like this:

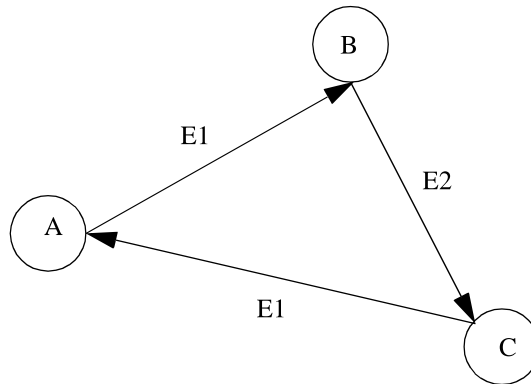

FIGURE 22. An example FSM.

The only thing that's not defined in the file is the FSMActions. These can't be defined here because they are not black box. Instead references to .ser files are given. These files are serialized Java objects. Java allows objects to write themselves to a binary stream. This is called serialization. Objects can also be read from a binary stream (called deserialization).

To build a .ser file, FSMAction classes must be created, instantiated and configured and then written to a file (using serialization). This way the FSMActions can be used in a blackbox way. One disadvantage of this approach is that it can only be done with languages like Java that are dynamic enough to allow things like serialization.

11.3.3 The Generator

The generator uses IBM's xml parser [xml4j]. This parser, called xml4j, is written in Java and can be used freely for non commercial projects. It parses a document like the example in paragraph 11.3.2. When the document is parsed, the parse tree can be accessed using the Document Object Model API which is standardized by the World Wide Web Consortium [w3c].

Using the DOM API the generator first retrieves all the state tags and uses the FSM class to create a state for each of them. Then it proceeds with the event tags and after that the transition tags. The result is a FSM object which contains the FSM as specified in the XML document. The parser also has the ability to process DTD's so if a DTD was developed for XML code that specifies a FSM, the parser would automatically check whether an input file is correct.

The DOM API could also be used to create XML. This could be useful if a graphical tool were developed. Through the DOM API such an application could create XML files or call the generator directly.

11.4 What does the pattern solve

The existing state pattern does not provide explicit representations for all the FSM concepts. Programs that use it are complex and it cannot be used in a blackbox way. This makes maintenance hard because it is not obvious how to apply a design change to the implementation.

The enhanced State pattern however provides abstractions for all of the concepts. There is a State class, an Event class (input events), a Transition class, a FSMAction class (output events). Furthermore, each State has state-entry and state-exit actions.

In addition to that it's also possible to define an action for FSM instantiation. FSM instantiation is a concept that is not part of the traditional state pattern. In the context of multi-threading OO systems it is a very useful concept though. One FSM can have multiple instances (contexts).

Each time a context is created, the init action is executed (useful for initializing variables that are going to be used during FSM execution).

The FSM must be able to support multiple instances. The state pattern (with a small extension) already solves this (by making state objects singleton). The enhanced State pattern uses a similar solution. The classes are no longer explicitly singleton though. The reason for this is that it might be useful to have multiple FSM objects from the same XML file in one system. On a multi processor machine for instance this might simplify memory management because each processor can have it's own dedicated set of FSM objects.

The original state pattern is not blackbox. Building a protocol requires the developer to extend classes rather than to configure them. The enhanced State pattern can be configured all the way. The only things that need to be implemented are the FSMActions. Configuring the FSM can be done from an XML file using the FSMGenerator class.

Advantages over the original State pattern:

- States are no longer created by inheritance but by configuration. The same goes for events. Also the context can be represented by a single component. Inheritance is only applied where it is useful: extending behavior. Related Commands can share behavior through inheritance. Also Commands can delegate to other commands (removing the need for events supporting more than one command).
- States, actions, events and transitions now have explicit representations. This makes the mapping between a FSM design and implementation easier. A tool could create all the event, state and context objects by simply configuring them. All that would be required from the user would be implementing the commands.

Disadvantages compared to the original State pattern:

- The context repository object possibly delivers worse performance than directly accessing variables. Instead of directly accessing variables, the variable will have to be obtained from a repository. The Hashtable implementation I chose is pretty efficient though.
  On a machine where processor performance is the bottleneck this might not be a good thing. With increasing processor speed I don't think this will be a big problem. I haven't done any measurements to support this claim though.
- It could be difficult to keep track of what's going on in the context. The context is simply a large repository of objects. All actions in the FSM read and write to those objects (and possibly add new ones).

There is no support for conditional transitions. With conditional transitions, a transition is only triggered if the event occurs and the condition holds true. Conditional transitions are a part of the finite state machine concept. If conditions are needed, they will have to be implemented in the FSMActions where the events are launched. Instead of evaluating the condition when a event is received it is checked whether the event should be launched or not. In most situations this should be good enough.

11.5 Composition & Evolution

I tried to keep composition and evolution problems in mind while developing this framework for FSMs. So I tried to design the framework using the guidelines in paragraph 3.

11.5.1 Specialized framework

One of my most important guidelines in paragraph 3 was to keep frameworks small and specialized. The FSM framework meets this requirement. The size of the framework is only six classes. Because the framework is so small, it is easy to understand. Also, because the number of LOC is limited, maintenance should not be a problem.

Also I tried to keep the interface small and localized by using encapsulation. Originally I had only five classes. Creating a framework meant creating instances of both State and Event classes. After that transitions had to be created by calling a method in the State class. I simplified this process by creating a new class called FSM, which provides a interface to do these things. In addition to that it also provides a factory method to create a new context. So normally the only class that a developer needs to work with is the FSM class (and of course the FSMAction interface).

After creating the six core classes, I found that there were several things that would be convenient to have but were not part of the core framework. An example of such a thing is the skip command which implements an empty execute method. Another example is the small GUI I created to test FSMs. All these convenience classes depend on the framework but are not part of it. To stress this I put them in a different package.

This way it is clear that they are not part of the framework but that their use is recommended anyway. Technically the convenience classes are small framework instances. When the framework changes they will need maintenance.

A typical interaction with the FSM class could look like this:

```
FSM fsm = new FSM();
fsm.addState("Closed");
fsm.addState("Listen");
...
fsm.addEvent("Passive open");
fsm.addEvent("Active open");
fsm.addEvent("Close");
...

fsm.addTransition("Closed", "Passive open", "Listen",
                  new WriteLogAction("create TCB"));
fsm.addTransition("Listen", "Close", "Closed",
                  new WriteLogAction("delete TCB"));
fsm.addTransition("Closed", "Active open", "SYN Sent",
                  new WriteLogAction("create TCB\nsend SYN"));
...
fsm.setFirstState("Closed");
```

In this small code example a fragment of the code that creates a FSM for the TCP connection procedure is shown. As this FSM is rather big, I removed most of the method calls in this example.

11.5.2 Loose coupling

Another thing I tried to accomplish is loose coupling. I wanted to avoid that it was necessary to extend classes from my framework as much as possible. Furthermore I wanted to be able to treat states, events and transitions as if they were components (if I can do it an application can do so to). So I implemented State as a single class that upon instantiation is given a name in the form of a String. In the same way I created an Event class.

Then I had to find a way to invoke specific operations from the transitions. In the State pattern those operations are simply implemented in the State objects. This would however require developers to create subclasses of State. So I decide to use the command pattern [Gamma]. In the command pattern, commands are put in a command object. This object implements the command interface (usually an execute(..) method) that must be called in order to invoke the command.

I created an FSMAction interface that provides one method:

```
public void execute(FSMContext fsmc, Object data);
```

The method has to parameters. The first is a reference to the context in which the command is executed. This way one instance of a command can be shared by multiple contexts. The second is just a way of passing additional data. In communication protocols this parameter can be used to pass the current frame to the command.

11.5.3 Dealing with changes

One of the problems with trying to fix the API for a framework is that after some time changes may be needed to this API. An example of this is the framework controller class I created. This class needs to know which events can be sent to the FSM in a certain state. I implemented this by giving the State class a giveEvents() method (which is a API change).

Another example of a API change I had to do is the addition of Object as a parameter to the execute method in the FSMAction class. At some point I realized that the FSMContext alone did not contain enough information for a transition to complete. Also a reference to some data that accompanied the event was needed. An example of such data is a network packet that arrives at a protocol.

This change was a bit more complicated because it required several changes in other classes. It would have been a lot of work if the framework would have been bigger.

12 Evolution of the FSM Framework

In paragraph 10.3, I briefly mentioned statecharts, David Harels extension of state diagrams. Statecharts are useful because they can model parallelism by using orthogonality and broadcasted events. Also the complexity of the diagrams can be reduced by using nesting.

I will show how these concepts can be implemented in the FSM framework I presented earlier. I will use these extensions to the framework later on to evaluate the effecitiveness of my guidelines in the evolution of my framework.

12.1 Nested State Machines

One of the things Harel introduced was hierarchical state machines. This means that any state can contain a sub-FSM that processes the incoming events. In Figure 23 this idea is illustrated. The example was borrowed from [Harel 88].

The example shows two FSMs. The two FSMs are equivalent. The right one is a nested one while the left one is a regular one. The nested FSM has an extra state D which contains the states A and C. Both A and C have a transition to B triggered by event f. In the nested FSM these transitions are replaced by one transition from D to B.

Another change is the transition from B to C triggered by event h. In the nested version of the FSM this transition is connected to D instead of C. D just sends it on to it’s default state C (marked with a bulleted arrow).

The nesting reduces the complexity and makes the diagram much easier to read. This is especially true for large examples (for a large example see figure 22 in [Harel 88] on page 525).

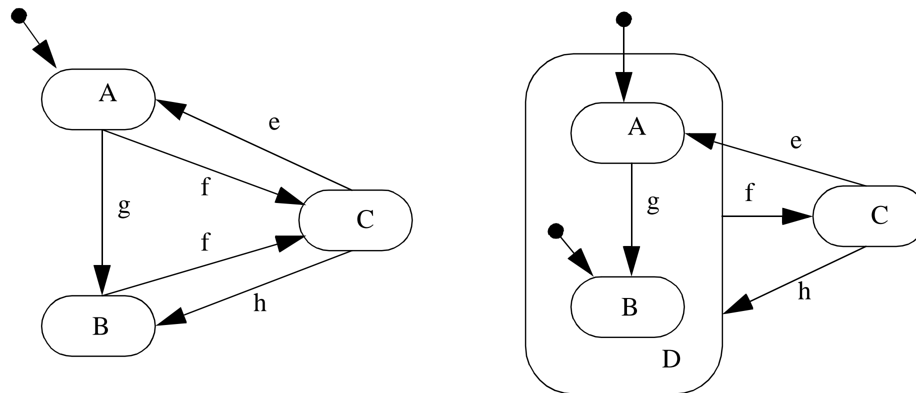

FIGURE 23. Nested FSM.

12.1.1 Adapting the FSM Framework

Harel's notion of hierarchy can be implemented in the FSM framework I created by making some small adjustments. The requirements for such a framework are as follows:

- There are two types of state objects: one with a nested FSM and normal state objects.
- normal state objects work exactly the same as in the old framework.
- nested state objects have special dispatching behavior.
- any state can be connected to any other state with a transaction (regardless of nesting)

Since non-nested states are exactly the same as in the old framework, we'll focus on the nested state objects. So a nested state has a reference to the current state of it's embedded FSM. This FSM operates in the same context however.

I have considered using a nested context. This is not possible because I would have to deal with transitions between states in different contexts. For this reason it is more convenient to have a single, global context. A nested context would be nice however to support distributed FSMs (for instance to model a workflow application).

If we take a look at Figure 23, we see several types of transitions.

- There are internal transitions which do not go outside the nested FSM. An example of such a transition is the one between A and B. This transaction is triggered by event g. A possible scenario for triggering this transaction: the current state of the entire FSM is D, event g is sent to D which forwards it to its internal FSM which happens to be in state A. State A receives the event and triggers the transaction.
- Then there are transitions from D to other states. An example of this is the transition from D to C. The transition does not really originate in D but can start from any state in the embedded machine of D (or in one of the states in the embedded states of an embedded state etc.
  The arrow from D to C is just an easy way of drawing this. Note that such a transition 'leaves' the embedded state machine. This means that the embedded state machine goes back to the default state. So if the transition from D to C was caused by state A, the embedded machine of D is now in state B (the default state) while the parent FSM changes state from D to C.
- There are transitions from any state to D. An example of this is the transition from C to D triggered by h. When this happens h, the embedded FSM changes to its default state.
- There are transitions from any state to states within the embedded state machine. An example of this is the transition from B to A. Apart from setting the current state of the embedded FSM to A it also sets D as the current state for the parent FSM.

With these requirements I adapted the FSM framework. Nearly all the core classes were affected in some way. The most significant change was the addition of two new classes:

- NodeState
- LeafState

These two classes replace the single State class in the old framework. Both of them inherit from the now abstract State class. A NodeState can be used to embed a FSM. A LeafState corresponds to a normal state. Both state types have different dispatching behavior and different Transition adding behavior.

With the two state types in mind, the four transition types can now be generalized:

- Transitions from LeafStates to LeafStates
- Transitions from NodeStates to any type of State
- Transitions from any type of state to a NodeState
- Transitions from outside a NodeState to a state within the NodeState

Note that combinations of these transition types are possible.

Another thing that changed is the FSMContext class. Instead of a getState() and setState(State) method this class now has a getRoot() method. This method returns a NodeState object that embeds the top layer of the FSM. Events are dispatched to this class.

Nested states are just there to make life easier for the people who have to draw (and maintain) FSM diagrams. The diagrams can be translated to regular FSMs without much trouble. This is the approach I took when implementing nested states.

A transition that is added to the FSM is always translated into one or more LeafState to LeafState transitions. Further more the dispatch mechanism translates events to NodeState objects into events to the right LeafState within that NodeState.

The transition from D to C in Figure 23 for instance is translated into two transitions (A to C and B to C) this way when D receives an event f (which would have triggered D to C), it can safely dispatch it to it’s current embedded state. In the same way transition C to D is translated into C to the default state of D (B).

The changes in the framework may seem radical compared with the old framework, but most of it is hidden by the API (FSM, FSMGenerator and FSMContext in particular). The normal way of using the framework is to create a XML file, generate a FSM object and use that object to create FSMContext objects. From then on this object is used to dispatch events.

This way of working is still in place (though some of the method headers involved have changed). The biggest change for framework users is the different structure of the XML file. The framework is not capable of handling the old versions of the xml files. This could however be done without much problems (by implementing a separate generator class).

The most serious API changes are in FSM. Because there are now two types of states, all the addState() methods have changed into addLeafState() and addNodeState() methods. If the FSMGenerator is used however, this won’t be noticed.

Porting the tools package. To test the framework I had to port all the tool classes I created for the old framework. Most of the classes in this package needed no changes at all. Exceptions were FSMGenerator which needed significant changes in order to be compatible with the FSM class and in order to be able to process the new xml files (with nested state tags).

FSMController needed no change at all. This class is a typical example of how the FSM framework can be used. So it is important that this class is compatible with the new framework.

12.2 Orthogonality and broadcasting events

Orthogonal states are defined as an AND relation between states in Statechart [Harel 86]. Such a situation could exist in a parallel system where each thread has it's own state and all threads share the same context (the system that owns the threads). Orthogonality is very useful for modeling multi-threaded systems.

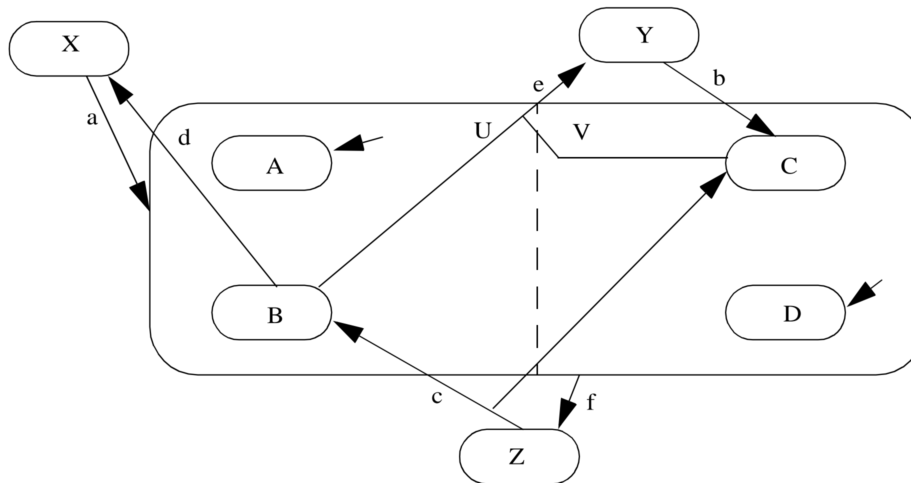

FIGURE 24. **Orthogonal states.**

In Figure 24 an example is given of orthogonal states. The states U and V are orthogonal (indicated by the dashed line). Since U an V have embedded FSMs, the situation can get a little complicated. In the diagram the internal states have been omitted. What's left are six transitions:

- X to U,V triggered by a. This transition causes both U and V to go to their default state.
- Y to C triggered by b. This transition causes V to go to state C and U to go to its default state.
- Z to B,C triggered by c. This transition causes U to go to state B and V to go to state C
- B to X triggered by d. This transition only happens if U is in state B, V is in state D and event d occurs
- B,C to Y triggered by e. This transition only happens if U is in state B, V is in state C and event e occurs
- U,V to Z triggered by f. This transition only happens if U is in state A, V is in state D and event f occurs

The biggest change here is that orthogonal states are represented as vectors of states. Transitions to the orthogonal state are really transitions to a vector of states (one state for each of the embedded FSMs). Likewise, transitions from orthogonal states have a vector of states as source state. If only one state is specified (like in transition d) this means that all the other FSMs in the orthogonal state have to be in their default state.

12.2.1 Requirements for an implementation of Orthogonal states

So a few changes are needed to the framework (i will take the framework with nesting as a starting point):

- a new orthogonal state class is needed
- this class has some very special dispatching behavior
- transitions need to support vectors of states (both as source and as target)

12.2.2 Implementation

Because this is a little out of the scope of this master thesis and because it is a lot of work to implement it, I have decided not to implement it. Instead I will just briefly reflect on likely problems I would have encountered.

If I would follow the same style of implementing as I did for the nested states (which is likely because that is the starting point), a new orthogonal state class would need to be created.

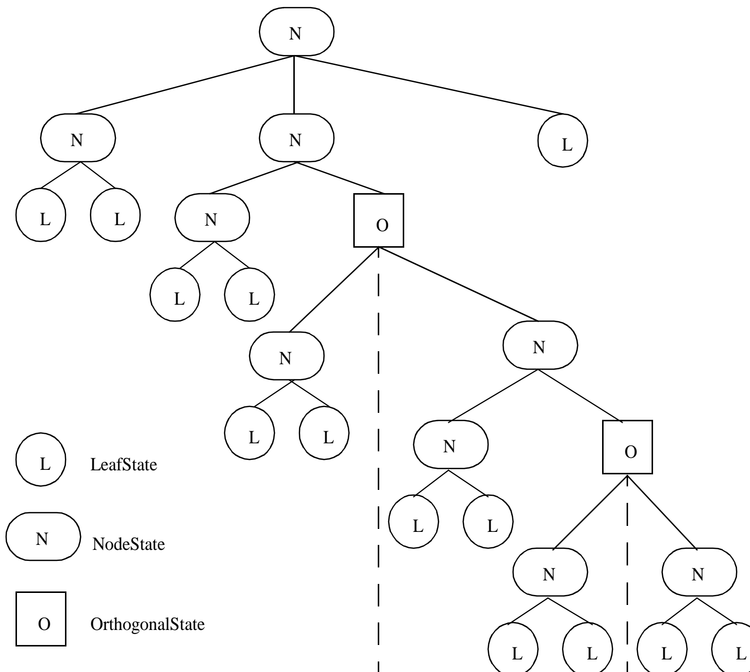

FIGURE 25. State hierarchy.

A FSM in such a framework could be represented as a tree of states. An example of such a state hierarchy tree can be found in Figure 25. Each line in this tree represents a containment relation, not a transition. The children of an orthogonal state are always nodestates (making a single state a child doesn't seem to make much sense and could be implemented anyway by giving the nodestate just one leafstate as a child).

The rules for adding transitions become quite complex. There are many cases which all need to be dealt with in a particular way. There are also illegal transitions. (between children of an orthogonal state). This can however be solved on the XML level (by creating a DTD that ensures correct transitions).

Also the dispatching algorithm becomes more complex. Mainly because it is now possible to have transitions from a group of states to another group of states. Implementing all this is a one time effort though. As soon as it is done, the framework will hide all the complexity.

13 Change scenarios

In paragraph 12 examples were given of two changes to the FSM framework. Those changes were restricted to the framework. A second type of change is when framework instances are changed. These kind of changes happen more often and it is important that a framework sup-

ports this kind of change. If, for instance, pieces of code that are likely to change are scattered over many classes, developers will have a hard time changing this code.

So there are two types of change or evolution:

- Framework Evolution: This is when the framework itself changes.
- Framework Instance Evolution: This is when the programs created with a framework are changed.

In this paragraph we will look at Framework Instance Evolution. We will take a concrete FSM and apply some changes to it. We analyze the effect of these changes on two implementations of the FSM. One implementation will use the regular state pattern and the other will use the FSM framework (together with the FSM Generator).

13.1 A case

Lets take the TCP connection FSM (see Figure 26) and consider different ways of implementing it.This FSM was borrowed from the TCP Request For Comment document [rfc793] that specifies how TCP works. A description of this protocol can be found in paragraph 5.2.

To connect to a computer, the TCP protocol uses the three way handshake protocol. There are two ways to establish a connection in TCP:

- Active connecting, This means that a request for a connection is sent to the other computer.
- Passive connecting. This means that the computer waits till a request for a connection is sent to the computer.

The three way handshake protocol ensures that a connection gets established in the correct way. The first step in establishing a connection is sending a SYN data block to the other computer. If this computer is in the Listen state, it responds by sending a second SYN block plus a ACK (acknowledge) for the received SYN block. The first computer responds to the second SYN block by sending a ACK and goes to the established state. When the second computer receives the ACK it also goes to the established state. The connection is now established and a bidirec-

tional data channel between both computers exists. Breaking the connection is more or less the same. Except now both parties can take the initiative.

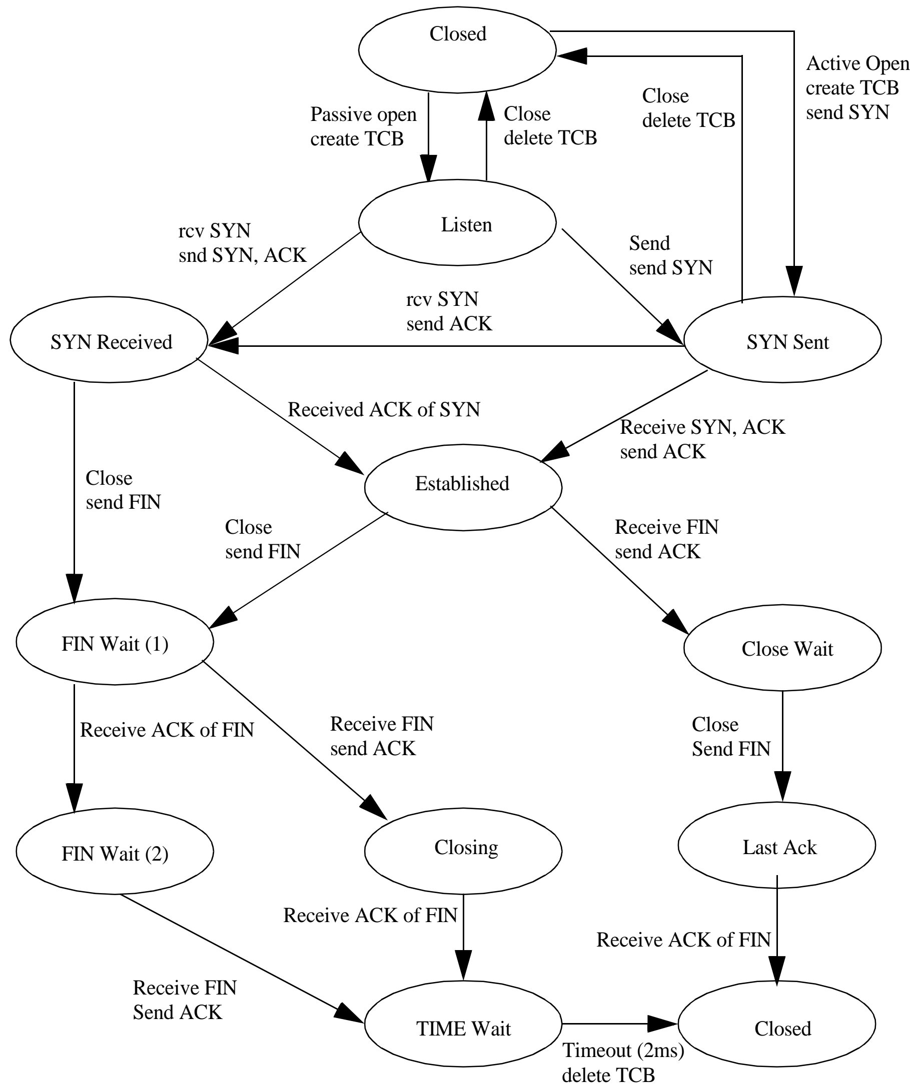

FIGURE 26. **The TCP connection/disconnection FSM.**

13.2 Description of the changes

I tested both implementations for a few cases. I did not actually do the changes but merely analyzed what type of changes would be required to the code. This allows me to reflect on possible problems I might not encounter in these simple examples. Yet, it is concrete enough to draw conclusions about maintenance and reusability.

Approximately 60% to 80% of the development cost goes into maintenance [Taylor]. The framework should reduce the amount of maintenance needed for framework instances. It can do so by imposing a clever design on the system. It can also offer predefined components that can be plugged into the framework instance.

This means that less maintenance has to be done on the framework instance. Maintaining the predefined components is still necessary though. When those components are maintained, all framework instances benefit from it.

Adding states to the FSM. One of the things that could be required in a FSM is adding a state. This should not be too difficult. Later we are going to use the added state to create a new transition.

Because we are working with an existing FSM, most of the changes won't make sense in real life. They are just meant to demonstrate how easy/difficult the change is. In this case I will add a state called "FIN Wait (3)".

Removing states from the FSM. Removing a state is a bit more complex because it also means removing transitions to and from that state. In other words it can be a quite radical change to the structure of the system.

Removing the state "Established" leaves the FSM in a inconsistent state. The states "Close Wait" and "Last ACK" become unreachable. Yet it could be required to do so as an intermediate step for some bigger change.

Adding entry actions. Entry actions are ideal for executing state specific initialization. In this case we'll change the init action of the state "Established". Of course most actions are empty in this implementation. So a change could be to make one not empty and make it do something trivial. We'll make it print "Hello world" on standard out.

Adding exit actions. We'll do the same for the exit action of this state.

Removing transitions. Because of the added transitions, the transition from "FIN Wait (2)" to "TIME Wait" can be removed.

Adding output events. This can be done by adding the "Hello World" action to the previously created transitions.

Adding transitions. For this case we'll use the state "FIN Wait (3)" we created earlier. We'll create a transition from "FIN Wait (2)" to "FIN Wait (3)". After that we'll create one from "FIN Wait (3)" to "TIME Wait". Lets assume both transitions have the same event and action as the existing transition between "FIN Wait (2)" and "TIME Wait".

13.3 The state-pattern approach

Adding states to the FSM. This requires creating new subclasses of State. Also if there exists a hierarchy of states, it may require rearranging this hierarchy in order to be able to inherit all the needed functionality. So we create a new class: FinWait3. Unfortunately we can't take the statename as a classname so we take something that resembles it. For the sake of simplicity we'll have it implement State directly. If we are going to add transitions, we may have to change this in order to be able to reuse behavior defined in other states.

Removing states from the FSM. First all transitions to and from this state have to be removed. If those transitions are not documented this means searching through all the classes where transitions can be initiated (all the State classes + all the visitor classes).

Luckily we have a state diagram so we find four transitions that have to be deleted first (see "Removing transitions" to find out how to do that). Then, if no other states inherit from EstablishedState, the class EstablishedState can be removed from the system.

If however there are States that inherit from EstablishedState, the situation becomes a little messy. We'll end up with a system in which a class exists that is no longer instantiated and possibly contains dead code. Yet we can't remove it because other states inherit from this state. This can be prevented by always making states final. Shared behavior can than be moved up in the hierarchy to abstract states.

Adding entry actions. The state pattern has no explicit mechanism for entry and exit actions. Such a mechanism can however be added by simply equipping the State interface with entry and exit methods. Assuming this was done from the beginning (so every state class has a entry and exit method), the EstablishedState class entry method can now be edited to contain the line of code that prints "Hello world".

Adding exit actions. In a similar way the exit method can be edited to do the same.

Removing transitions. Transitions are not explicitly present in the state pattern. Most likely they will be implemented using methods in one of the State classes. In this case the transition from "FIN Wait (2)" to "TIME Wait" has to be removed.

Transitions are implemented as methods in the State classes. Those methods are called by messenger objects which model the event. Both have to be edited. The problem is finding the points in the code where the messenger objects are sent to the finite state machine. This can be anywhere. In conduits it can even be outside the FSM. This makes removing transitions a bit tricky.

In this case the transitions trigger event is called "Receive FIN" and the accompanying action is "Send ACK". In conduits, the messenger object selects behavior using polymorphism. The context gives a state to the messenger object. The messenger object selects the correct method header and executes it. In this case the method for the source state of the transition has to be edited. Instead of "Send ACK" it should do nothing or throw an error (this event is no longer applicable on this state). Consequently the FinWait2State class should be checked for dead code (code that used to be called by the message object and is no longer used by other transitions).

Adding output events. This is very similar to changing/removing transitions, since the code for these two is mixed. The code for actions is split in two locations. The messenger object contains code and the state object from which the transition starts may also contain code. The messenger object can work on more than one state (by selecting on a superclass of the state instead of the state itself) so changing behavior here changes it for all the states that inherit from that super class.

Adding transitions. In this case two transitions have to be added. For the first transition we will have to find the "Receive FIN" messenger and change the receiveFIN method for the state "FINWait (2)" so that it sets the new state to be "FIN Wait (3)".

Then "FIN Wait (3)" state has to be edited in a similar way.

13.4 The enhanced state-pattern approach

For this case I'll use the FSM framework described in paragraph 11.Naturally I will use the FSMGenerator to generate the FSM. This means that there has to be an xml file describing the FSM. The source for this file can be found in Appendix C.

Adding states to the FSM. This is as simple as editing the xml specification of the FSM. It can be done by adding a line to the states section. The only implementation that may need to be done is implementing the entry and exit actions.

So in this case we'll add the following line:

```
<state name="FIN Wait (3)"/>
```

In this case I won't add any entry or exit actions but that would have been easy to do.

Removing states from the FSM. This is also very simple. If no transactions refer to this state, the corresponding line in the xml file can be removed. Otherwise first the transitions that refer to the state (either as a source or target state) have to be removed.

In this case the state “Established” needs to be removed. This state has two incoming and two outgoing transitions which have to be removed first. After they have been removed the line containing the tag for the “Established” state can be deleted.

Adding entry actions. To add the “Hello world” action to the established state. We first have to create a class (implementing FSMAction) that does this. I’ll assume that actions can be plugged into the FSM as serialized JavaBeans.

So we have to create one first. This can be done with a tool like the Bean Development Kit (BDK) from SUN. We load the new FSMAction class into the tool. Then we create an instance, we configure it (trivial for this class) and we save the configured JavaBean as “HelloWorldAction.ser”.

After that we change the state tag in the xml file for the “Established” state:

```
<state name="Established"
       initactionclass=”HelloWorldAction.ser”/>
```

Adding exit actions. This can be done in the same way as adding an entry action (only the attributename in XML would be exit-action class). We can reuse the HelloWorldAction so all we have to do is find the tag for the established state and change it into this:

```
<state name="Established"
       initactionclass=”HelloWorldAction.ser”
       exitactionclass=”HelloWorldAction.ser”/>
```

Removing transitions. This can be done by removing the corresponding line in the xml file. So in this case we’ll just remove the two tags that contain the transitions. Then of course all actions have to be checked that may launch the event that triggers the transition.

Adding output events. Again we reuse the HelloWorldAction. This time we have to edit the transition tag for

Adding transitions. To add the two transitions to the FSM, two lines need to be added to the xml file:

```
<transition sourcestate="FIN Wait (2)"
            event="Receive FIN"
            targetstate="FIN Wait (3)"
            actionclass="Send ACK"/>
<transition sourcestate="FIN Wait (3)"
            event="Receive FIN"
            targetstate="Time Wait"
            actionclass="Send ACK"/>
```

13.5 Conclusions

The type of changes that are done here are typical design level changes. They involve restructuring a FSM or extending it in some way. These kind of changes are typical for framework instance evolution.

The statepattern does not support all the FSM concepts so often a change in the FSM requires multiple changes in the sourcecode. Removing a transition, for instance, requires changes in several places. The main reason for this is that the code that makes up a transition can be scattered over several classes.

Removing a transition requires the developer to check all those classes. The enhanced statepattern makes this easier. Transitions are defined on a high level in the xml file. So removing a transition means removing one line from the xml file. Of course the place where transitions are initiated is still in the source code (in the action objects).

For all the changes done above, working with the enhanced state pattern is easier than working with the normal state pattern. This would also be true if we hadn’t used the generator because

the enhanced state pattern contains explicit representations of all the FSM concepts (transitions, entry/exit actions, etc.).

Using the generator reduces complexity of the framework enormously. It also makes sure that developers only have to use a small part of the API. In a normal use situation, a developer would create an xml file and serialized action objects. Then the generator would be used to create a FSM object. From then on only the API of FSM and FSMContext is relevant to the developer.

14 Integrating the FSM Framework with Conduits

The reason we developed the FSM Framework is that we wanted to extend the blackbox behavior of Conduits. Conduits provides a blackbox way of composing protocol stacks. The FSM Framework provides a blackbox way to implement FSMs. So the combination would quite powerful. Ideally it would reduce the effort of building a protocolstack to:

- drawing the FSMs (or writing the xml if no drawing tool is available)
- implementing the actions in the FSM
- drawing the protocol stack (assuming that a tool to do so is available)

In this paragraph I will show how the two frameworks can be composed. To do so, I will give a small example to show how everything works. In the example two layers will be implemented. they will communicate together through a conduit like framework. Since there's only two layers I won't need the entire conduits framework. So instead I will use a simplified version of this framework that captures the basics of the original framework.

14.1 Stack specification

The protocol stack we are going to built is an extremely simplified version of TCP/IP. To simulate host two host communication We'll will just launch two of those stacks and link them together by a piece of software (the semi hardware layer). This way host to host communication can be simulated. The stack diagram can be found in Figure 27.

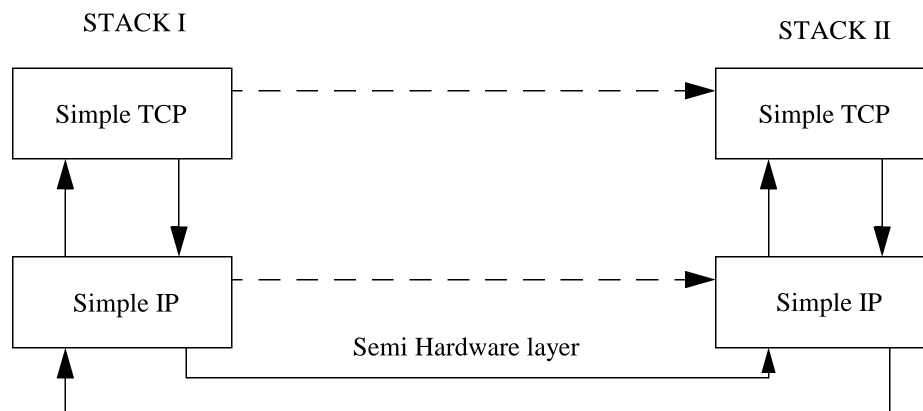

FIGURE 27. **The stack diagram for simple tcp/ip.**

14.1.1 Simplified Conduits

The communication between the layers will be done by a simplified version of conduits. Since there’s only two layers The Mux Conduit won’t be needed and since there’s no hardware the Adaptor Conduit won’t be needed either.

14.1.2 Simplified TCP

For the TCP protocol I will once more use the TCP connection diagram (see Figure 26). By adding one transition it can also send data. The transition will go from the “Established” state to the “Established” state.

```
<transition sourcestate="Established"
            event="Transmit"
            targetstate="Established"
            action="Send data"/>
```

By adding another transition it is also possible to receive data from the lower layer:

```
<transition sourcestate="Established"
            event="Receive"
            targetstate="Established"
            action="Show received data"/>
```

The rest of the diagram remains the same. If the FSM is in the “Established” state it is possible to receive and send data. Data will be ordinary strings in the example. The ACK’s FIN’s and SYN’s that are communicated by the connection FSM can be represented as characters.

A potential problem is the fact that some events are really events with a condition. The “Receive ACK” event for example is really a “receive data” event. The condition that accompanies this event is that the data should contain an ACK. This is going to be a problem because somewhere the “receive data” events will have to be converted to events the FSM understands.

One solution would be to put the condition checking at the place where the event is launched (as suggested in paragraph 10.3). This would mean that the simple IP protocol would have to analyze the data it sends to the simple TCP layer. It is clear that this is not a good solution. A better solution would be to put an extra layer between the simple IP and the simple tcp layer.

This layer would know how to interpret the data received from the simple IP protocol and would translate the events to something the simple TCP FSM understands. But it would be hard to express such a layer as a FSM. Besides it would be a waste of system resources to introduce an extra layer just to preprocess events.

A second place where this can be done is the messenger object. In conduits data and events are delivered by messenger objects. Messenger objects contain some data and implementation. The data would be the packet that is transferred. The implementation part could be used to translate the events. This way the simple IP layer doesn’t have to know anything about the structure of the data that is sent to the simple TCP layer.

This small solution may prove to be a very useful one. It creates a notion of external, non conditional events and internal conditional events. Each external event gets its own Messenger object that triggers the right internal conditional event by checking the condition. To check the condition the Messenger object can use FSMContext and the data sent along with the packet.

Of course a big disadvantage is that part of the blackbox characteristics of the FSM framework is lost here.

14.1.3 Simplified IP

In the simplified IP protocol large packages from the layer above are split into small ones and small packets from below are glued together. This can be expressed in a simple FSM. The simplified IP FSM will only have two states:

```
<state name="ready"/>
<state name="receiving" initaction="check if all data received"
                        exitaction="release collected data"/>
```

There will be four transitions:

```
<transition sourcestate="Ready"
            event="Receive packet below"
            targetstate="Receiving"
            action="Collect data"/>
<transition sourcestate="Receiving"
            event="Receive packet below"
            targetstate="Receiving"
            action="Collect data"/>
<transition sourcestate="Receiving"
            event="Release"
            targetstate="Ready"
            action="Send data"/>
<transition sourcestate="Ready"
            event="Receive packet above"
            targetstate="Ready"
            action="Split & Send data"/>
```

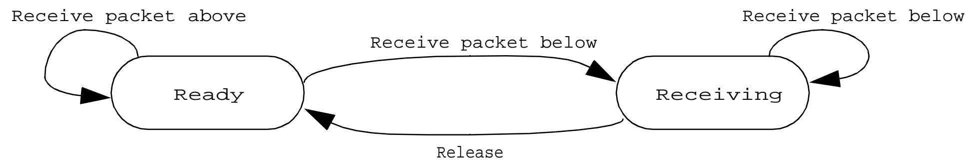

FIGURE 28. **Simplified IP.**

When the simplified IP is in the Ready state it will be capable of both receiving data from below (the hardware layer) or above (the simplified tcp layer). When a packet is received from above, the packet is chopped into pieces of a certain size and given to the lower layer.

When a packet is received from below, the FSM goes to the receiving state. The packet is stored in a special datastructure by the action triggered by the transition event. Then the entry action of the receiving state checks whether more packets can be expected. If so it stays in the receiving state. If not it sends a release event to the context. This causes a transition from “receiving” to “ready”. Upon leaving the “receiving” state the exit action composes all the received data packets into one chunk and gives it to the layer above. Then the action associated with the release event cleans up the datastructure.

Since the simplified TCP protocol sends strings (arrays of characters), a nice choice of packet size is a single character. There will also be a header that indicates the position of the character in the string and the length of the string.

14.2 Implementation

Instead of implementing this mini protocol I will just show how it could be done. The code for the xml for the individual protocols I have already given in paragraph 14.1. All that needs to be specified is the actions. In the xml code I have given descriptions of actions rather than concrete filenames for serialized FSMAction objects.

Most of those actions are quite simple to implement and I won’t go into detail over that. What does need clarification is the communication between the layers. As said before we’ll use a simplified version of conduits (just to show it’s possible).

14.2.1 Simplified conduits

What needs to be done is wrap FSMContext in a conduit like object. As can be seen in paragraph 6.2, conduits have two sides (called A and B). These side can be used to connect to other conduits. Each conduit has a method to accept so called Visitors. A visitor is an object that can interact with the conduit. Vistors are passed to the conduit using the accept(..) method. The conduit then calls the at(..) method on the visitor to activate it. This mechanism allows for several types of visitors. For a more detailed description of this mechanism look at paragraph 6.2.2.

In the simple conduits stack we'll adopt this mechanism. We'll model a conduit as a simple container. Nested in the conduit is a FSMContext (usually the creation of FSMContext is taken care of by the mux and the conduitfactory but since we let those out of the model we'll have to create them ourselves). Each conduit has the capability to connect to other conduits on its A and B side. Those will also be available in the FSMContext repository so that FSMActions can use these connections to dispatch new events and send data.

Sending events and data requires the creation of a new Messenger object that can be wrapped in a MessageTransporter. As stated in paragraph 14.1.2, the messengers will also be used to translate the events. This makes them protocol specific.

14.3 Example

14.3.1 A scenario

In this example the two simple TCP/IP stacks will set up a connection and exchange the line "Hello world". At first the simple TCP layers and the simple IP layers in both stacks are in their default state. Then the sTCP (simple TCP) layer is requested to establish a connection. In Figure 29, the eventtrace for establishing the connection can be seen. Note that the dashed arrows are 'virtual' meaning that in reality the communication flows through the lower layers (sIP -> semi hardware layer -> sIP).

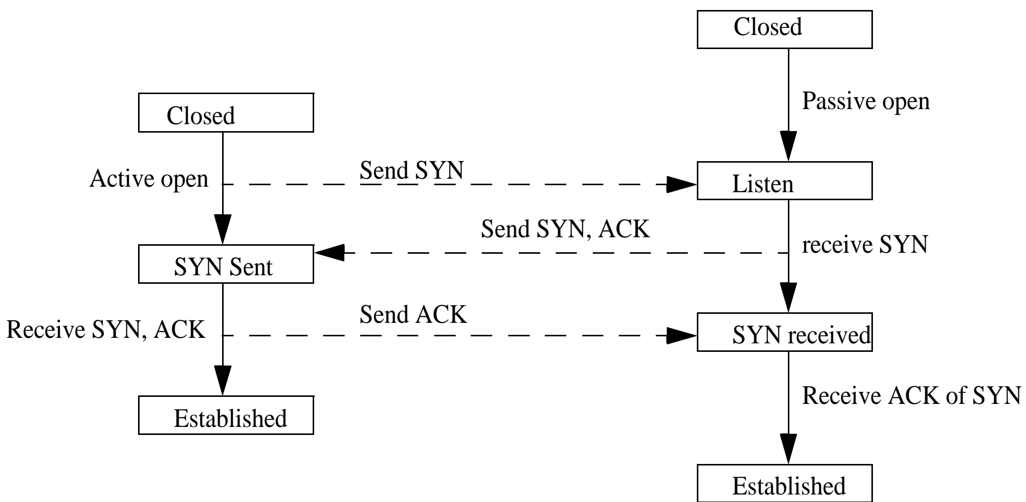

FIGURE 29. Establishing a connection in the sTCP layer.

The communication starts with the sTCP layer in stack II going to the listen state. This requires no communication with the lower layers. After a while stack I is requested to establish a connection with stack II. Stack I goes from the closed state to the SYN sent state by sending a SYN packet to stack II.

This action uses the lower layer (sIP) to sent the SYN packet. So a Messenger is created that encapsulates a Receive data event and the SYN character.

The sIP layer is in the Ready state. It accepts the Messenger and treats the SYN as normal data. Since a SYN is represented as a single character, the data does not have to be split. The sIP layer adds a header to the data packet saying that this is packet 1 of 1 packets. Then this packet is given to the Semi hardware layer (in a appropriate Messenger).

This layer delivers the packet to the sIP layer in stack II. This layer is in the Ready state. Upon receiving the packet it goes to the Receiving state. Upon entering this state it is noted that no more packets are expected so a Release event is sent to the layer. This causes a transition to the ready state. Upon leaving the Receiving state, the packet is given to the upper layer (without the sIP header).

To do that a Messenger is created for the sTCP layer that contains the data. The messenger analyses the data and concludes that a SYN has been sent. So the SYN Receive event is given to the sTCP layer. This triggers a transition to the state SYN Received.

The rest of the communication goes in a similar way until both sTCP layers are in the Established state. Now it is time to sent the string “Hello world“ to the other side. So a Transmit event is sent to the sTCP layer. This causes a transition from the Established state to itself. The transition causes string to be wrapped in a Messenger for the sIP layer.

The sIP layer splits the data up in 11 packets of 1 character each. One by one they are sent (using the semi hardware layer) to the sIP layer in the other stack. After this layer receives the first packet, it goes from the Ready to the receiving state (storing the data in a buffer). After that each incoming packet causes a transition from the Receiving state to itself (adding the data to the buffer).

Each time the Receiving state is reentered, it is checked whether all data has been received. When the last packet arrives a Release event is sent to the sIP layer. That causes a transition form Received to Ready. The data is collected and wrapped in a Messenger for the sTCP layer. The messenger translates the data in a Receive event and this causes a transition from Established to Established.

14.4 Experiences

14.4.1 Problems with the FSM framework

This small experiment with the FSM framework and conduits revealed a problem. The FSM framework does not support conditional transitions. This means that the condition checking has to be done elsewhere. Checking conditions cannot be done in a blackbox way (i.e. it is not possible to create a Condition component that can be configured). So some implementation is needed.

The conduits framework offers some space for condition checking in the form of Messenger objects. This however not an ideal solution because it requires that glue code must be written to use simple Conduits with the FSM framework.

An alternative solution requires moving the condition checking into the FSM framework. This could be done by creating a Condition interface (like the FSMAction interface) with a boolean check method. Objects that implement this interface could be associated with Transitions. This would be a good solution because it would effectively add support for conditions to the framework while retaining as much of the blackbox behavior as possible.

14.4.2 Composition

The main reason I did this small experiment with the FSM framework was to see how easy it was to compose it with other frameworks. Considering the success of conduits with the ‘plumbing’ of a protocol stack, conduits was a nice choice to compose the FSM framework with. Featurewise they complement each other.

Unfortuantely it was not possible to plug the FSM framework in a out of the box version of Conduits. One reason for this was the fact that Conduits has it's own notion of a statemachine. To solve this problem a new type of Conduit had to be developed that (as far as I can see) could be used in conjunction with the existing one. The new conduit wraps around a FSMContext object. The messenger objects in conduits can be used to deliver the events to the FSM.

Section 5 Conclusion

The main goal of this thesis was to find a way to make sure that frameworks evolve in a nicer way and are easier to compose with other frameworks. In Section 2 a set of guidelines is presented that attempt to achieve this. The guidelines are merely recommendations and should not be treated as rules that should always be followed blindly. However, they are a step forward in developing better frameworks.

A secondary goal of this thesis was to improve frameworks in the domain of communication protocols. When the guidelines from Section 2 were matched against existing frameworks in this domain, I found that a few guidelines had not been followed. The most important conclusion was that existing frameworks are addressing two domains (stacks and protocols) which, according to my guidelines, should result in splitting the framework.

Based on that conclusion I decided to built my own framework to support the development of protocols. The resulting framework implements a variation of the State Pattern [Gamma] that improves on the original pattern in several ways. Unlike the State Pattern it is blackbox and it does supports more of the finite state machine paradigm's concepts. The resulting framework was also used to validate the guidelines. This was done by applying changes to the framework and it's instances.

15 Summary

The research for this thesis was done in four phases. In the Section 2 a set of guidelines for building frameworks is presented. In Section 3 the domain of communication protocols is introduced. Also an overview of important frameworks in that domain is given. In Section 4 a framework for implementing finite state machines is presented. In the same section evolution and composability of this framework are testsed.

15.1 The Guidelines

By analyzing framework evolution and composition I was able to construct a set of guidelines (presented in paragraph 4). The guidelines are grouped into four classes:

- Process. The guidelines in this class focus on the process of making a framework. This process is slightly different from the process of developing a normal OO system. The most important difference is that the process cannot be iterated endlessly like in a normal OO system. The main reason for this is that during development the framework can be used to create applications. This results in a more conservative change policy for the framework.
- Design. The guidelines in this class focus on the design of the architecture. These are the most important guidelines since they determine the flexibility of the framework. Flexibility is essential when the frameworks needs to be adapted (either for evolution or composition).
- Component. The guidelines in this class focus on improving the blackbox behavior of the framework. Blackbox frameworks are much easier to use than whitebox frameworks. In many

cases framework configuration can even be automated (making the task of customizing a framework trivial).
- Implementation. A few simple implementation tricks can make life of developers a lot easier. The guidelines in this class could apply to any OO system. For frameworks it is essential that the source code is 'clean' and easy to understand.

15.2 Existing Communication Protocol Frameworks

In Section 3 I analyzed a few frameworks in the domain of communication protocols. The following frameworks were examined: the x-kernel, Conduits+ and Java Conduits and the Network Protocol Framework of Axis. All of the frameworks used a OSI like layered model for protocol stacks. All those framework used more or less the same abstractions. All of the frameworks had a notion of demultiplexing data to different modules in the stack.

The Conduits frameworks tried to model another thing: The protocols itself. It was assumed that protocols could be represented as Finite State Machines. Thus the state pattern was used to model protocols. Based on my guidelines in paragraph 4 (Guideline 5 and Guideline 6 in particular) I concluded that Conduits tried to model two domains and should be split into two frameworks. The two domains were:

- Protocol stack configuration & management
- Protocol/layer implementation (in the form of a FSM)

15.3 Addressing the shortcomings of the State pattern

The solutions offered by Conduits for the first domain (protocol stack configuration & management) appeared to be good ones. Protocol stacks could be composed in a blackbox way. The application of design patterns such as the Visitor pattern and the Prototype pattern ensured great flexibility.

For the second domain there were some problems however. Though it was assumed that protocols can be represented as FSMs, the framework offered little support for implementing them. All that was done was ensure that the state pattern was used. I found four problems with the state pattern that prevented easy implementation of protocols:

- The existing state pattern does not provide explicit representations for all the FSM concepts. This makes maintenance hard because it is not obvious how to apply a design change to the implementation.
- The state pattern is not blackbox. Building a protocol requires the developer to extend classes rather than to configure them.
- Implementations are complex.
- The state pattern prevents reuse of behavior by tying behavior to states.

To solve these problems I created a framework that offers support for FSMs. This also allowed me to test my own guidelines. My framework addresses these problems by providing implementations for all the key elements in FSMs (states, events, transitions, entry-actions, exit-actions, transition actions). By using explicit representations of these elements, it is possible to configure the framework in a blackbox way.

To demonstrate the blackbox configuration, I made a FSM Generator tool that configures the framework based on a specification in XML. The only things that have to be implemented by a developer are the actions that are plugged into the FSM (entry-actions, exit-actions and transition actions).

15.4 Validation

To validate the guidelines. I took my own framework and applied some changes to it. The changes can be grouped into three classes:

- Framework evolution.
- Framework instance evolution.
- Framework composition.

To test my framework for evolution, I tried to change it to support Harel's FSM extensions [Harel 86] (nesting and orthogonality). I managed to implement nesting. I also showed how orthogonality could be implemented in a similar way.

Though the framework changed considerably the effect on existing applications was minimal since the framework was configured almost completely by providing an FSM specification in xml. This stressed the importance of good blackbox behavior. Apparently good blackbox behavior in a framework allows for more radical changes in the framework.

To test the framework's blackbox behavior I applied some changes to a little test application. Two versions of this application were used (one implementation with the state pattern, one implementation with my new framework). The main conclusion of this test was that it was considerably easier to do the changes with the new framework.

Finally, to test composability I tried to compose my framework with Conduits. Since I didn't have a working version of Conduits, I had to restrict myself to an experiment on paper. To do so I made an example protocol stack (simple TCP/IP) and tried to show how both protocols could be composed in a stack.

16 Lessons Learned

The set of guidelines I created are a useful tool to judge frameworks. They helped me to find shortcomings in the Conduits framework. And also they explained why parts of this framework are very good and other parts aren't. They don't guaranty composability and good evolution though. They are to general for that. Guidelines are less formal than a real development method. This good because this allows for some flexibility in applying the guidelines.

The main point of my guidelines is that it's important to keep frameworks specialized and that it's important to make frameworks easy to use. The guidelines seem to work both for evolution and composition.

Good frameworks simplify the structure of a program for the developer by providing implementation for the more complex part. Their primary value is not in offering code reuse but by imposing structure on a program. Only once structure is in place, code reuse becomes more important. This can be achieved by providing components that use that framework.

In a way a framework becomes a new language/paradigm. It has its own constructs, it has to be used in a certain way and developers tend to think in terms of framework constructs rather then in terms of the underlying language. Especially when a configuration language is involved, this becomes apparent.

17 Future work

17.1 Communication protocol frameworks

Though on paper the composition of my framework with Conduits seems simple, only a real implementation could confirm this. A nice test would be to implement a complete protocol-stack using both conduits and the FSM framework.

Such a test would also be useful to do some performance tests. Though I don't expect that the FSM framework will deliver worse performance than the original State pattern, only a performance test can deliver evidence for that.

The FSM framework offers a starting point to offer default services like queues and timers. Implementations for these kind of things are now independent of the FSM they are used in.

The FSM framework could be extended to support all of Harel's statechart features [Harel 86]. It already supports nesting. And I've shown it would be easy to implement orthogonality as well. A nice test for such a framework would be to implement the example FSM (a statechart modeling Harel's digital watch), Harel uses in his article.

17.2 Framework composition & evolution issues

Though I tested my framework for evolution and composition, a larger experiment would be needed to draw any conclusions. This would also be an opportunity to discover more guidelines.

The guidelines I presented in paragraph 4 appear to contribute to the quality of frameworks. Perhaps more guidelines can be found. If enough guidelines are collected, the result might be a complete development method. Also it would be useful to rewrite the guidelines in the form of a pattern language like in [Johnson].

References

[Bosch]. J. Bosch, P. Molin, M. Mattson, PO Bengtsson, *"Object Oriented Frameworks - Problems & Experiences"*, University of Karlskrona/Ronneby, Dep. of Computer Science and business administration, S-273 25 Ronneby, Sweden

[taligent]. *"Building Object-Oriented Frameworks"*, http://www.ibm.com/java/education/oobuilding/index.html

[Mattsson 96a]. M. Mattson, J. Bosch, *"Framework Composition – Problems, Causes and Solutions"*, University of Karlskrona/Ronneby, Dep. of Computer Science and business administration, S-273 25 Ronneby, Sweden

[Mattsson 96b]. M. Mattson, *"Object-Oriented Frameworks – A Survey of Methodological Issues"*, Department of computer science, Lund University, 1996

[Demeyer]. S. Demeyer, T. D. Meijler, O. Nierstraz, P. Steyaert, *"Design Guidelines for Tailorable Frameworks"*, Communications of the Acm, 40, 10 October 1997, p60-64

[Johnson]. D. Roberts, R Johnson, "Patterns for evolving frameworks", Pattern Languages of Program Design 3 (p471-p486), Addison-Wesley, 1998

[Gamma]. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *"Design Patterns - Elements of Reusable Object Oriented software"*, Addison Wesley, 1995

[rfc791]. *"Internet Protocol – DARPA Internet Program Protocol Specification"*, RFC 791, September 1981

[rfc793]. *"Transmission Control Protocol – DARPA Internet Program Protocol Specification"*, RFC 793, September 1981

[rfc765]. *"File Transfer Protocol (FTP)"*, RFC 765, October 1985

[rfc1945]. *"Hypertext Transfer Protocol - HTTP/1.0"*, RFC 1945, May 1996

[Hutchinson]. N. C. Hutchinson, L. L. Peterson, *"The x-kernel: An Architecture for Implementing Network Protocols"*, IEEE Transactions on Software Engineering, vol 17, no 1, January 1991

[xkerneltutorial]. L. L. Peterson, B. S. Davie, A. C. Bavier, *"The x-kernel Tutorial"*, http://www.cs.arizona.edu/xkernel/index.html

[Zweig]. J. M. Zweig, R. E. Johnson, *"The Conduit: a Communication Abstraction in C++"*, Usenx C++ Conference 1990

[Hüni]. H. Hüni, R. Johnson, R. Engel, *"A Framework for Network Protocol Software"*, Object Oriented Programming Systems , Languages and Applications Conference Proceedings (OOPSLA '95), ACM Press 1995

[Nikander]. P. Nikander, A. Karila, *"A Java Beans Component Architecture for Cryptographic Protocols"*, Proceedings of the 7th Usenix Security Symposium, January 26-29, 1998 San Antonio, Texas

[jacob]. http://www.tcm.hut.fi/Research/TeSSA/Jacob/index.html

[javasoft]. http://www.javasoft.com/

[Ungar & Smith]. D. Ungar, R. B. Smith, *"SELF: The Power of Simplicity"*, OOPSL '87 Conference Proceedings

[Tanenbaum]. A. S. Tanenbaum, *"Computer Networks - second edition"*, Prentice Hall, 1988

[xml]. http://www.w3c.org/XML/index.html

[xml4j]. http://www.alphaworks.ibm.com/Home/index.html

[w3c]. http://www.w3c.org/index.html

[Harel 86]. D. Harel, *"Statecharts: a Visual Approach to Complex Systems(revised)"*, report CS86-02 Dep. App Math's Weizman Inst. Science Rehovot Israel, March 1986

[Harel 88]. D. Harel, *"On visual formalisms"*, Communications of the ACM vol 31 no5, May 1988

[Taylor]. M. J. Taylor, A. T. Wood-Harper, *"Methodologies and Software Maintenance"*, Software Maintenance: Research and Practice, vol 8, p295-308 (1996)

Appendix

Appendix A Meeting at Axis January 25, 1999

After several contacts with Torbjörn Söderberg about Axis' network protocol framework, he brought me in to contact with Per Flock, the person responsible for this framework. He gave me some design documents and we agreed to meet.

In this meeting we discussed the framework currently under design at Axis. This paragraph is a report of that meeting. I grouped my questions into three classes:

- Framework usage
- The Protocol Stack
- Individual Protocols

A.1 Framework Usage

How much coding is needed to develop a protocol using the framework? A lot of things like routertables, timers, queues and different algoritms need to be implemented. This is a lot of work.

How much is the framework influenced by older software? Compatibility with older source code is important to some extent. One of the design goals of the new framework was to make it easy to port old software to the new framework.

How portable is the framework? The framework is programmed entirely in C++ which in itself is different across platfporms. OS specific systemcalls don't add to the portability too.

What's the advantage of using the framework? It enforces a design upon the code. The individual layers are implemented more uniformly which makes them easier to understand and maintain. The designer (Per Flock) also mentioned that he likes smaller frameworks (i.e. to the point and simple)

A.2 The Protocol Stack

How much work is it to build a protocol stack from existing protocol components and drivers? The system is configured at boottime. The modules that make up the stack are loosely coupled. Some thought has been given to making a tool to configure the stack.

How much did Hüni's work influence the design? Some of the patterns inspired the design. Key elements of his design (the four different conduits for instance) are not represented as separate entities in Axis' though. Rather than conduits, the concept of layers is used.

A.3 Individual Protocols

How much support does the framework offer when developing individual protocols?

The framework does not offer a reusable design for individual protocols. Part of the reason is that the framework aims at supporting a wide range of protocols from simple stateless protocols like IP up to complex high-level protocols like HTTP. Most of the things offered by the framework are low level features like memory management and thread management.

Protocols are usually modeled using FSMs. Do you use them too? If so how? FSMs are only used for more complex protocols like TCP. Simpler protocols like IP can be implemented without a state machine because no state exists between individual frames. If a FSM is used, the State pattern is applied to implement it. Nested case statements exist in legacy code (up to five levels)

Is each layer in the protocol stack a single FSM? The framework does not enforce a structure for the layers. Mostly complex protocols are implemented as nested FSMs (using the state pattern).

Appendix B Enhanced State Pattern

B.1 FSM

```
package newstate;

import java.util.*;

/**
* This class serves as a access point for the whole framework. An FSM object
* encapsulates a FSM and provides a factory method to create a FSMContext
* object for this FSM.
*/
public class FSM
{
        Hashtable states = new Hashtable();
        Hashtable events = new Hashtable();
        State first = null;
        FSMAction initaction = null;

        /**
        * This method can be used to add a state to the FSM.
        * FSM uses the State class to create a new State object
        * @param statename The name of the state
        */
        public void addState(String statename)
        {
                if(!states.containsKey(statename))
                {
                        State s = new State(statename);
                        states.put(statename, s);
                }
                else throw new RuntimeException("state; " + statename +
                                        " already declared");
        }

        /**
        * This method can be used to add a state to the FSM.
        * FSM uses the State class to create a new State object
        * @param entryAction The action that is executed upon state-entry
        * @param statename The name of the state
        */
        public void addState(FSMAction entryAction, String statename)
        {
                if(!states.containsKey(statename))
                {
                        State s = new State(statename);
                        s.setStateEntryAction(entryAction);
                        states.put(statename, s);
                }
                else throw new RuntimeException("state; " + statename +
                                        " already declared");
        }

        /**
        * This method can be used to add a state to the FSM.
        * FSM uses the State class to create a new State object
        * @param statename The name of the state
        * @param exitAction The action that is executed upon state-exit
        */
        public void addState(String statename, FSMAction exitAction)
        {
                if(!states.containsKey(statename))
                {
                        State s = new State(statename);
```

```
                s.setStateExitAction(exitAction);
                states.put(statename, s);
        }
        else throw new RuntimeException("state; " + statename +
                                        " already declared");
}

/**
* This method can be used to add a state to the FSM.
* FSM uses the State class to create a new State object
* @param entryAction The action that is executed upon state-entry
* @param statename The name of the state
* @param exitAction The action that is executed upon state-exit
*/
public void addState(FSMAction entryAction, String statename,
                     FSMAction exitAction)
{
        if(!states.containsKey(statename))
        {
                State s = new State(statename);
                s.setStateEntryAction(entryAction);
                s.setStateExitAction(exitAction);
                states.put(statename, s);
        }
        else throw new RuntimeException("state; " + statename +
                                        " already declared");
}

/**
* This method can be used to add an event to the FSM.
* FSM uses the Event class to create a new Event object
* @param name This is the name of the event.
*/
public void addEvent(String name)
{
        if(!events.containsKey(name))
        {
                events.put(name, new FSMEvent(name));
        }
        else throw new RuntimeException("event; " + name +
                                        " already declared");
}

/**
* This method creates a transition between the sourcestate and the
* target state. The method checks whether the given states and the
* event exist before it creates the transition. If they don't exist
* a RuntimeException is thrown.
* @param sourcestate The name of the sourcestate
* @param eventname The name of the event that triggers the transi-
tion
* @param targetstate the name of the targetstate
* @param action The action that will be executed when the transition
* is triggered.
*/
public void addTransition(String sourcestate, String eventname,
                          String targetstate, FSMAction action)
{
        if(states.containsKey(sourcestate))
        {
                State s = (State)states.get(sourcestate);
                if(states.containsKey(targetstate))
                {
                        State t = (State)states.get(targetstate);
                        if(events.containsKey(eventname))
                        {
                                FSMEvent e =
                                     (FSMEvent)events.get(event-
```

```
name);
                                        s.addTransition(e, t, action);
                                }
                                else throw new RuntimeException("event; " +
                                      eventname + " not found");
                        }
                        else throw new RuntimeException("state; " + targetstate
                               + " not found");
                }
                else throw new RuntimeException("state; " + sourcestate +
                                                " not found");
        }

        /**
        * This method is used to set the default state for the FSM. If a
        * FSMContext is created, this state is set as the current state.
        * @param statename The name of the first action.
        */
        public void setFirstState(String statename)
        {
                first = (State)states.get(statename);
        }

        /**
        * Sometimes it's necessary to do some initialization before the FSM can
        * be used. For this purpos a initial action can be set. This action is
        * executed when the FSMContext is created.
        * @param action The initial action.
        */
        public void setInitAction(FSMAction action)
        {
                initaction = action;
        }

        /**
        * This method serves as a factory method to create FSMContexts from
        * the FSM. Also the init action is run (if available).
        * @return A new FSMContext for the FSM.
        */
        public FSMContext createFSMInstance()
        {
                FSMContext fsmc;
                if(first == null) throw new Error("first state not set");
                else
                {
                        if(initaction != null)
                        {
                                fsmc =  new FSMContext(first, initaction);
                        }
                        else
                        {
                                fsmc =  new FSMContext(first);
                        }
                        fsmc.initialize();
                        return fsmc;
                }
        }
}
```

B.2 FSMAction

```
package newstate;

/**
* The FSM uses the command pattern to implement actions. All actions must
* implement this interface.
```

```
*/
public interface FSMAction
{
        /**
        * @param fsmc This is the context in which the command is executed.
The
        * context can be used as a repository for objects. That is because
        * FSMContext extends from java.util.Hashtable.
        * @param data Some extra data that can be given to a command
        */
        public void execute(FSMContext fsmc, Object data);
}
```

B.3 FSMContext

```
package newstate;

import java.util.*;

/**
* This is the context of a FSM. A context holds a reference to the current
state
* and also functions as a repository for objects. To do that it extends from
* java.util.Hashtable. Objects can be stored with the hashtable's put method
and
* retrieved with the hashtables get method.
*/
public class FSMContext extends Hashtable
{
        private State state;
        private State firstState;
        private FSMAction initialAction = State.skip;

        public FSMContext()
        {
        }

        /**
        * Create a new context with s as the first state
        * @param s The first state
        */
        public FSMContext(State s)
        {
                super();
                setFirstState(s);
                initialize();
        }

        /**
        * Create a new context with s as the first state. an then execute i
to
        * initialize the context.
        * @param s The first state
        * @param i The initial action
        */
        public FSMContext(State s, FSMAction i)
        {
                super();
                setFirstState(s);
                setInitialAction(i);
                initialize();
        }

        /**
        * Find out what events can be sent to the current state
        * @return A list of events
```

```
        */
        public Vector getEvents()
        {
                return getState().getEvents();
        }


        /**
        * Initialize the context. Set first state and execute initial action.
        */
        public void initialize()
        {
                this.clear();
                initialAction.execute(this, null);
                setState(firstState);
                firstState.getStateEntryAction().execute(this, null);
        }

        public void setState(State s) { state = s; }
        public State getState() { return state; }

        public void setFirstState(State s) { firstState = s; }
        public State getFirstState() { return firstState; }

        public void setInitialAction(FSMAction a) { initialAction = a; }
        public FSMAction getInitialAction() { return initialAction; }

        /**
        * Dispatch an event e.
        * @param e The event
        - @param data Some additional data
        */
        public void dispatch(FSMEvent e, Object data)
        {
                getState().dispatch(e, data, this);
        }
}
```

B.4 FSMEvent

```
package newstate;

public class FSMEvent
{
        public String name;

        /**
        * Creates an event with name s
        * @param s The name of the new event
        */
        public FSMEvent(String s)
        {
                setName(s);
        }

        public void setName(String s)
        {
                name = s;
        }

        public String getName()
        {
                return name;
        }

        /**
```

```
        * @return The name of the event
        */
        public String toString()
        {
                return getName();
        }
}
```

B.5 State

```
package newstate;

import java.util.*;
import newstate.tools.*;

/**
* This class models a State. A state has a name and entry /exit actions.
* Further state also contains a dispatch mechanism for incoming events.
*/
public class State
{
        private String name;
        private Hashtable transitions = new Hashtable();

        public static SkipAction skip = new SkipAction();

        FSMAction onStateEntry = skip;
        FSMAction onStateExit = skip;

        /**
        * Initilizes a stateobject with name s.
        * @param s The name of the new state
        */
        public State(String s)
        {
                setName(s);
        }

        public void setName(String s) { name = s; }
        public String getName() { return name; }

        public void setStateEntryAction(FSMAction action)
        {
                onStateEntry = action;
        }
        public FSMAction getStateEntryAction()
        {
                return onStateEntry;
        }

        public void setStateExitAction(FSMAction action)
        {
                onStateExit = action;
        }

        public FSMAction getStateExitAction()
        {
                return onStateExit;
        }

        /**
        * Convenience method that returns the name of this state.
        */
        public String toString()
        {
                return getName();
```

```
        }

        /**
        * Adds a transition with this state as source and parameter to as a
        * target.
        * @param trigger The event that triggers the transition
        * @param to The target state.
        * @param action The associated action
        */
        public void addTransition(FSMEvent trigger, State to, FSMAction action)
        {
                transitions.put(trigger, new Transition(this,to,action));
        }

        /**
        * Dispatch an event.
        * @param trigger The event that needs to be dispatched. The correct
        * transition is located and than executed.
        * @param data Some additional data that may be needed by the action
        * @param fsmc The context in which the action is executed. This may be
        * useful for retrieving global variables.
        */
        public void dispatch(FSMEvent trigger, Object data, FSMContext fsmc)
        {
                getStateExitAction().execute(fsmc, data);
                ((Transition)transitions.get(trigger)).execute(fsmc, data);
                getStateEntryAction().execute(fsmc, data);
        }

        /**
        * Method to find out which events can be dispatched by this state.
        * @return A vector with the events
        */
        public Vector getEvents()
        {
                Vector v = new Vector();
                for(Enumeration e = transitions.keys() ; e.hasMoreElements();)
                {
                        v.addElement(e.nextElement());
                }
                return v;
        }
}
```

B.6 Transition

```
package newstate;

/**
* Transitions are created on the fly and should not be created manually.
*/
class Transition
{
        State target;
        State source;

        FSMAction action;

        public Transition(State s, State t, FSMAction a)
        {
                source = s;
                target = t;
                action = a;
        }

        public void execute(FSMContext fsmc, Object data)
```

```
        {
                // trigger a state exit event in the old state
                // execute the action
                if(action != null)
                        action.execute(fsmc, data);
                fsmc.setState(target);
        }
}
```

Appendix C TCP Connection Protocol

Below is the xml code that specifies the TCP connection/disconnection fsm. It's based on the diagram in [rfc793]. Note that the actions specified for the transitions in this file are just descriptions. The file can be used as input for the FSM Generator.

```
<?xml version="1.0"?>
<fsm firststate="Closed">
        <states>
                <state name="Closed"/>
                <state name="Listen"/>
                <state name="SYN Received"/>
                <state name="SYN Sent"/>
                <state name="Established"/>
                <state name="FIN Wait (1)"/>
                <state name="FIN Wait (2)"/>
                <state name="Closing"/>
                <state name="Close Wait"/>
                <state name="Last ACK"/>
                <state name="Time Wait"/>
        </states>
        <events>
                <event name="Passive open"/>
                <event name="Active open"/>
                <event name="Close"/>
                <event name="Send"/>
                <event name="Received ACK of SYN"/>
                <event name="Receive ACK of FIN"/>
                <event name="Receive SYN, ACK"/>
                <event name="Receive FIN"/>
                <event name="Receive SYN"/>
                <event name="Timeout (2ms)"/>
        </events>
        <transitions>
                <transition sourcestate="Closed"
                            event="Passive open"
                            targetstate="Listen"
                            actionclass="create TCB"/>

                <transition sourcestate="Listen"
                            event="Close"
                            targetstate="Closed"
                            actionclass="delete TCB"/>

                <transition sourcestate="Closed"
                            event="Active open"
                            targetstate="SYN Sent"
                            actionclass="create TCB\nsend SYN"/>

                <transition sourcestate="SYN Sent"
                            event="Close"
                            targetstate="Closed"
                            actionclass="delete TCB"/>
```

```
<transition sourcestate="Listen"
            event="Receive SYN"
            targetstate="SYN Received"
            actionclass="sent SYN, ACK"/>

<transition sourcestate="Listen"
            event="Send"
            targetstate="SYN Sent"
            actionclass="send SYN"/>

<transition sourcestate="SYN Sent"
            event="Receive SYN"
            targetstate="SYN Received"
            actionclass="send ACK"/>

<transition sourcestate="SYN Sent"
            event="Receive SYN, ACK"
            targetstate="Established"
            actionclass="send ACK"/>

<transition sourcestate="SYN Received"
            event="Received ACK of SYN"
            targetstate="Established"
            actionclass="do nothing"/>

<transition sourcestate="SYN Received"
            event="Close"
            targetstate="FIN Wait (1)"
            actionclass="Send FIN"/>

<transition sourcestate="Established"
            event="Close"
            targetstate="FIN Wait (1)"
            actionclass="send FIN"/>

<transition sourcestate="Established"
            event="Receive FIN"
            targetstate="Close Wait"
            actionclass="send ACK"/>

<transition sourcestate="FIN Wait (1)"
            event="Receive ACK of FIN"
            targetstate="FIN Wait (2)"
            actionclass="do nothing"/>

<transition sourcestate="FIN Wait (1)"
            event="Receive FIN"
            targetstate="Closing"
            actionclass="send ACK"/>

<transition sourcestate="FIN Wait (2)"
            event="Receive FIN"
            targetstate="Time Wait"
            actionclass="Send ACK"/>

<transition sourcestate="Closing"
            event="Receive ACK of FIN"
            targetstate="Time Wait"
            actionclass="do nothing"/>

<transition sourcestate="Time Wait"
            event="Timeout (2ms)"
            targetstate="Closed"
            actionclass="delete TCB"/>

<transition sourcestate="Close Wait"
            event="Close"
            targetstate="Last ACK"
```

```
                        actionclass="Send FIN"/>

            <transition sourcestate="Last ACK"
                        event="Receive ACK of FIN"
                        targetstate="Closed"
                        actionclass="do nothing"/>
        </transitions>
</fsm>
```