REENGINEERING DEPRECATED COMPONENT FRAMEWORKS: A CASE STUDY OF THE MICROSOFT FOUNDATION CLASSES

Robert Neumann, Sebastian Günther, Niko Zenker [1]

Abstract
In today's application engineering, the implementation of frameworks and related technology boosts development quality and reduces related effort. Framework functionality embodies expert knowledge and is driven towards reuse. While stable from a conceptual point of view, technological changes require constant adaptation and reengineering. This article presents overall framework engineering principles and practices (FEPP) and shows their concrete application using the example of the Microsoft Foundation Classes. Abstracting from the case study, the focus of this work is upon introducing particular methods for how to cut down on the complexity of maintenance projects by considering the FEPP during framework development.

1. Motivation

As offsprings of the object-oriented programming paradigm, component frameworks in the past were subject to intensive research that reached its peak in the 1990s. Economies of frameworks were derived from framework engineering principles which have established a collection of concepts and best practices describing how to efficiently develop software frameworks. Even though framework development, maintenance, and discontinuance are well analyzed phases in the framework lifecycle, the reactivation of an already discontinued framework seems to be a rather unexplored discipline.

A very successful and well known example in the domain of Windows application development is the Microsoft Foundation Classes (MFC). However, as with the introduction of the .Net framework in 2002 Microsoft changed its focus away from native towards managed development, the MFC were decided to be not longer maintained. Contrarily, the number of Independent Software Vendors (ISVs) outside the managed world was still significant, whereby some of them had established gigantic code bases in native C/C++ code with the MFC interfacing between their application and the Windows operating system. Those ISVs were dependent on the MFC being updated to expose new Windows features as they wanted to support their products even on modern Windows versions.

Visual Studio 2008 (released in October 2007) for the first time after more than ten years contained a major set of updates for the MFC demonstrating an attempt to bring the MFC back to the market.

[1] Otto-von-Guericke-Universität Magdeburg, Universitätsplatz 2, 39106 Magdeburg

The resurrection of the MFC moved the question of how to assess the maintainability of a "reactivated" framework into the focus of our investigations. Using the MFC, we investigated places in a framework that are sensitive to future refinement projects. We analyzed how well the architecture of the framework supports maintenance and what can be done to reduce the intensity in effort those projects require. A resulting catalog of refinement points compares the architecture of a framework to framework engineering principles and practices (FEPP) and discusses how deviations from the FEPP might impact the complexity at which future maintenance projects are possibly driven. The catalog depends on a study that was conducted in cooperation with the Microsoft Visual C++ team in 2008.

The paper is organized as follows: The second section gives an overview of the basics of software frameworks, clarifies terminology, and discusses the significance of frameworks. It follows a listing of framework key characteristics which represent the starting point for our further explanations. Based on this, the third section introduces an approach for how to improve the maintainability of a framework using the FEPP. Thereby, the focus is upon demonstrating that perspicacious development embodies the foundation for easy-to-accomplish future maintenance.

2. Backgrounds of Framework Engineering

The term "*software framework"* generally refers to a high-level design that is abstract enough to be applicable to various problems of an application field. It can be used to bootstrap concrete implementations which are based on a common architecture [10]. Referring to Taligent Inc., frameworks are defined as "a set of prefabricated software building blocks which programmers can use or customize for specific computing solutions" [15]. Thereby, Taligent emphasizes that a framework captures the problem-solving expertise necessary to solve a particular class of problems. Companies can use frameworks to obtain such problem-solving expertise without having to fully develop it. Furthermore, frameworks provide a well-designed infrastructure, so that when new pieces are created, they can be added easily or substitute old pieces with minimal impact [14].

All of the above definitions share the idea of two basic aspects: Firstly, a framework provides the users with a certain set of functionality and secondly, frameworks allow the users to customize or modify this functionality. Framework reengineering is becoming evidently a vital activity in the software industry [4]. Its goal is to understand, analyze, and improve frameworks to form new framework versions. Reengineering includes tasks, such as refactoring (changes to the appearance of code), redesign (changes to the architecture), and refinement (new functionality, bug-fixing, or performance improvements). Since the reengineering process can be triggered as part of the software lifecycle -especially the maintenance phase-, we will in the remainder of this work also refer to it as "framework maintenance".

In the following, we will further discuss the significance of frameworks and show their different key characteristics.

2.1. Significance of Frameworks

Frameworks support the application development process by providing prefabricated solutions to reoccurring problems. They capture and leverage the expertise of domain experts in a software component that can be included by other application programs. The use of frameworks can result in a dramatically shortened development time with fewer lines of code. This is because several common aspects of the applications are already captured by the framework. The effort required for

maintaining applications can also be significantly reduced when multiple applications are built on top of one framework [8]. In case of modifications or fixes being made to the framework, applications implicitly benefit from the changes, since those are automatically propagated through the framework. The applications that are built from the framework follow the same design and share the same code base; thus frameworks provide consistency and therefore a better integration across platforms.

Above all, using frameworks is related to two major aspects: Reuse and Quality. Thereby, not only the implementation of a system, but also the design of the system is reused. Since the design of successful frameworks has already proven to be efficient and has run through an in-depth testing and refinement process, it forms a quality base for developing new applications. Once a framework has been developed, the problem domain has already been analyzed and a working design and implementation have been produced [1].

2.2. Framework Key Characteristics

With the term "Framework Key Characteristics" (FKC) we refer to properties that are commonly embodied by successful frameworks to a relatively high extent. The FEPP relate to particular FKC by describing ways of how to implement them. Both FEPP and FKC will again be mentioned together, when addressing how they can help reducing maintenance.

The design and the functionality of a framework incorporate the following FKC:

Reusability means that software and ideas are developed once and then used to solve multiple problems. This leads to an enhanced productivity, since applications can now be built on top of already existing solutions for generic problems [11].

Ease-of-Use encompasses the application developer's ability to use the framework [8]. The framework should be easy to understand and facilitate the development of applications. Ease-of-use is also established by providing a detailed documentation including descriptions of the framework's functionality as well as sample applications demonstrating how to solve easy problems using the framework.

Extensibility means that new components or properties can be added easily to the framework. Extension typically is achieved by deriving from existing classes (inheritance) or adding customized components to the framework (composition). So called hook methods provide a way to extend stable interfaces with new functionality. This is important to "ensure timely customization of new application services and features" [6].

Flexibility describes a framework's ability to be used in more than one context. The more problems the framework can be applied to, the higher the problem domain coverage. Very flexible frameworks are reused more often than frameworks with a lower degree of flexibility [11].

Completeness refers to a framework's ability to cover all possible variations of a problem. Since even the best frameworks can never provide solutions to all possible problems with an arbitrary level of detail, it makes it consequentially impossible for frameworks to be complete. However, a certain degree of completeness can be achieved and is referred to as "relative completeness" [8]. Relative completeness encompasses default implementations for the abstractions within a framework, so that these abstractions do not necessarily have to be implemented by the user.

Consistency is a characteristic which reflects that the rules and conventions which determine the framework are followed throughout the whole framework without exception. Consistency in frameworks speeds up the developers' understanding of the framework and helps to reduce errors in its use [11]. Consistent frameworks always follow the same interface conventions and class structures as well as the same notations for naming variables, functions, and classes.

During our investigations, we found out that the MFC incorporate those characteristics to a very high extent. Even though the MFC core was designed more than 17 years ago, it considers the FKC in an exemplary fashion. Ease-of-use, for example, is achieved by providing an elaborated interface, a very comprehensive and always up-to-date documentation, and a complex, but easy-to-use development environment. On the other hand, due to their age, the MFC do not consider certain aspects of the object-oriented programming paradigm, such as *Polymorphism*. An example of why this can be disadvantageous will be given in section three. In conjunction with the problem of maintenance, it will be discussed pros and cons of the way MFC incorporates the FKC.

3. Addressing Framework Maintenance

A study conducted by the National Bureau of Standards estimated that 60% - 85% of the total software development cost is due to maintenance [5]. These numbers are determined mostly by errors that were not found during operational testing and thus needed to be fixed at the customer's side, generally an expensive undertaking. Therefore, it seems reasonable to attempt a reduction of potential future maintenance efforts from the very beginning. Certainly intensive and exhaustive testing prior to the release plays a major role, but maybe as important as that is to initially design the framework in a fashion that makes future engagements easier to accomplish. The incorporation of best practices, such as design patterns or object-oriented methods in general, standards, guidelines, and substantiated documentation can significantly support the creation of better preconditions for future maintenance and extension development.

The catalog we have derived from the MFC case study distinguishes the reduction of framework maintenance into two core activities: *preventing* and *enabling* maintenance. Preventing maintenance refers to an attempt to reduce the likelihood of future maintenance while enabling maintenance encompasses a concept that makes the framework accessible for future maintenance.

3.1. Preventing Maintenance

Preventing maintenance represents a concept to increase the overall quality of a framework while reducing the likelihood of future maintenance. The quality of a framework is determined by the degree to which it incorporates previously mentioned FKC on the one hand and by the amount of errors it contains on the other hand. Even though the number of bugs can be reduced by instantiating methods of testing, the integration of FKC can further reduce the likelihood of erroneous behavior. The following four FEPP describe how to achieve a relatively high saturation of the FKC in a framework that help cutting down on the likelihood of maintenance.

3.1.1. Documentation

The documentation of a framework represents a driver that can make a critical contribution to the overall success of the framework. *Ease-of-use* and *reusability* can be established by providing a detailed documentation including descriptions of functionality as well as sample applications demonstrating how to solve easy problems with the framework. Without documentation, the only

way for application developers to understand how the framework is used would lie in trying to comprehend the way the framework is used from the source code. However, if there was not even the source code available, the value of the framework to the framework applicants would be very low. With the Microsoft Developer Network (MSDN)[2], Microsoft provides a comprehensive and highly up-to-date documentation for the MFC that is always accessible over the internet.

3.1.2. Contracts

One of the key problems when working with frameworks, such as MFC, is that, provided the inputs to a function or component, the framework users do often not receive the output they expected. In other words: the perceived behavior of a function might sometimes differ in some way from the behavior that the users anticipated. Due to this misunderstanding, users often open bug fixing enquiries on the vendors' side, whereby the vendors have two possibilities of dealing with them: Either they modify the way in which the respective functionality is exposed by the framework or they refine the documentation and clarify how to correctly use this part of the framework.

To avoid this misunderstanding between component designers and component users from the very beginning, contracts provide a way of determining beforehand, whether a class or a component used within a certain context generates a correct result [2]. They "specify preconditions on participants to establish the contract and the methods required for the instantiation of the contract" [12]. While adding a clear communication between the framework users and the framework designers, contracts can help reducing the frequency at which users open bug fixing enquiries that aim at clarifying the way hooks[3] are used. Contracts can help to enhance the ease-of-use and thus the reusability FKC of a framework and allow a slimmer documentation. Furthermore, they can improve the flexibility of the framework architecture making maintenance projects easier to conduct.

Regarding MFC, we could identify only a weak contractual behavior. Behavioral contracts, for example, are incorporated by the macros VERIFY, ENSURE, and ASSERT. In case of an error, the MFC application terminates with a runtime exception, whereby a dialog states the problem that caused the shutdown. Even though this kind of contractual behavior seems to comply with the idea of behavioral contracts, it is only very inchoate and can be easily circumvented. MFC's exception handling mechanisms can trivially be bypassed by just overriding and reimplementing the function that wraps a macro accordingly. To split the contract of the base class, it is enough to just not implement previously mentioned macros in the overridden function.

3.1.3. Standard Conformity

Standard conformity incorporates the *flexibility* and *reusability* characteristics of frameworks. Furthermore, standards can help improving the *consistency* of the overall framework architecture and code. Incorporating standards not only improves the product quality (standards are geared to principles and best practices), but also can help improving the flexibility of the framework.

Due to specific aspects of the Windows operating system, Microsoft had to rely on an extended C/C++ standard in its compiler. Standards allow code to become independent from its base technology. Standardized code should run on every platform that complies with the standard and enables software developers to use their products with a variety of base technology distributions of multiple vendors.

---

[2] http://www.msdn.microsoft.com

[3] Hooks are understood here as the means to perform customization to a framework.

Since the MFC exclusively targets the Microsoft Windows platform, it stands to reason that MFC does not support the development of platform-independent code. Other compilers on the Windows platform do not explicitly support the MFC; a fact that binds the MFC developer community to the Microsoft compiler. In case of a specific compiler becoming superior to the Microsoft compiler by, for example generating particularly high-performing executables, MFC applications could not benefit from this. Furthermore, bugs within the Microsoft compiler could not be by-passed by simply switching to another compiler that does not show these bugs.

An alternative solution to circumvent this problem could encompass concentrating all MFC-related client code in one module. This module can then be compiled with the Microsoft compiler while the MFC-independent rest of the code is passed to a compiler that can achieve better performance. However, from the application developers' perspective, an MFC that works without any Microsoft- specific extensions would certainly be favored as it would increase their flexibility.

3.1.4. Default Behavior

One important question concerning the *completeness* of a framework is how default behavior is incorporated and exposed. If the users, for example, do not need or want to customize certain abstractions within a framework, a default implementation that fits their particular requirements might save them time and effort. On the other hand, if a default implementation of an abstraction should not be sufficient, they could simply override it and provide their own customized functionality.

Even though providing default behavior on the first glance increases the size of the framework code base, this does not automatically result in a potentially increased maintenance complexity. Since an enhanced completeness is connected to an enhanced *ease-of-use*, the framework users will potentially open less support enquiries that aim at refining hot spots[4] in the framework. In many cases, the aggregated effort for maintaining the code that adds the default behavior might be less than the effort that evolves from refining badly designed hot spots.

Regarding the MFC framework, default behavior is incorporated in classes that are intended to be used as base classes (e.g. CObject) as well as in the hook methods that represent the set of Windows message handlers. Instead of leaving the implementation of the message handlers to the developers of the application extensions, the MFC provides a default implementation for every method.

3.1.5. Summary

As can be seen in the following table 1, each individual FEPP incorporates different FKC.

FKC/ FEPP	Documentation	Contracts	Standard Conformity	Default Behavior
Reusability	x	x	x	
Ease-of-Use	x	x		x
Extensibility				
Flexitbilty		x	x	
Consistency			x	
Completeness				x

Table1: Framework Key Characteristics (FKC) vs. Framework Engineering Principles and Practices (FEPP)

[4] Hot Spots are places within the framework that require customization from the user.

3.2. Enabling Maintenance

Even though the previously discussed methods help reducing the likeliness of future maintenance, this does not mean that the product does not need to be maintained at all. Changes in the domain of the framework, extensions requested by the user, or even bugs represent reasons to make modifications to the framework. Thus, it seems even more important to establish the framework on an architecture that is open and flexible enough to efficiently support future maintenance, use appropriate design patterns, and establish and follow conventions.

3.2.1. Architecture

Inside the architecture, we want to point at two concepts that effectively enable the user to customize the framework's behavior: *Message Mapping*[5] vs. the use of *Polymorphism*. Message Mapping describes the way MFC binds Windows messages (integer IDs) to Windows message handlers (typically classes), whereby the event handlers announce which events they are able to process. The basic idea of Message Mapping is that the users of MFC in their client applications are able to customize the message handling to hook in their own responses on events.

As MFC evolved as a child of the established programming standard C++, it could benefit from object-oriented language concepts. One powerful feature of C++, however, was not utilized: Polymorphism. The concept of Polymorphism captures the potential to more intuitively perform what was implemented with Message Mapping in MFC. Modifying the MFC to make use of Polymorphism could encompass to automatically attach a certain virtual message handler function to a specific Windows message in the MFC message pump (figure 1). Changing the message handler of a Windows message could now be achieved by deriving a new class from the MFC's Windows message processing class (CWinApp) and overriding the virtual message handlers in the new class respectively.

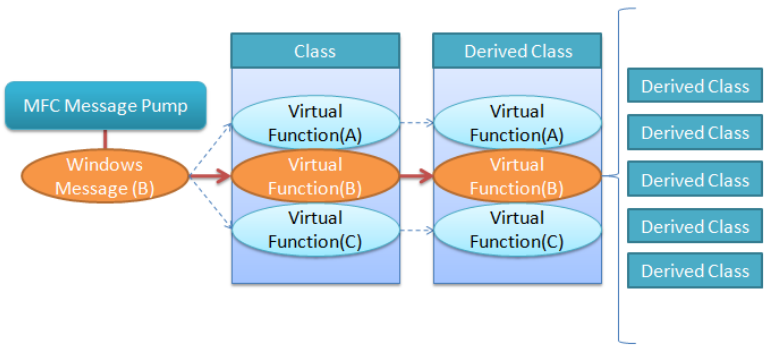

Figure 1: A polymorphism-based alternative to Message Mapping.

Following conclusions are drawn: While Message Mapping is connected to manually editing the Windows message/Windows message handler entries in the map and then defining the message handler in a class accordingly, the Polymorphism-based alternative makes the first step unnecessary and merely requires overriding the appropriate virtual functions. Internally, however, the Polymorphism-based alternative does something similar to the Message Mapping mechanism, but is rather exposed as an object-oriented feature of the C++ programming language itself.

---

[5] For more detail on Message Mapping, please refer to the MSDN (http://www.msdn.microsoft.com).

Since there are no message map entries to be taken care of, the Polymorphism-based approach would not only improve the *ease-of-use* FKC of the MFC, but also make the code of the application less susceptible to errors. The less code needs to be written by a developer, the fewer the number of errors he can potentially commit to the application. With respect to the effort that is related to maintaining and extending the MFC, the last statement also indicates that the Polymorphism-based approach could result in a decrease of resources necessary for maintenance (as there is less code affected). Hooking a new Windows message handler into an application could be achieved by simply overriding a virtual function from the MFC's message dispatcher class CWinApp rather than fist declaring and defining a new entry in the message map and secondly creating the message handler class.

3.2.2. Design Patterns

Design Patterns provide solutions to reoccurring software design problems that have proven to work in practice [9]. With respect to frameworks, Design Patterns are particularly useful for designing Hot Spots, the parts of the framework that determine its flexibility. Design Patterns can support the development of software frameworks by improving the *flexibility* and *enhancability* of a framework's Hot Spots and thus ultimately paves the way for less complicated modifications due to maintenance.

During our investigations, we identified three design pattern in the MFC: The *Singleton Pattern* (by accessing the main application object CWinApp), the *Bridge Pattern* (in serialization), and the *Observer Pattern* (as the basis of the Document/View architecture). For the ongoing explanations, we will concentrate on MFC's implementation of the Observer Pattern only.

The Observer Design Pattern refers to a pattern that is used to observe the state of an object in a program and is mainly used for realizing distributed event handling mechanisms [9]. The essence of this pattern is that one or more objects (observers) are registered to observe an event that may be raised by the observed subject. The subject provides an interface for attaching and detaching observers as well as notifying all observers that were attached about a new event. The observers contrarily define a notification function that is called by the subject as soon as a new event occurs.

MFC embodies the Observer Pattern in its Document/View architecture. Documents in MFC are usually used to store the application's data and thus act as subjects. Views on the other hand are attached to windows and display the data within documents on the screen while acting as observers. Since a document can have many views to display its data in different ways, a document updates all attached views when its content was changed by calling the function UpdateAllViews. The Observer Pattern in MFC's Document/View architecture (see figure 2) ensures that the modification of a document's data is propagated to all views that were attached to this document.

3.2.3. Conventions

Conventions can help developers to better understand code that was written by other persons and thus try to address problems that evolve from the nature of very large projects. In order to be able to handle even big software development projects, they are broken down into smaller ones and assigned to teams. Code that is written in a more natural way with a structure that easier maps to a human's native language can consequentially be easier to comprehend and understand [3]. To enable developers to use the associations and experiences they have gained from past projects, it is essential that all code produced within a software project (or even in general) follows a similar design. This design aims at providing a coding foundation for all developers and thus tries to

converge their work-related way of thinking expressed in so called coding conventions. Thereby, coding conventions not only suggest writing short and easy-to-understand statements, but also advices to use expressive variable and function names [13].

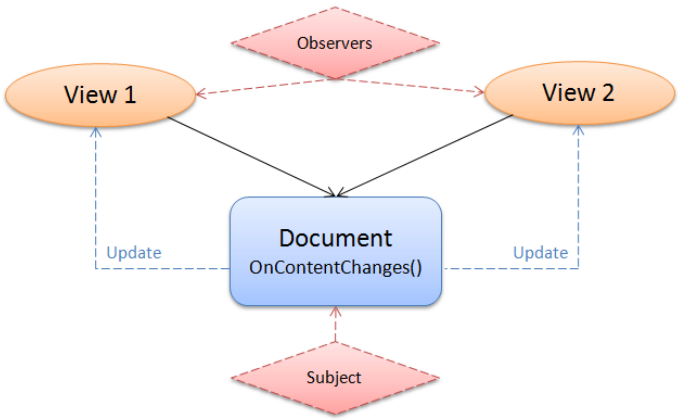

Figure 2: The Observer Pattern in MFC's Document/View architecture.

Across the Windows API, the foundation for the MFC, we identified the use of the Hungarian notation. In the Hungarian notation the first character or first several characters of a variable or parameter name identify the type of this variable or parameter. In addition, the MFC also defines its own set of naming conventions (e.g. "m_" for member variables or "On" for event handlers).

With respect to maintenance and extension development of MFC, the used naming conventions certainly help developers to quickly acquaint themselves with existing code. They thus address the *reusability* and flexibility FKC, but from a developer's point of view.

4. Conclusion

This work aimed at presenting an approach that describes what to consider when reengineering frameworks with respect to their maintainability. As shown in the example of the MFC, frameworks might be subject to reactivation, even though their deprecation was already decided earlier. Thereby, the decision about reactivating a framework might be the result of an abruptly changing market or an organization's strategy.

The approach we presented identifies two main aspects that should be taken into account, when analyzing a framework's internal condition after a reactivation: Preventing maintenance and enabling maintenance. We have shown that preventing maintenance can be achieved through a detailed documentation, the incorporation of contracts and default behavior as well as the consideration of standards. Enabling maintenance on the other hand helps reducing the complexity of future maintenance by designing the framework architecture in a way that it is easy to access by incorporating design patterns at places where it makes sense. Furthermore, conventions can support maintenance by allowing associations that were made earlier in the development phase.

During our investigations, we identified several areas within the MFC that urgently require action to match this framework with the state-of-the-art in framework engineering. Refining the MFC's internals at discussed places might help cutting down on the complexity of future maintenance projects. However, a careful investigation on the diversity of the MFC applicants would most likely reveal that additional constraints exist which would increase the complicacy of conducting those

changes. Further research could concentrate on the engineering of an approach that describes how to perform fixes to reactivated frameworks considering the situation of the framework community. Another open point might be the use of tools, such as Jfreedom [7], to support the framework reengineering process.

5. Bibliography

[1] ARRANGO, G., PIETRO-DIAZ, G. and PIETRO-DIAZ, R., Domain Analysis Concepts and Research Directions, IEEE Computer Society 1991

[2] BEUGNARD, A, JÉZÉQUEL, J. M., PLOUZEN, N. and WATKINS, D., Making Components Contract Aware, IEEE Computer Society 1999

[3] CWALINA, K., ABRAMS, B. and RAGSDALE, S., Framework Design Guidelines: Conventions, Idioms and Patterns for Reusable .NET Libraries, Amsterdam, Addison-Wesley Longman 2005

[4] DEMEYER, S., MENS, K., WUYTS, R., GUEHENEUC, Y. G., ZAIDMAN, A., WALKINSHAW, N., AGUIAR, A. and DUCASSE, S, Workshop on Object-Oriented Reengineering, 19th European Conference on Object-Oriented Programming (ECOOP) 2005

[5] EAGLE, D., Evaluating Larch/C++ as a Specification Language: A Case Study Using the Microsoft Foundation Class Library, Iowa Sate University, Department of Computer Science, Iowa, USA 1995

[6] FAYAD, M. E. and SCHMIDT, D. C., Application Frameworks, Communications of the ACM, vol. 40, no. 10, pp. 32-38 1997

[7] FLORES, N. and AGUIAR, A., Jfreedom: a reverse engineering tool to recover framework design, Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP), Workshop on Object-Oriented Reengineering 2005

[8] FROEHLICH, G., HOOVER, H., LIU, L. and SORENSON, P., Designing Object-Oriented Frameworks, in: CRC Handbook of Object Technology, CRC Press, pp. 25-1 - 25-22 1998

[9] GAMMA, E., HELM, R., JOHNSON, R. and VLISSIDES, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman, Amsterdam 1995

[10] GREENFIELD, J. and SHORT, K., Software factories: assembling applications with patterns, models, frameworks and tools, Wiley Publishing, Inc., Indianapolis, USA 2004

[11] KOSKIMIES, K. and MOSSENBACK, H., Designing a Framework by Stepwise Generalization. Proceedings of the 5th European Software Engineering Conference 1995

[12] LAJOIE, R. and KELLER, R. K., Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert, Proceedings of the 62nd Congress of the Association Canadienne Francaise pour l'Avancement des Sciences (ACFAS), Colloquium on Object Orientation in Databases and Software Engineering, Montreal, Canada, pp. 94-105 1994

[13] MOSER, H., Auswirkungen von Code Conventions auf Software Wartung und Evolution 2003

[14] NELSON, C., A Forum for Fitting the Task, IEEE Computer 27, pp. 104-109 1994

[15] TALIGENT, The Power of Frameworks, Addison Wesley 1995