MICROSOFT CORPORATION (REDMOND, USA)
OTTO-VON-GUERICKE UNIVERSITY (MAGDEBURG, GERMANY)

Maintaining Object-oriented Component Frameworks

A Case Study of the MFC

Robert Neumann
(robert.neumann@densetsu.org)

Bachelor Thesis
May 9th, 2008

M. Sc. Ayman Shoukry (Microsoft Visual C++)
Prof. Dr.-Ing. habil. Georg Paul (University of Magdeburg)
Dipl. -Inform. Niko Zenker (University of Magdeburg)

Acknowledgement

To the loving one, Ksenia, with whose pure patience I endured this endeavor with great joy.

My gratitude is directed to Ayman Shoukry (MS Visual C++) who supervised my investigations, Alessandro Contenti (MS Visual C++) who always had an answer to all of my questions, Damien Watkins (MS Visual C++) who gave me valuable hints regarding my proceedings as well as the whole Visual C++ team.

Thanks to Professor Dr. Georg Paul who coordinated my activities from abroad and always had time for any of my concerns.

An expression of gratefulness to Niko Zenker and Sebastian Günther who have given me the necessary academic support and whose valuable feedback is to a high extend incorporated in this work.

Finally, big thanks to Herb Sutter (Chairman of the ISO C++ Standard Committee) and Jeffrey Richter (Wintellect) for having provided me with free copies of their books. Herb's book in particular helped me gaining a better understanding of standard C++. Jeffrey's book served as a foundation for my understanding of the Windows API.

May, 2008 *Robert Neumann*

Table of contents

Table of Abbreviations

AC	Accumulated Maintenance Cost
Afx	Application Framework Extensions
API	Application Programming Interface
CL	Microsoft's C/ C++ compiler
CLI	Common Language Infrastructure
DLL	Dynamic Link Library
ICC	Intel's C/C++ compiler
ID	Identifier
IDE	Integrated Development Environment
IDL	Interface Definition Language
ISV	Independent Software Vendor
G++	GNU's C/C++ compiler
GoF	Group of Four
GUI	Graphical User Interface
LIB	Static Link Library
MDI	Multiple Document Interface
MFC	Microsoft Foundation Classes
MSDN	Microsoft Developer Network
SDI	Single Document Interface
STL	Standard Template Library
VC	Visual C++
V-Table	Virtual Function Table
WinAPI	Windows Application Programming Interface

1. Overview

1.1. Motivation

When Bill Gates for the first time announced the *.Net* framework in the year 2000, he certainly envisioned the future software development world being quickly conquered by his latest invention. Easier development of managed code that is executed in a runtime environment, a massive set of new features, platform independency, and even the brand new programming language C# that in particular addressed the needs of Microsoft's managed initiatives promised to become a big success (Hejlsberg, Wiltamuth, & Golde, 2006).

With the introduction of version 1.0 of the *.Net* framework in 2002, not only Microsoft's C/C++ engagements got under pressure, but also other languages. In order to move their focus away from native towards managed code to eventually better support the newly established *.Net* framework, the Visual C++ organization (VC) had to bring up considerable investments. Integrating the *.Net* framework into the language syntax of native C/C++, the result of VC's efforts, the C++/CLI[1], provided an enhanced interoperability between native and managed code.

In the course of that, Microsoft's largest native C++ framework for development on the Windows platform, the Microsoft Foundation Classes (MFC), got subsequently more and more out of scope since its managed brother was considered as its future successor. One important fact, however, was overseen during this period: 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 for sure were dependent on the *MFC* being updated to expose new Windows features as they wanted to continuously support their products and keep them attractive, even on newer versions of Microsoft's operating system. Migrating their code bases towards *.Net* would have required an investment in time and money that did not relate in any way to the return they would have gained; the functionality would eventually still remain the same.

After ten years of not performing any major investments into the MFC, the ISVs ask for updates that would enable them to provide their applications with a modern look and feel and make use of up-to-date technology. With Visual Studio 2008 the MFC for the first time for more than 10 years experienced a major set of updates. Without performing any code work, but by simply recompiling their applications, MFC customers can add the new Vista look and feel to their applications. Another big investment was brought up with the acquisition of BCG Soft's graphical user interface library. After this library was integrated into MFC, MFC customers are now able to create applications that have a fully customizable Office 2007 look and feel, including the Fluent UI. Those and future investments will be a determining factor in answering the question whether or not the MFC can continue its past success.

[1] CLI stands for Common Language Infrastructure

1.2. Goal of this work

With the resurrection of the MFC in 2007 and further updates planned in future, a reassessment of its architecture and internal structures has become necessary. The goal of this work is to provide answers to the following questions:

1. What are frameworks and how do they differ from normal applications? **- Chapter 2**
2. How are frameworks developed and which framework engineering principles and practices are available? **- Chapter 3**
3. To which extent does the MFC incorporate framework engineering principles and practices? **- Chapter 4**
4. How can the framework engineering principles and practices help reducing maintenance and extension development cost? 5. Which alternatives exist to the current implementations and what value could they add to the MFC and its customers? 6. How up-to-date is the MFC and which investments are necessary in order to continue its success? 7. How practicable is the introduction of major changes and what effects would they trigger? **- Chapter 5**

Prior to the maintainability study of the MFC, a comprehensive literature research was performed. Thereby, in particular publications and books about software framework engineering have helped to establish an understanding for the necessary basics. Results from the literature research are presented in chapter two and three in condensed form, when the concept of frameworks is introduced and a selection of framework development techniques explained.

Conversations with team members of the Visual C++ unit, source code reviews of the MFC as well as additional Microsoft literature were used to extract information about the architecture and mechanisms implemented in MFC. In combination with the knowledge that was gained in chapter two and three, an overview of the MFC was crafted including a first assessment of how the MFC in general complies with principles and practices of the framework engineering discipline.

Regarding the actual topic of this work, the MFC's internal architecture and structure was analyzed and the appropriateness of certain design structures discussed. Additionally, suggestions for how to realize alternatives that could potentially address weak spots and areas of improvement were stated. Finally, conclusions with respect to the complexity of future maintainability and extension projects were summarized in chapter five.

Concerning the question which scientific proceeding was instantiated, due to the rational nature of this work rationalism was favored to empiricism. Thereby, conclusions about the research object were made based on knowledge extracted from several already existing publications (see bibliography section). An empirical approach did not seem to be feasible as enough material was available to conduct this study and the influence of random variables was negotiable. Hence, the results of this work were not accomplished by observation of the research object, but by degeneralization and logical implication from an existing knowledge base.

1.3. Chapter overview

From an engineering point of view, this work aims at investigating to which extend modern principles and practices of the software and framework engineering domain were incorporated and which impact those aspects will have on upcoming investments into the MFC. To support the understanding of this maintainability study, chapter two and three provide an overview of the state-of-the-art framework engineering principles and practices. Chapter two in particular introduces to the terminological world of frameworks, states their characteristics, and discusses which value frameworks can add to software development projects. Chapter three provides an overview of the framework lifecycle, discusses special aspects of the design and requirements gathering, and illustrates general as well as specific software development techniques. After the foundation for this maintainability study was constructed, chapter four explains the scientific procedure and methodology that was instantiated to conduct the study. Chapter five follows with an introduction to the MFC and an application of the in chapter two and three outlined principles and practices to the MFC framework. With chapter six representing the main part of this work, several peculiarities of the MFC are introduced, it is discussed how well those peculiarities support future maintenance and extension development efforts, and weak spots and possible problems areas that could eventually become of matter are revealed. This work closes with a conclusion derived from the conducted investigations and a discussion of future prospects of the MFC in chapter seven.

2. Fundamentals of software frameworks

This chapter introduces basic ideas and concepts of software frameworks. The covered clarification of terms and principles will serve as a foundation for the understanding of later chapters. Starting with the definition of the term framework and followed by an introduction to concepts and characteristics of frameworks, it will next be shown how frameworks are classified and what a framework domain is. Furthermore, the framework lifecycle will be introduced and it will be shown how roles can be assigned to the various framework users. The chapter will close with a framework's related benefits and difficulties being opposed to each other as well as an economic evaluation of the value frameworks can add to the software development process.

2.1. Defining frameworks

The term "*framework*" reflects a basic conceptual structure that solves or addresses certain complex issues. At the same time, the conceptual structure consists of a sequential set of directions which lead to a solution of the problem. Furthermore, the conceptual structure must be generic enough to be applied to various problems that are possibly spread throughout different problem domains. Problem domains, thereby, incorporate an abstract set of problems that is general enough to unify certain problems within one group (Gangopadhyay & Mitra, 1995). Figure 1 depicts how a sequence of specific directions out of a set of directions together with a specific problem out of a problem domain produces a solution.

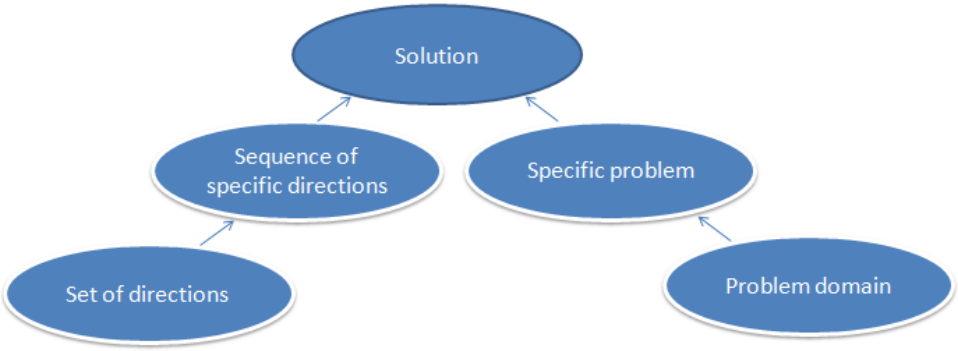

Figure 1: Specific directions applied to a specific problem result in a solution.

The following more concrete example will help to add understanding to the concept of frameworks by illustrating above explanations. In our case, the problem situation is described by a student who is hungry, but does not know how to cook. Therefore, the student utilizes a cookbook which, with respect to its characteristic of being a framework, addresses certain complex issues. To be more precise: in our case the cookbook represents a collection of directions of how to cook meals. When those directions are followed -means the ingredients are combined in the correct manner- they will lead to the prescribed result, in our example of the hungry student a ready meal. Furthermore, the cookbook fulfills the framework property to be general enough to be applied to various cooking

scenarios as it contains several directions, namely recipes, of how to cook meals. Finally, the problem domain the cookbook addresses can be determined as *"How to cook?"* with the cookbook containing solutions, in this case referred to as recipes, to various problems among the *"How to cook?"* problem domain. Thus, a cookbook can be considered as a framework for hobby cooks or hungry students who want to make use of existing proposals for how to craft a tasty meal.

Since the remainder of this work concentrates on software frameworks as well as on the development and maintenance of those, it is necessary to next condense the general definition of the term "framework" and concretize a bit more the term "software framework". "A framework is a skeleton of an application which can be customized by application developers" (Lücke, 2005). Taligent Inc. furthermore defines a framework as "a set of prefabricated software building blocks which programmers can use or customize for specific computing solutions" (Taligent, 1995). Thereby, Taligent Inc. emphasizes that a framework captures the programming expertise necessary to solve a particular class of problems. Companies can purchase or license frameworks to obtain such problem-solving expertise without having to develop it independently. Hence, frameworks provide developers with solutions for problem domains and help them to better maintain those solutions. On top of that, frameworks provide a well-designed and thought out infrastructure so that when new pieces are created, they can be substituted with minimal impact on the other pieces in the framework (Nelson, 1994). Figure 2 illustrates this with B being substituted by B' in the framework. The application A, B, and C that set on top of the framework automatically and transparent for the applications themselves incorporate the substitution of B by B'.

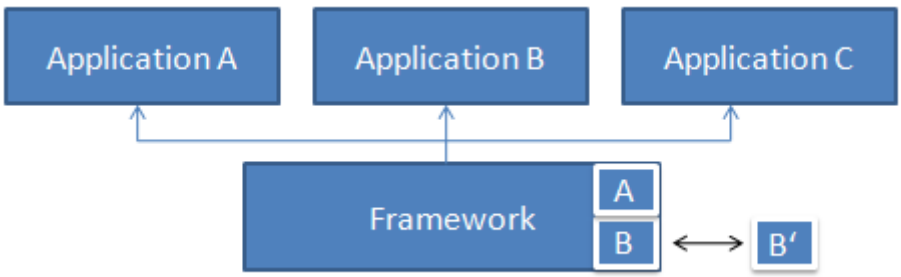

Figure 2: Part B is substituted by a new B' and automatically propagated through the framework to the application extensions.

Lücke defines a framework as "a reusable design that is described by a set of abstract classes as well as the interaction of instances of these classes" (Lücke, 2005). For more detail on abstract classes, please refer to the following excurse.

Excurse - Abstract Classes

Abstract classes are classes that declare a set of virtual methods, but do not provide an implementation, means a definition, for these methods. As a result, abstract classes cannot be instantiated through an object, but are rather used as interface classes to objects of subclasses. These subclasses usually define implementations for the base classes' virtual methods and thus become concrete classes that are instantiable by a constructor. On the other hand, subclasses can again be abstract by either not providing implementations for all virtual methods of the base classes or by defining new virtual methods on their own.

Even though the above definitions attempt to characterize the concept of software frameworks by highlighting the key elements and properties of such, they cannot provide an absolutely clear understanding of the concept of software frameworks. Though, all of them share the idea of two basic aspects: First, a framework provides the users with a certain set of functionality and secondly, frameworks allow the users to customize or modify this functionality. However, since frameworks, due to their very nature, try to abstract from a specific problem with a specific solution towards a more generic solution for various specific problems, it is common sense that there can only be an attempt to find a common understanding of how to consider software frameworks rather than a concrete definition (Froehlich, Hoover, Liu, & Sorenson, 1997).

Quick Review - Defining frameworks

This section has shown that, even though there are some attempts, there can hardly be a concrete definition of the term "framework". However, all definitions shown in this section share the following two aspects: First, frameworks provide users with functionality and secondly frameworks allow the user to customize this functionality.

2.2. Framework concepts

To give a more picturesque introduction to this section about framework concepts, the following analogy was chosen. When architects plan houses, they need to know about which parts the house will be made of and which materials are used. With this information, the architects can determine an appropriate construct for the house that addresses the designer's concept and concurrently serves the architectural needs. Thereby, architects identify parts of the design that strain certain parts of the house's statics and might result in a not solid construction with possible threats for the safety of future residents. Similar to architects who need to know about how to assemble a house from bricks, framework engineers also require knowledge about general concepts of building frameworks. Understanding general architectural aspects of frameworks helps establishing more stable and flexible products. In the following the architecture of software frameworks will be analyzed.

Frameworks, in general, consist of two different parts: The *framework core* and the *framework library*. The **framework core** not only serves as a basis for all interaction between the framework or later applications built with the framework respectively, it also defines the generic structure and behavior of the framework (Sparks, Benner, & Faris, 1996). The framework core usually encompasses an ensemble of abstract classes creating an interfacing methodology with applications or with other framework-intern concrete specializations of these abstract classes. An example could be a file system with one abstract class defining the general interface to several file system implementations. While every concrete file system implementation has its own internal representation of data and ways of handling it, the read and write part of each file system always remains the same to the client. The abstract interface introduces a layer of transparency to the client that treats all file system implementations as they were the same and thus hiding internal implementation details. Apart from that, the framework core can also contain public concrete classes that are meant to be used as every normal class or in combination with other parts of the framework. A graphics library, for example, could contain concrete representations of geometric objects, such as spheres or cubes that are first rotated, then translated, and finally scaled before being projected to a surface.

The **framework library,** on the other hand, consists of extensions to the framework core with concrete components that can be used directly by applications developed from the framework (Froehlich, Hoover, Liu, & Sorenson, 1997). In most cases, only a certain set of framework library functionality is used within an application. However, the framework library is the part of the framework that addresses the criteria of completion, which will be introduced later, by adding concrete and immediately usable functionality to the mostly abstract core. Therefore, the framework library generally provides solutions for more scenarios than a single application can cover. Figure 3 depicts the two parts of a framework that were introduced in this paragraph.

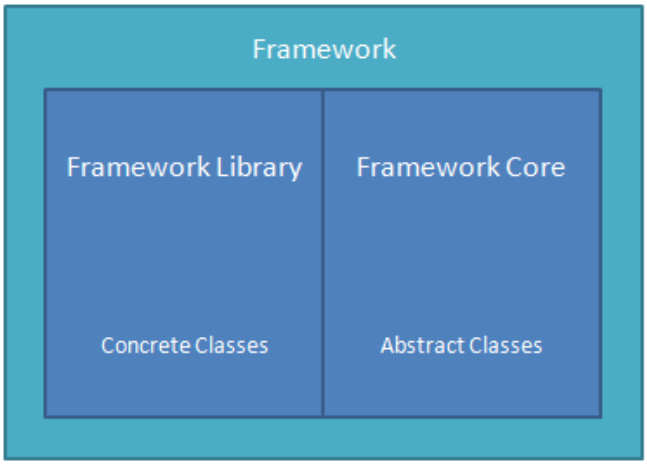

Figure 3: A framework consists of a core consisting of abstract classes and a library consisting of concrete classes.

From a framework engineering perspective, the whole application is the aggregate of the framework core, the used framework library functionality, and any application specific extension made (Froehlich, Hoover, Liu, & Sorenson, 1997). Thereby, the application extension is the additional functional part that the developer has to add on top of the framework, so that the application can perform its specific task. The following graphic illustrates how framework core, framework library, and application extension form the whole application.

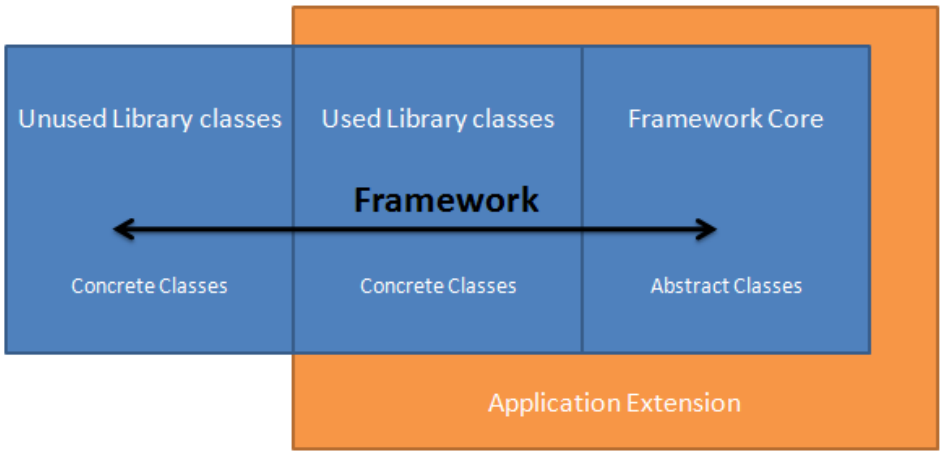

Figure 4: A complete application consists of the framework core, the used library classes, as well as the application extension.

Quick Review - Framework concepts

In this section the two parts of a framework, namely core and library, were introduced. Furthermore, it was shown that the application-specific part is assembled in the application extension.

2.3. Characteristics of frameworks

When software developers started using frameworks, they did this because of several beneficial characteristics they incorporate. Application engineers noticed that those characteristics help raising the productivity of their everyday work. Instead of troubling with reoccurring problems, such as memory management or file access, they were able to concentrate on the concrete task their application had to perform. This section highlights common characteristics of software frameworks and how these characteristics contribute to an improved software development experience.

As shown in section 2.1 *Defining Frameworks*, the term “software frameworks” generally refers to a high-level design that is abstract enough to be applicable to various problems of an application field. On top of that, frameworks offer functionality in form of already partially implemented classes. Both the design and the functionality of a framework incorporate the following key aspects:

Reusability means that software and ideas are developed once and then used to solve multiple problems. Using and reusing frameworks typically leads to an enhanced productivity since applications can now be built on top of already existing solutions for generic problems (Koskimies & Mossenback, 1995). Furthermore, frameworks have usually passed numerous iterative phases of refinement and testing, so that software that is developed from a framework typically shows an increased reliability and quality. The here introduced **reusability** aspect refers to the code as well as the design of the framework, since application extensions not only make use of the already existing code, but are also bound to the general design concept of the framework .

Ease of Use encompasses the application developer's ability to use the framework (Froehlich, Hoover, Liu, & Sorenson, 1997). The framework should be easy to understand and facilitate the development of applications. In general, badly designed frameworks will not be used, if it is hard for the user to understand them and so will fail to meet considerable utilization. In order to develop easy-to-use frameworks, the interactions between the application extension and the framework should be simple and consistent. Hooks (covered later in this text), for example, should be small, simple, and easy to place and access. Furthermore, ease of use is 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. In order to be truly useful, frameworks not only have to be easy to use, but also easy to extend. Extension typically is achieved by deriving from existing classes (inheritance) or adding customized components to the framework (composition).

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. With a higher problem domain coverage, means the framework provides more solutions to problems within one single problem domain, also the framework's flexibility increases. If frameworks can be applied across multiple domains, their flexibility is especially high. Very flexible frameworks are reused more often than frameworks with a lower high degree of flexibility. On the other hand, very flexible frameworks are likely to have a lower ease of use since flexibility mostly is achieved through abstraction. More abstraction is generally connected to more concrete functionality that needs to be added by the user before the framework can perform its task. Thus, framework design always raises the question of how to find the balance between flexibility and ease of use. Finding this balance can become a tough engineering activity, but is important since neither absolutely inflexible nor very hard-to-use frameworks will appeal to future users (Koskimies & Mossenback, 1995).

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 arbitrary details, 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" (Froehlich, Hoover, Liu, & Sorenson, 1997). 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. If the user wishes to hook in specific functionality for an abstraction, he can do so by overriding the default implementations. On the other hand, if a user does not want to trouble with implementing abstractions that are not meaningful to the solution of his particular problem, then the default implementations for the abstractions might be sufficient. Thus, by providing default implementations, the framework becomes usable without the user performing a lot of customization, but still remains flexible enough to allow the user to do so. In addition to the framework core, the framework library can provide implementations for common operations making the framework easier to use as well as more complete (Froehlich, Hoover, Liu, & Sorenson, 1997).

Consistency is a characteristic that 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 (Koskimies & Mossenback, 1995). Consistent frameworks always follow the same interface conventions and class structures as well as the same concepts for naming variables, functions, and classes.

In addition to the characteristics that were detailed by Froehlich et al., Lücke identifies another three basic characteristics of software frameworks: *Reuse of Analysis*, *Hot Spots and Hook Methods*, and *Inversion of Control* (Lücke, 2005).

Reuse of Analysis: Before applications are developed, a detailed description of the problems that the application tries to solve needs to be identified. This process is referred to as *Application Domain Analysis*. An application domain for financial software, for example, might include portfolio evaluation, an automatic transaction mechanism for stocks, as well as a forecasting model. The results of the application domain analysis include the width of the application domain as well as the application domain's stability. Software that serves the needs of a wide application domain is applicable to accordingly more problems compared with software that covers only a narrow application domain. If the second result of the *Application Domain Analysis*, the stability aspect, turned out to be low, then sudden changes within the application domain itself are likely. In this case, software with a narrow application domain is more likely not to apply to the application domain anymore as compared to software with a broader application domain. So, when application developers perform feasibility studies of frameworks they consider to use, they implicitly make use of the software frameworks' domain analysis. They will decide for the framework that turns out to best fit the targeted problem domain and provides an appropriate degree of stability towards changes.

Hot Spots and Hook Methods: Another characteristic that is typical for frameworks are "Hot Spots" and "Hook Methods" that represent the customization aspect of the framework. Hot Spots, sometimes also referred to as *Hinges*, are the particular places in a framework that need to be or can be customized by the framework users. From a developer's perspective, Hot Spots also represent the areas within a framework where placing hooks for the user is beneficial. A networking framework, for

example, demonstrates that one Hot Spot can assemble many hooks. In this case, the Hot Spot concentrates on the remote connection management of the networking engine. A user could register a callback function with the framework to notify his application, when another computer tries to open a remote connection. Additionally, the developer might want to specify a hook method that is called, once a remote computer left the network session and closed the connection. A third hook could be installed to notify the application of a refused client due to a limitation in the number of parallel open connections. Hook Methods are understood as the means to perform this customization and mostly encompass virtual functions on a class that need to be or can be overridden by the user in order to fit the framework into the desired behavior. In contrast to Hot Spots, so called Frozen Spots capture the common parts across applications. Frozen Spots are fully implemented by the framework and typically have no hooks associated with them.

Inversion of Control describes the property of software frameworks to not reveal the flow of control to the application extension. Frameworks rather keep the flow of control internally while user code or class libraries are called by the framework itself. This is an important demarcation criterion between software frameworks and class libraries: In contrast to software frameworks, code that is assembled in classes as part of a class library is executed by the calling environment. Frameworks on the other hand rather represent those calling environments (Froehlich, Hoover, Liu, & Sorenson, 1997). Figure 5 summarizes above explained differentiation into a framework's and a library's flow of control.

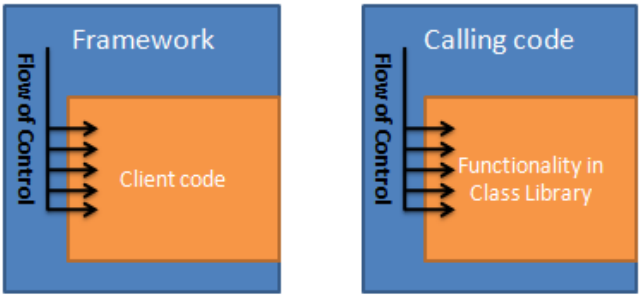

Figure 5: Left a framework's flow of control, right a class library's flow of control: The figure depicts that a framework calls functionality from the user code, and thus maintains the flow of control, whereas the functionality of a class library is called by the user's code and thus does not have any influence on the flow of control.

Quick Review - Characteristics of frameworks

This section gave an overview over the framework characteristics *reusability*, *ease of use*, *extensibility*, *flexibility*, *completeness*, *consistency*, *reuse of analysis*, *Hot Spots and Hook Methods*, as well as *inversion of control*.

2.4. Classification of frameworks

As already indicated in section 2.1 and 2.4, it is essential for frameworks to be structured general enough to be applicable to a large domain of problems. In order to achieve high flexibility and thus high domain coverage, the functionality the framework provides is kept as abstract as possible. The more abstract a framework is, the higher its flexibility. On the other hand, higher flexibility usually affects the completeness aspect negatively (see previous section). This section opposes the flexibility and completeness aspects in the means of methods to customize frameworks by discriminating them into two basic classes.

As proposed by Adair, frameworks are distinguished into *architectural-driven* and *data-driven* frameworks (Adair, 1995).

Architectural-driven frameworks, also referred to as "Whitebox frameworks", are considered to be still in the initial phase and thus not yet stable. Covered earlier in this text, this means that unstable frameworks are more likely to change due to evolving requirements. Whitebox frameworks do not yet identify stable Hot Spots, which means that it is not yet clearly defined at which place the framework needs to be customized regarding the different application domains. As a result of that, users of Whitebox frameworks have to acquaint themselves with the inner structure of the framework in order to identify the places they need to target for their problem-specific customizations. In most cases, the specialization of Whitebox frameworks is done through inheritance. New functionality is added by creating subclasses of classes that already exist within the framework. This makes it easy for the user to extend the framework and hook in more specific functionality. With respect to *architectural-driven* frameworks, the flexibility clearly dominates the completeness aspect.

Data-driven frameworks, also referred to as Blackbox frameworks, are more sophisticated and mature than Whitebox frameworks and define clear interfaces for their Hot Spots. Furthermore, data-driven frameworks often come with components which are easily hooked up with the frameworks. These components are combined to achieve the same customization that is done through inheritance in Whitebox frameworks. Selection and composition of already existing components tends to be a lot easier to the user than inheritance from the framework (Adair, 1995). Thus, Blackbox frameworks show an increased ease of use towards Whitebox frameworks. Furthermore, data-driven frameworks tend to show a higher degree of completeness than flexibility as compared to architectural-driven frameworks. The increased stability of the hot spots as well as the components within the framework library provide ready-made solutions for a larger set of problems, but on the other hand are harder to modify and extend. Figure 6 opposes architectural-driven and data-driven frameworks to each other.

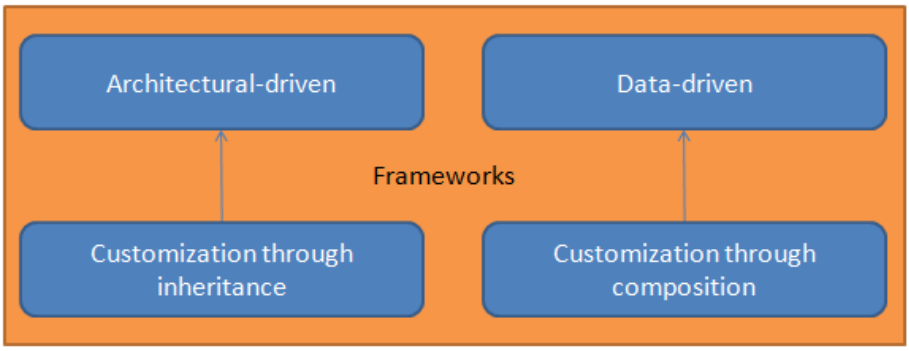

Figure 6: In architectural-driven frameworks customization is achieved through inheritance, whereas in data-driven frameworks customization is achieved through composition.

Generally frameworks start out as Whitebox frameworks that rely upon inheritance. As the domain becomes better understood, more concrete support classes are developed and the framework evolves to use more composition, mutating into a Blackbox framework. Due to the unstable nature of their Hot Spots, Whitebox frameworks require a more in-depth knowledge from the user than Blackbox frameworks. Typically, a framework will have elements of both Whitebox and Blackbox frameworks rather than be clearly one or the other (Lücke, 2005).

Quick Review - Classification of frameworks

This section classified frameworks into two classes: *Architectural-driven* frameworks are customized through inheritance, *data-driven* frameworks through composition.

2.5. Framework domains

In addition to Adair's classification of frameworks, which was introduced in the previous section, Froehlich et al. introduce further demarcation criterions, namely *scope*, *primary adaptation mechanism*, and mechanisms *by which it is used* (Froehlich, Hoover, Liu, & Sorenson, 1997).

The **scope** is understood as the area the framework is applicable to. Further classification of frameworks by their scope results into *application frameworks*, *domain frameworks*, as well as *support frameworks*.

> **Application frameworks** encompass functionality that can be applied horizontally across problem domains. In other terms this means that application frameworks typically provide functionality for problems of various problem domains. The solutions application frameworks offer are mostly common to a wide variety of problems. Graphical User Interface (GUI) frameworks, for example, find appliance in all applications that need to interact with the user via the screen and stretch horizontally across the domain. The horizontal nature of application frameworks is depicted in Figure 7.

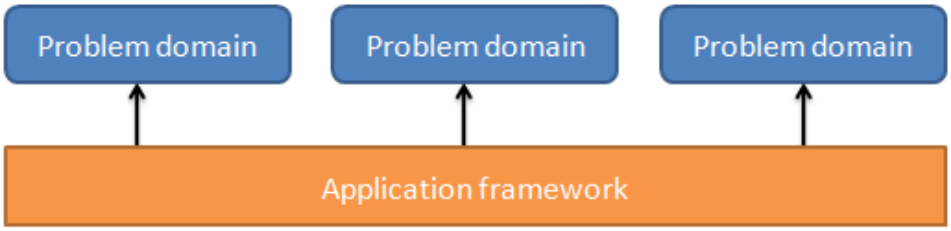

Figure 7: An application framework spans horizontally among several problem domains.

Domain frameworks on the other hand offer vertical functionality for a particular problem domain including problems of a similar nature. Operating systems, for example, all offer fundamental memory and process management functionality to other applications that are built on top of the framework. Support frameworks encompass basic system level functionality upon which other frameworks or applications can build. The vertical characteristic of a domain framework is depicted in Figure 8.

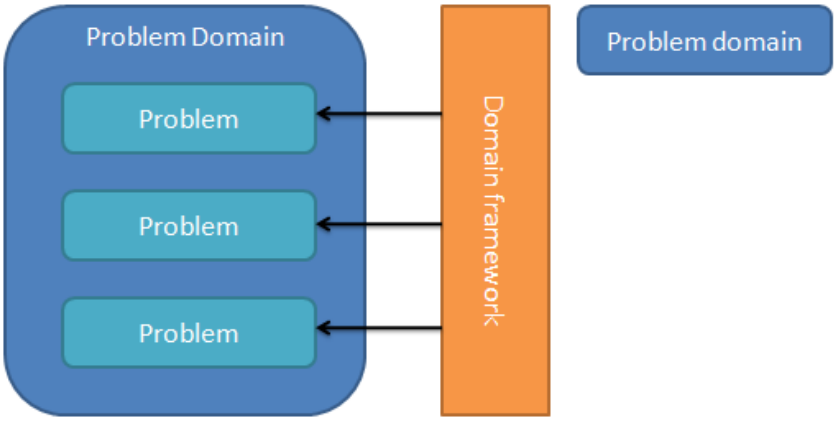

Figure 8: A domain framework spans vertically among one single problem domain.

Support frameworks provide other frameworks or application with low-level functionality and service them with, for example, file access and basic drawing technology. Application developers typically use support frameworks directly to build their applications or set on top of modifications generated by systems providers.

The other demarcation criteria that were mentioned at the beginning of this section were a framework's *adaptation* and *usage* mechanisms.

Adaptation mechanisms represent the means of framework customization, whereby a framework can primarily rely on composition or inheritance for reuse. Blackbox frameworks, for example, are customized and reused by passing existing components from the framework library back to the framework, whereas users of Whitebox frameworks inherit from existing classes to achieve the necessary customization.

Finally, the **usage mechanism** details how a framework interacts with its application extensions. Either the framework calls the application extension through function pointers or virtual methods

respectively or the application extension itself calls the framework. The first case is more typical for frameworks whereby the second case generally applies to class libraries. Related to the **Usage mechanism** is the in section 2.3 *Characteristics of Frameworks* introduced *Inversion of Control*.

> **Quick Review - Framework domains**
> This section showed how frameworks can be distinguished by their scope, adaptation mechanism and usage mechanism. Application frameworks, domain frameworks, and support frameworks are instances of the scope criterion being further subclassified.

2.6. Framework lifecycle

As an attempt to describe changes in the requirements and behavior of a software project over time, the software lifecycle model outlines common and reoccurring phases in software development. For the purpose of simplicity, this section concentrates on a software lifecycle running through the four phases *development, application, maintenance,* and *discontinuance* (Singh).

During the first phase, namely the **development phase**, framework developers first analyze the problem domain and as a result specify the requirements for the framework. Based on the gathered requirements, framework developers then start designing and implementing the framework. After the development phase is completed and the framework has reached a useful stadium, the target group that is referred to as framework users, starts incorporating the framework into their applications. If during the second phase, referred to as **application phase**, changes to the framework should become necessary, framework maintainers refine and redevelop the framework to fit the new requirements. These refinements are usually done during the **maintenance phase**, but might also be interpreted as a start-over in the framework's lifecycle.

Often framework maintainers have earlier participated in the design and development and therefore represent an experienced subgroup of the former development team. This group will also be responsible for later decisions about the discontinuance of the support of a framework at the end of its lifecycle. Mostly, this decision is made based on how excessive and expensive updating or upgrading adjustments for the framework would become due to changes in the domain. At this point, a new framework architecture that better serves the needs of the evolving problem domain might be the basis for a new framework. The old framework lifecycle ends with the **discontinuance phase**. Even though with the discontinuance phase the framework lifecycle practically ends, the hollow arrow in figure 9 from discontinuance to development symbolizes a possible restart of the lifecycle, this time with a new framework or a successor for the old one. For a graphical representation of the above explained, please refer to the following graphic.

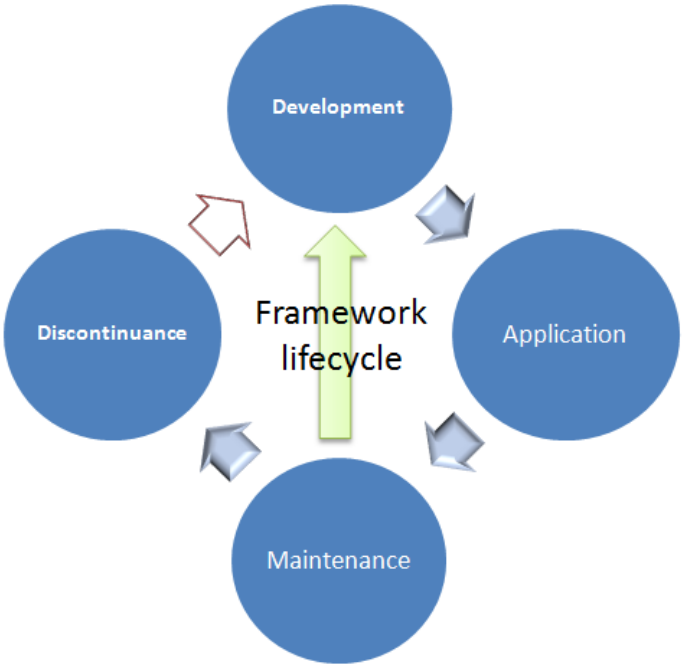

Figure 9: The framework lifecycle: The Maintenance phase can either result in a redevelopment and refinement of the framework or be the predecessor of the discontinuance of the framework.

Finally, a small remark concerning the framework lifecycle shall be stated. The model above does not show a direct transition from the application phase to the discontinuance phase (means an arrow from application to discontinuance). An indirect transition, however, is possible by passing the maintenance phase as well. Even though, from a formal point of view, this direct transition is a valid one, it would also mean that shortly after a framework has been released it would have turned out to be so inefficient or hard to use that its potential clients would immediately have started rejecting any further operations with the framework. In practice, this transition has only little significance since it is usually in the interest of the project management not to release a framework that will be of no use and thus not having any clients.

Quick Review - Framework lifecycle

This section gave an overview of the framework lifecycle and its stages. It was shown that, after a framework was developed, it is rolled out to the clients, applied, maintained and someday discontinued.

2.7. Benefits of frameworks

After the previous sections have given a profound overview of the characteristics of frameworks, this section will deduce beneficial aspects from the individual characteristics. As already indicated earlier in section 2.1, frameworks support the application development process by providing standardized 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. Thus, application

developers can focus on providing a solution for the actual problem their software addresses rather than troubling with secondary problems, such as the implementation of I/O operations or mathematical functionality. 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 application extensions are already captured by the framework. Also, the effort required for maintaining applications can be significantly reduced as multiple applications are built on top of one framework (Froehlich, Hoover, Liu, & Sorenson, 1997). The result of that is the following: In case of modifications or fixes being made to the framework, application extensions implicitly benefit from the changes since those are automatically propagated through the framework to the application extensions. As application extensions that are built on top of a framework follow the same design and share the same code base, frameworks provide consistency and therefore a better integration across platforms, meaning operating systems or simply hardware. Lücke claims in his article that frameworks can trivialize the application development to first find a fitting framework, to then learn this framework, and finally to customize it in order to satisfy the specific demands of the application extension (Lücke, 2005).

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 refining process, it forms a quality base for developing new applications. Thus, once a framework has been developed, the problem domain has already been analyzed and a working design and implementation have been produced (Arrango, Pietro-Diaz, & Pietro-Diaz, 1991).

Quick Review - Benefits of frameworks

This section discussed the beneficial aspects of software frameworks. Frameworks provide standardized solutions to reoccurring problems, capture expertise of domain experts, and enable applications to have a shortened development time. Furthermore, frameworks can reduce the effort that is necessary for maintenance, and provide a consistent design and better cross-platform integration.

2.8. Difficulties with frameworks

In contrast to the benefits that were shown previously, this section points out difficulties and downsides of framework development. In particular, problems within the development and maintenance, as well as problems framework users are confronted with are taken into focus.

Concerning the development of software frameworks, Froehlich et al. state that the building cost for a good object-oriented framework is significantly increased compared to single applications. A large amount of time is spent on defining and refining the abstraction of the framework in cyclic phases. Additionally, framework designers and developers have to incorporate shifting problem domains into their thinking and keep their framework flexible enough to adopt future changes. If the problem

domain changes and the framework fails to adopt these changes, it becomes less valuable to the community of developers. In general, frameworks built for a stable problem domain tend to live longer than frameworks created for rather unstable problem domains. Parallel to changes within the framework due to emerging requirements, the impacts of the framework's evolutionary transcend on depending applications must be taken into account. If the architecture of any interfaces of the framework change, then upgrading the applications built on top of the framework might be a costly procedure. Contrarily, if the applications are not upgraded to newer versions of the framework, the advantage of having a common code base gets lost (Froehlich, Hoover, Liu, & Sorenson, 1997).

But not only does the development of frameworks show an increased complexity as compared to the development of normal applications Also, the framework users are confronted with several difficulties. One of the major challenges when working with frameworks is to identify and choose the framework that best serves the needs of the application that is under development. Thereby, the choice should be guided by a set of criteria, such as the quality of the framework, the documentation that is available, as well as the support that is offered by the framework providers.

Another major concern about the use of frameworks is that the learning of the framework, which is a necessary step prior to the development of the application extension, is most often a time consuming process. Covering a general aspect of the framework learning curve, the 90-10 rule states that the progression of the curve reveals the following peculiarity: 90% of the application development is done in 10% of the time, whereby the remaining 10% require 90% of the overall development time (Froehlich, Hoover, Liu, & Sorenson, 1997). In the beginning of the learning phase users can quickly solve easy problems since they are utilizing ready-made components of the framework library that address general reoccurring problems. As soon as more difficult and more specific problems need to be solved, whereby no ready-made components yet exist, the users will find themselves lacking the understanding of how to achieve the goals using the framework. They therefore will have to learn more in detail about the internals of the framework and understand how to extend the built-in functionality or modify the framework's existing behavior to serve their needs. They will have to acquaint themselves with the framework's Hot Spots –places that allow modification- and understand which Hook Methods to override in order to inject their user-defined code into the framework. The following illustration depicts above explained interrelation in the framework learning curve (Werfel, Xie, & Seung).

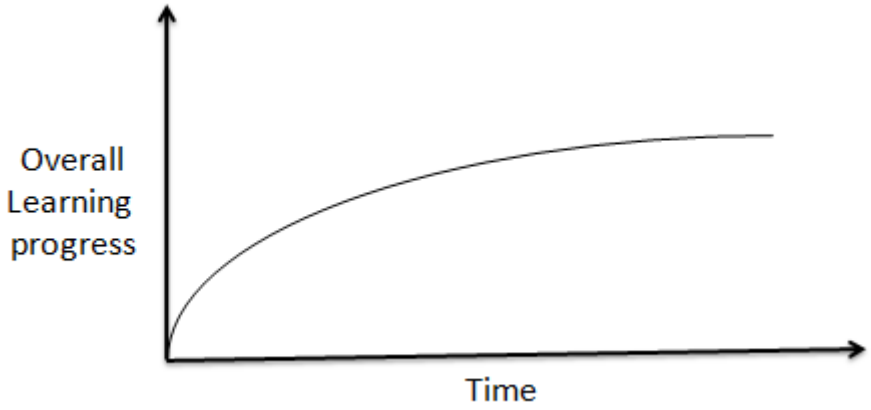

Figure 10: The framework learning curve: In the early phase, the user's knowledge about the framework increases quickly. Over time, the knowledge gain becomes more and more marginal.

Quick Review - Difficulties with frameworks

This section revealed several difficulties connected to the utilization of software frameworks. Not only do frameworks show a significantly increased building cost, but they also suffer, depending on their flexibility, from shifting problem domains. On the application extension's side, difficulties evolve from the process of identifying the right framework, and the subsequent learning that needs to be done.

2.9. Economic significance of frameworks

The previous two sections covered benefits and difficulties regarding framework development and use. It was shown that frameworks can help to significantly shorten the development time of derived application extensions and further simplify the maintenance of those. In the following benefits and downsides of frameworks are opposed to each other and an attempt to give an economic conclusion about the overall profitability of software frameworks is made.

Due to the repetitive refinement cycles during the development phase and the intensive testing that is necessary to ensure the framework's quality, a framework needs more time to be developed than a normal application. Furthermore, the clients that use the framework also need a certain amount of time to learn the framework before they can start writing their application extensions. Therefore, frameworks should be considered as long-term investments whereby the benefits gained from the development of the framework are not necessarily immediate. First, frameworks require more effort to be build and learned compared to normal applications. Since application extensions are developed on top of frameworks and the users usually rely on the framework as being a bug-free foundation for their software development project, frameworks must run through significantly more intensive and repetitive testing. Additionally, the future success of the framework is not only dependent on the usability and flexibility towards various problem scenarios, but also on the reliability the framework

provides. Frameworks with significant bugs and flaws will not be accepted by the clients and thus are predetermined to fail.

Another concern about the usage of frameworks is that debugging the application extension can require a significantly increased amount of time and money as compared to fixing the bug on the framework's side. In frameworks which are not open source this due to the developer lacking the ability to understand what's going on within the framework and just being presented the results of his method invocation. For example, a non-open source framework might appear to the user as a magical black box where he merely specifies the input for and gets back the result of the operation without understanding the actual transition from input to output. If the developer does not understand the internals of the method he called, he might also not know in which format the input is expected to be specified in order to obtain a desired output. Open source frameworks on the other hand enable the user to read the source files, but the high complexity of the source code, poorly commented instruction lines, as well as thin documentation might turn out to be an obstacle when trying to reconstruct how to correctly call functions and pass arguments. Figure 11 and 12 give an example for how non-open source frameworks can hinder the users in their understanding of the framework.

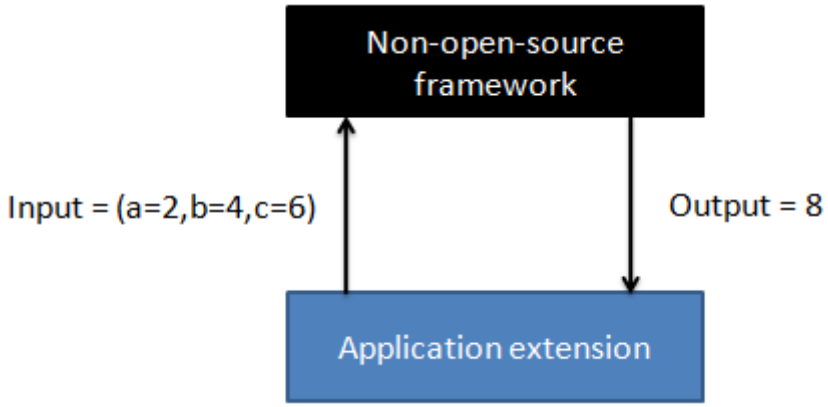

Figure 11: Non-open-source frameworks do not provide the user with the understanding of how input is transformed into output.

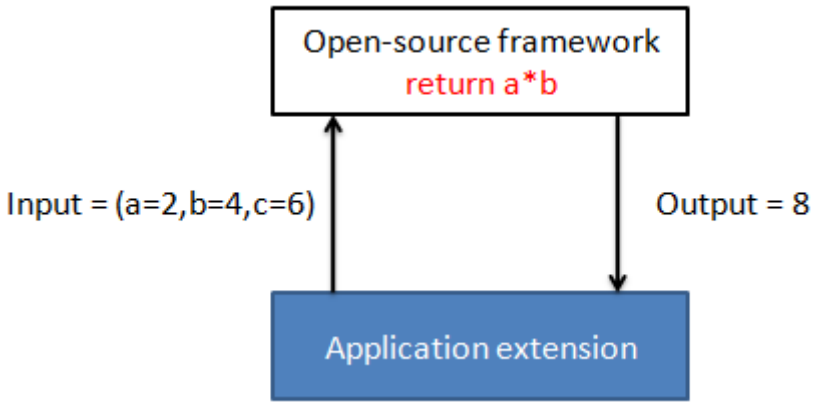

Figure 12: Open-source frameworks on the other hand allow the user to understand the internal process.

Another important aspect is that the framework users expect the framework to be a solid basis for their development and thus works correct in every situation (Froehlich, Hoover, Liu, & Sorenson, 1997). From this it follows that, in case of an application not delivering correct results, framework users will first continue to search for failures within their own application before they for sure will have identified the failure not lying within their own code.

Further difficulties can evolve from the circumstance that frameworks require detailed documentation including use cases and examples. As mentioned earlier, framework users will hardly have the chance to understand the internals of the source code, regardless of the framework being open source or not. Therefore, it is even more important to provide the user with all information that is necessary to operate the framework in form of a detailed documentation and coding examples. Identifying and gathering the documentation requirements as well as specific attitudes of the target group makes the creation of documentation a costly and timely process. The following example helps illustrating the last statement: Documentation for a framework that children will use in school for learning the basics of programming differs in language and form from a documentation that is written for experienced programmers.

The last cost driver that shall be discussed and also represents the main topic of this work arises from the continuance of maintenance and support for the adoption of changes within the problem domain the framework services. Frameworks that will not respond to changing requirements or provide fixes for bugs will soon or later be rejected by their clients and substituted by alternatives. In general companies are only able to discontinue the servicing of their framework without losing their clients, if they can or at least intend to push their clients towards a successor of the older version or an alternative. Since the framework developers will have to provide support to their clients for migrating away from the old framework towards a possible successor, another considerable cost driver evolves at this place (Froehlich, Hoover, Liu, & Sorenson, 1997).

Quick Review - Economic significance of frameworks (Cost drivers)

This section detailed that frameworks tend to require more effort to be built and learned than usual applications. Cost drivers evolve from the fact that frameworks require detailed documentation and sufficient maintenance and support. Also, later migration towards newer frameworks must be supported, if the goal is to retain the client community. On the application extension's side, a higher debug complexity might bind an increased amount of resources.

While the last paragraph has given an overview of several cost drivers related to the utilization of frameworks, the following will detail the economic benefits contrarily. Early in its life, a framework will probably require more cost-intensive routine maintenance to fix initial bugs and respond to more frequent client requests (figure 13:a). Over time, however, the cost for maintenance and support will

drop and remain at a level that is sufficient to allow updates and necessary changes to the framework (figure 13:b). With the replacement of an existing framework by a successor the maintenance and support cost curve rises again (figure 13:c) before it will finally drop to zero. A maintenance and support cost curve that reached zero stands for the migration away from the framework towards an alternative has completed and the support and maintenance for the framework has been discontinued entirely (figure 13:d). The following illustration depicts the maintenance and support cost curve associated with frameworks.

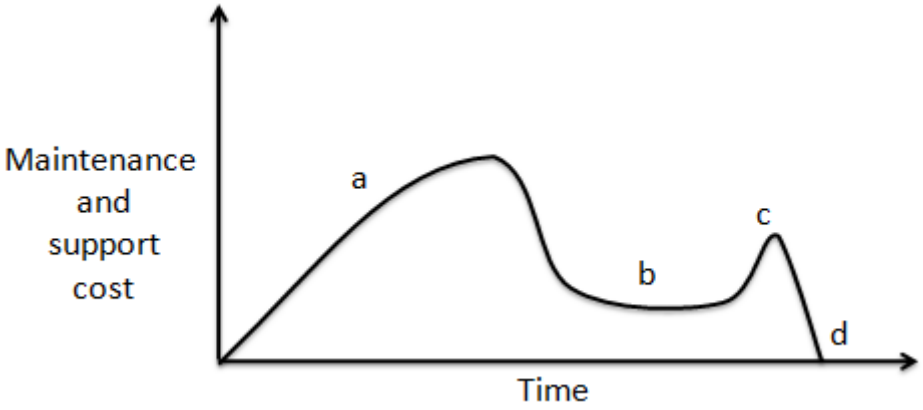

Figure 13: Due to initial bug fixing and refinement, the maintenance and support cost curve rises in the early phase of the framework (a) and afterwards falls down to a point that is sufficient to keep the support alive (b). With the discontinuance of the framework, additional effort for migrating away from the old version to its successor becomes necessary (c) until the cost curve finally drops to zero (d).

In section 2.7, the benefits resulting from applications that are built on top of frameworks were discussed in detail. One of the key benefits software frameworks offer are standardized solutions to reoccurring problems developers might face. This results in frameworks enabling application developers to concentrate on their actual application rather than to trouble with the implementation of low-level support functionality. Another key benefit of applications that sit on top of frameworks was that changes to the frameworks are automatically propagated to all dependent application extensions which makes bug fixing and extension programming a lot easier and less expensive (Froehlich, Hoover, Liu, & Sorenson, 1997). However, in order to realize those benefits the organization has to commit resources to support the earlier identified cost drivers. Organizations must be able to assist clients with their issues and to respond to their problems and requests. This can be achieved through, for example, maintaining mailing lists, news groups, as well as support hotlines or community forums.

After all, over the lifetime of a framework, the cost of supporting it becomes a benefit. The cost of supporting one framework with three dependent applications will be less than the cost of supporting three independent applications with duplicate code. In general the following rule applies: The more applications that use a framework, the bigger the savings on cost. In figure 14 and 15, the measure of the accumulated maintenance cost (AC) illustrates this circumstance. In the long run, using frameworks can reduce the accumulated maintenance cost of applications since all applications are based on top of one common code base. Changes to the code base would result in the same changes

to all depending applications, which makes it easier and less cost-intensive to propagate fixes and updates. Figure 14 and 15 demonstrates how the accumulated maintenance cost of single applications behaves in contrast to the accumulated maintenance cost of applications built from one common framework.

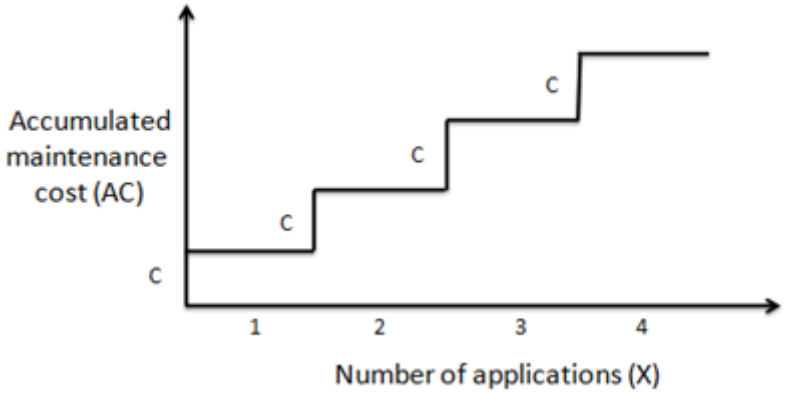

Figure 14: Provided the same changes are required by every application: The overall cost for maintaining independent applications is the aggregate of the single maintenance costs of the applications c or a multiple of the single maintenance cost (AC = X*C).

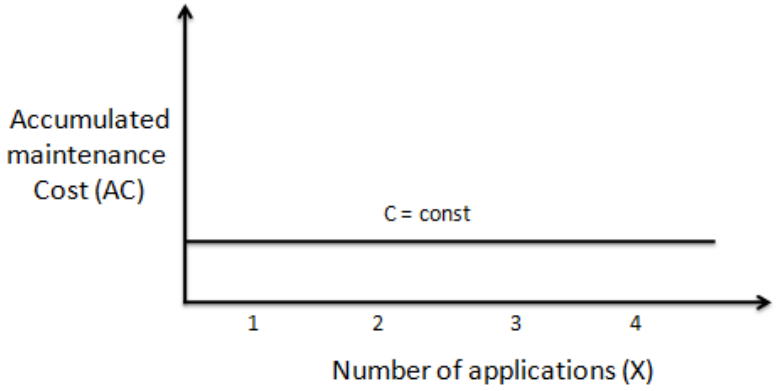

Figure 15: Provided the same changes are required by every application: The overall cost for maintaining applications built from one common framework is constant and exactly the cost that refers to the maintenance of the framework (AC=C).

Other economic outcomes of frameworks are sometimes not so obvious and interrelated to other technologies and platforms. As shown in the case of the Windows Mouse Wheel API, frameworks sometimes can also support platform adoption. When wheel mice were introduced to the market Microsoft's operating system Windows provided access to the mouse and the mouse wheel via the Windows Application Programming Interface (WinAPI). With the WinAPI developers could now make use of the new mouse wheel technology and incorporate mouse wheel functionality within their applications. However, before the Microsoft Foundation Classes (MFC) provided an encapsulated version of the Windows mouse wheel API, the mouse wheel functionality was not as popular

amongst developers as Microsoft has expected it to be. Due to the MFC providing an improved interface to the Windows API that is easier and more effective to use, developers started perceiving the new technology as interesting for their own applications and realized the benefit the mouse wheel offers to the users of their applications. The Microsoft Foundation classes in this case supported the introduction of a new technology on the Windows operating system.

Quick Review - Economic significance of frameworks (Benefits)

The remainder of this section stated the economic benefits that are related to the utilization of software frameworks. When a framework is upgraded or maintained, those changes are automatically propagated to the application extensions that were built from the framework. In case of multiple single applications that were not built from a framework, the effort that is necessary for maintaining those single applications would be a multiple of the maintenance effort necessary for the maintaining the framework.

3. Framework design and development

After the previous chapter has introduced the basic terminology that exists within the framework world, this chapter will concentrate on architectural and project-related topics of the actual framework development process. First, managerial aspects of how to deal with project dependencies and how to arrange resources will be discussed. The following paragraphs will cover general aspects of the framework requirements gathering and then switch over to an outline of the actual framework development process. The chapter will close with an introduction to general and specific framework development techniques as well as concerns related to the deployment of frameworks.

3.1. Managing project dependencies

Before addressing concrete development techniques in the following sections, this section will start with an overview of managerial aspects of software and in particular framework development projects. The issue is mainly broached of how to structure the project and how to avert negative effects evolving from team dynamics within the work force.

As described by Taligent Inc., software projects, such as frameworks, can often be factored into a number of separate small projects and then assigned to teams. Thereby, Taligent states that development teams of three to four developers are generally more effective than larger teams (Taligent, 1995).

McShane in his work states something similar about team dynamics (McShane, 2002). In order to tackle big projects, it is not enough to simply divide the overall workload of the project through the hours a person can work during the project time-frame and put the estimated number of persons together in one big team. While the managing aspect of small teams is clear and straightforward, the effort required to manage big teams increases disproportionally. The bigger the team, the higher the rate the required project management effort grows at. In practice, this refers to certain dynamics and characteristics that lower the average productivity of each worker with increasing team size. Big teams suffer from the fact that controlling how much every team member contributes to the project's success becomes increasingly harder. Some team members might even be aware of this and consciously reduce their work output. Furthermore, dispatching work and tasks across big teams is significantly more difficult as compared to small ones. It can be assumed that the actual task each team member has been assigned to gets more and more out of focus, so that it becomes harder to avoid duplication of effort. Figure 16 illustrates how team performance and team size relate to each other (McShane, 2002).

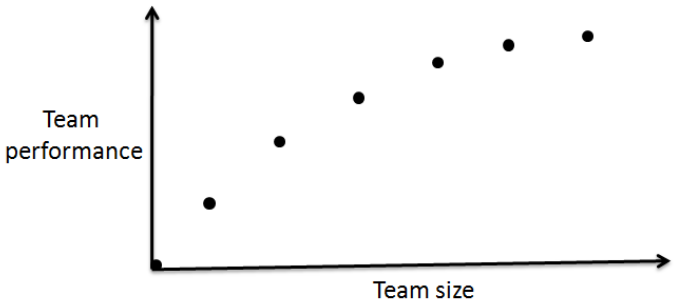

Figure 16: The marginal team performance decreases with increasing team size.

Based on above considerations, Taligent suggests splitting the development into smaller teams, if a project takes more than three to four developers. This ensures that the overall project is broken down into non-redundant subprojects which can be processed independently and integrated with each other after completion (Taligent, 1995). Moreover, the management can efficiently dispatch the workload across the team while ensuring that each team member performs at the desired level.

On the other extreme, teams of two to four developers are usually more effective than a single developer unless the developer is both experienced and an expert in the required field. The outcome of single persons working alone on a project is often of a lower quality than the outcome of a small team, even though the single person might have a higher productivity in the first instance. Combining the experience and different point of views of several people contributes to identifying flaws early in the development process and thus usually generates higher quality results.

However, working with several small teams introduces additional challenges. At first, programmers might lack the understanding for the interrelations of the overall project since they are focused on one aspect only. Thus, it might happen that the team's result deviates from what was required which complicates further integration work. Furthermore, it is inevitable that all teams stick to the architectural obligations they were given in order to later integrate the teams' results with each other and fit them in the overall project's pattern. A more serious downside of smaller teams is that, in case of one team being dependent on the progress of another team, these dependencies might create bottlenecks and block the teams from working effectively.

To alleviate these problems, Taligent Inc. proposes to appoint a project architect who maintains the overview and ensures that the subcomponents resulting from the individual teams ultimately work together in the combined project. Furthermore, following standard design and coding guidelines helps the understanding of developers who will later work on the code and potentially extend or maintain it. Decoupling the components and isolating dependencies in intermediary classes ensures that work on every component can be performed independently and the final integration can be done without any complications. If one subcomponent requires the service of another subcomponent, the connection can be implemented through an interface or server object resulting in only one object being dependent on the other subcomponent. Frameworks consisting of loosely coupled components generally are more flexible, not only from a user's perspective but also from the

perspective of future maintenance. Figure 17 shows components being tightly coupled and maintaining several dependencies to other components. Figure 18, contrarily, depicts how components can be decoupled from each other through a server object.

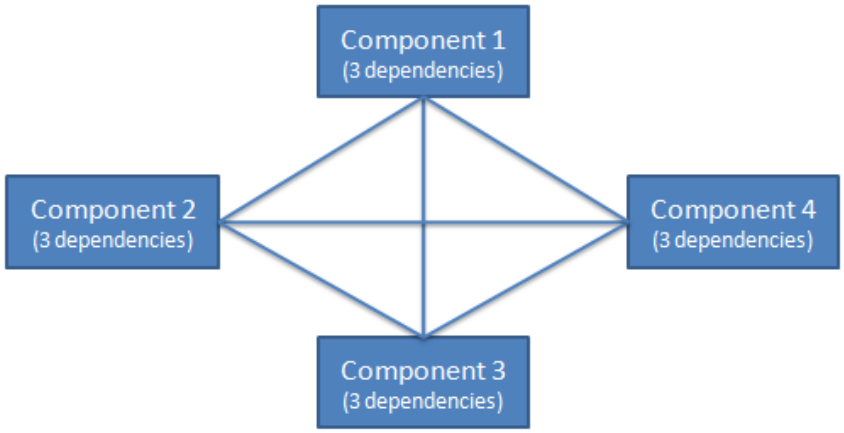

Figure 17: Components communicating with each other establish connections for each component and thus have multiple dependencies.

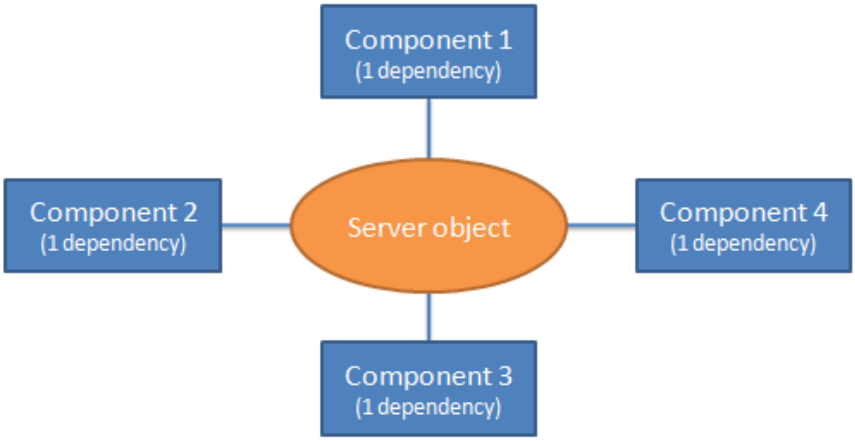

Figure 18: communicating with each other establish connections to each other through a server object and thus have only one dependency.

Quick Review - Managing project dependencies

This section showed how larger framework development teams can be split into several smaller ones in order to avoid negative phenomena and increase the overall team performance. Furthermore, it is important to decouple the individual components each team is working on, which can be achieved through server objects. To maintain the overview of all existing development initiatives and ensure their final integration will work without complications, a project architect can be appointed.

3.2. Framework requirements gathering

After the previous section has given an introduction to managerial aspects of the framework development process, this section emphasizes the process of requirements gathering. Requirements gathering takes place prior to the framework development and design phase whereby some important differences to normal application development exist.

Designing frameworks differs significantly from designing single applications (Froehlich, Hoover, Liu, & Sorenson, 1997). On the one hand, the level of abstraction turns out to be an important differentiation criterion: Frameworks provide generic solutions for a set of similar problems whereby single applications provide solutions for one specific problem only. On the other hand frameworks are by their very nature incomplete. While an application has all the components it needs to execute, a framework might have places inside that need to be extended by concrete solutions. Furthermore, a framework does not cover all the functionality that is required to solve a particular problem.

While normal applications are usually developed within one development cycle and then released, frameworks are developed iteratively. This helps clarifying Hot Spots that are necessary to enable the users to repeatedly make use of the framework (for more detail on framework lifecycle, please refer to section 2.6). Relating to a framework's lifecycle, the so called *Framework Release Problem* points out that there are no clear criteria existing for when a framework should be released (Froehlich, Hoover, Liu, & Sorenson, 1997). Even though iteration through several development cycles helps manifesting the Hot Spots and eliminating bugs, there can hardly be a state where the framework does not contain any minor flaws anymore. On the contrary side, at a later point in time after the framework has already been released, it can still be revealed that one Hot Spot should better have been placed at a different location. As a consequence of lacking the possibility to identify the number of cycles that are necessary to deliver a framework that does not require any later maintenance, the decision about the release is mostly made spontaneously.

To better understand the requirements to the framework and the technical challenges related to its development, it is common practice to first start out with a series of application prototypes that incorporate certain aspects of the future framework. Based on the insights gained from the application prototypes the modeling of the Hot Spots can begin. It might even happen that an application prototype seems to be mature enough to function as the actual framework or at least as its basis. However, as Froehlich et al. state, transforming an application prototype into a framework is no trivial step and comes with several challenges evolving. This is because the application's architecture mostly does not fit the framework's architecture as the application was developed as a design study without covering reusability and extensibility aspects. Furthermore, changes in the application's architecture might be even more time consuming than directly redeveloping the framework from scratch (Froehlich, Hoover, Liu, & Sorenson, 1997).

Quick Review - Framework requirements gathering

This section covered some of the important differences in the development of frameworks compared to the development of usual applications. While usual applications are generally developed within one development cycle and then released, frameworks are developed iteratively. The *Framework Release Problem* states that even though the iteration through several development cycles helps eliminating bugs, there can hardly be a state where the framework does not contain any minor bugs anymore. Hence, there is no analytical way to determine the optimal release date of a framework.

3.3. Framework development

After the requirements for the framework have been clarified, the development process can start. Several approaches for developing frameworks exist. Each of these approaches follows the steps **analysis, design and implementation, testing, and refinement**. This section shows that a framework is not built during a single pass of this sequence, but rather passes multiple iterations of these steps. (The content of this section is extracted from a work by Froehlich et al.)

Analysis:

The analysis is the first step in developing a framework and refers to the analysis of the actual problem domain. Analyzing the framework's domain helps determining the framework's primary or key abstractions which will form the framework core. Domain experts identify the size of the domain the framework will cover, the abstractions that will be incorporated within the framework, and how variations between applications within the domain will be dealt with. The following illustration summarizes how the analysis of a problem domain delivers several key aspects for the development of the framework core.

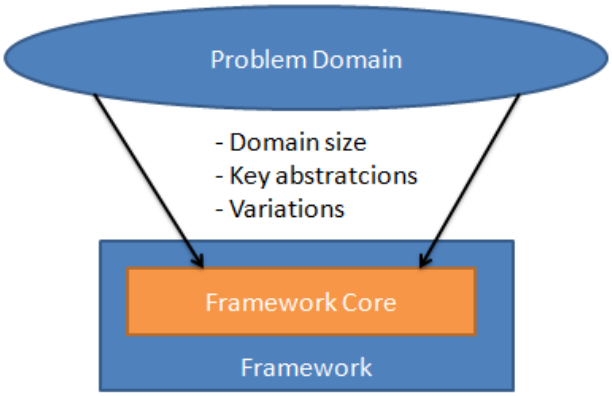

Figure 19: The first step in the development is the analysis which delivers results determining the framework core.

One key decision that domain experts will have to make is how large the domain coverage of the future framework will be. Will the framework apply to a large or a small domain, a narrow part of a

domain, or even several domains? Frameworks with high domain coverage will be reusable in more situations and thus be more flexible, but therefore might be difficult to maintain. Contrarily, building widely applicable frameworks is a significant undertaking and can be very costly. Narrow frameworks on the other hand might be insufficient for many cases and developers would have to add a considerable amount of additional functionality on their own. Furthermore, narrow frameworks are easily affected by changes in the domain. While wider frameworks might evolve to fit changing requirements, narrow frameworks might no longer be applicable due to their inflexibility. Contrarily, narrow frameworks tend to be smaller, less complicated, and therefore easier to maintain and potentially easier to use. Whereas large frameworks often require using a big amount of functionality that is within these, regardless of whether it is needed or not, several smaller frameworks instead allow users to only incorporate the functionality into their applications that they really need. In general, the benefits of small frameworks outweigh the drawbacks.

The following figure opposes the characteristics of wide to the ones of narrow frameworks:

Wide frameworks	**Narrow frameworks**
- flexible - applicable to many cases - difficult to maintain - costly - stable against changes in domain - evolve	- less flexible - applicable to a few cases only - easier to maintain - less cost-intensive - affected by changes in domain - hardly evolve
- complicated - harder to use	- less complicated - easier to use

Figure 20: Wide and narrow frameworks. With respect to usability, the benefits of narrow frameworks in general outweigh the drawbacks.

Design and implementation

During the design phase the structure for the abstractions, Frozen Spots, and Hot Spots are determined. Parts of the framework might undergo a redesign while other parts are already being implemented. For the process of refining the abstractions, reviews are recommended. During these reviews not only the functionality is examined, but also hooks and other means of client interaction.

In order to develop easy to use and flexible software, Taligent Inc proposes a number of suggestions (Taligent, 1995). First, the number of classes and methods that the user will have to override should be reduced to a minimum. This increases the maintainability of application extensions and supports upgrading to newer framework revisions as with a reduced number of overridable functions, less application code needs to be considered. Furthermore, the less overridable functions within a framework a user will have to take into account, the faster the learning progress and the easier the use of the framework. Related to the first suggestion, simplifying the interaction between the

framework and the application extension increases the framework's ease of use. Instead of passing a function pointer to an event handler, for example, that calls the function on a particular event, the events could rather be explicitly exposed through virtual overridable functions. A framework's success can further be determined by its portability to other platforms. Frameworks that are easily usable across platforms and are, for example, not dependent on a particular operating system will be used by more clients and thus promise to be more successful. Therefore, it is important to isolate or even avoid platform-dependent code as far as possible. Isolation can be achieved by putting platform-dependent code into separate modules that can be linked to the framework on demand. Several modules for different platforms might already be delivered with the framework release while developers are free to write additional modules for platforms that are not yet supported. During the maintenance or revision phase, these additional platform modules might be added to the framework core and thus become an integral part of the framework. Another advice Taligent Inc gives is to keep as much as possible done within the framework rather than escalating code work to the user. Instead of instantiating an object that is used within the framework, setting up its properties and calling several methods on it before it is passed to a framework function, the user could just pass the necessary parameters to a function that then handles the further processing of the object and the passing on to other functions. Even though it is beneficial to provide the user with certain default behavior, the default behavior should always be overridable through virtual functions of factored code. Blocking the user from modifying the default behavior might result in the framework not being applicable to the framework's problem domain anymore as a result of limited flexibility. Finally, it is suggested by Taligent Inc to provide notification hooks within the framework, so that users can react to important state changes (Taligent, 1995). Users of a networking framework, such as RakNet[2], might be interested in clients joining or leaving the current network session. In this case, the notification hook could consist of a general message loop the user can query and react on certain messages.

The following summarizes above introduced criteria for developing easy-to-use and flexible software:

- **Reduce number of classes that need to be overridden to a minimum**
- **Simplify interactions between the framework and the application extensions**
- **Isolate platform-dependent code**
- **Keep as much code work as possible within the framework**
- **Keep default behavior overridable**
- **Provide notification hooks**

More general advices concerning the framework development are proposed by Birrer and Eggenschwiler. They point out that the consolidation of similar functionality into a single abstraction requires less code to be written and supports the user's understanding of the overall architecture of

[2] http://www.jenkinssoftware.com (May 5th, 2008)

the framework. Furthermore, larger abstractions should be broken down into smaller ones with greater flexibility (Birrer & Eggenschwiler, 1995). A graphics framework, for example, that defines a certain set of geometric primitives, should define separate rendering functions for each primitive rather than having one general rendering function for all primitives together. Thus, binding certain abstractions to concrete objects as our native perception would do again helps the user's understanding of the overall concept of the framework. Additionally, maintaining separate encapsulations of functionality is a lot easier and less likely to accidentally affect other parts as well. Concerning the way customization is achieved, Birrer and Eggenschwiler favor composition to inheritance. They argue that composition of already existing objects is a lot easier and faster than extending abstract classes from the framework and instantiating the derivatives. Even though composition might yield faster results in the beginning, more complex problems for which there is no ready-made component yet existing can only be solved by extending the framework's functionality through inheritance (Birrer & Eggenschwiler, 1995).
General advices for the framework development are summarized in the following:

- **Consolidate similar functionality into a single abstraction**
- **Break down larger abstractions into small ones**
- **Maintain separate encapsulations of functionality**

During the design phase, trade-offs must be taken into accounts between hooks and structuring Hot Spots, since frameworks cannot be arbitrarily flexible. Furthermore, trade-offs between flexibility and usability must be considered, since the most flexible framework would have very little concrete implementations and thus would require a big deal of work on the part of the framework user.

Testing

After the framework has been developed, it will enter, as every normal application, the testing phase. Testing frameworks is distinguished into two types, referred to as *Isolated Testing* and *Framework Use Test* (Froehlich, Hoover, Liu, & Sorenson, 1997). *Testing in isolation* is done without any application extensions possibly influencing the testing results and helps identifying defects within the framework itself. With regards to the later applicability of a framework, isolated testing is an inevitable procedure, since defects within the framework will be passed on to all application extensions developed from the framework. Defects within the framework would force the users to either fix them on their own or to find a work-around. From the perspective of the framework's usability and connected to that its success, neither of both options is acceptable, so that serious bugs are to be avoided within the release version of the framework in all conscience. The second here in this work covered testing method refers to the framework being used to actually develop applications and thus represents the more significant test. This is due to the development team not being able to know about their framework's usability unless it actually has been used. Furthermore, making practical use of the framework serves as means of testing the hooks that were designed to

address the Hot Spots within the framework. During the *Framework Use Test* the way hooks are provided will prove as efficient or not and the places the hooks were installed at as feasible or as in need of refinement. Ultimately, using the framework to create applications helps exposing areas where the framework is incomplete or not flexible enough. It might turn out, for example, that components that are delivered together with the framework are not well designed or address rarely occurring scenarios in practice. On the other hand, adding additional components to the framework library might significantly improve the framework's ease of use.

Refinement

After the testing, abstractions of the framework that turned out to be insufficient often need to be extended or refined. Thereby, the results from the isolated testing as well as the framework use test are incorporated within the enhancements. As indicated in the beginning of this section developing frameworks is a highly iterative process that often requires many cycles through above mentioned steps.

Quick Review - Framework development

This section highlighted the individual steps in the development of frameworks. The analysis is the first step and refers to the analysis of the problem domain. After the second step, the design and implementation, has finished, the framework passes through the testing phase. In this section, testing was distinguished into *testing in isolation* and the framework use test. The last step, finally, handles the refinement or extension of abstractions that turned out to be insufficient during the testing phase.

3.4. General framework development techniques

Frameworks show certain invariants and common aspects regarding their development that need to be considered. Before more specific development techniques are covered in the next section, this section details those invariants of the framework development that are in the following referred to as **abstract and concrete classes**, **composition and inheritance**, **Hot Spots and Frozen Spots**. Additionally, the concept of **framework families** will be briefly introduced.

Abstract and concrete classes

The framework core often consists of a set of *abstract classes* that embody the basic architecture and interaction of the framework. The abstract classes should be flexible and extensible since they capture the interactions between the framework's elements, generally in the form of hook methods. The number of core classes is usually very limited as it is the goal of the framework design to provide clear and manageable means of interaction. *Concrete classes* that usually contain very specific solutions to specific problems are added to the framework library and thereby aim at enhancing the

framework's completeness. In most cases, the concrete classes of the framework library are specializations of the abstract classes within the framework core supporting more specific problem scenarios. Even though frameworks can never be absolutely complete, a good framework library supports the framework through reaching a level of relative completeness by supplying ready-made components for common scenarios. If certain parts of the framework library are not needed in a arbitrary application, they can simply be excluded.

Composition and Inheritance

In order to provide hooking into the framework, composition and inheritance are considered to be the two main ways (Adair, 1995). By nature, **composition** of already existing components tends to be easier than deriving from classes and typically encompasses the use of parameterized types or callback functions. A class, for instance, that needs to be customized will have parameters that are to be filled in by the user. One of the benefits of composition over inheritance is that the user does not need an in-depth knowledge of how particular components operate. Furthermore, composition allows the components to be changed dynamically at runtime which is hardly possible using inheritance. In case of changes in the framework's behavior becoming necessary, instantiating objects of classes that support the newly desired behavior and passing these objects back to the framework will be enough to achieve behavioral modification. Composition in combination with a large number of concrete classes within the framework library makes adapting the framework a lot easier and reduces this process to simply choosing the right component. Developers then just need to understand which existing component can perform the desired functionality.

Inheritance on the other side describes the procedure of specializing methods from abstract classes. Compared to composition it requires a considerable understanding of the abstract classes and their interactions with other classes. As a result of this, inheritance is likely to be more difficult and error prone. The advantage of inheritance over composition is that a high degree of flexibility is sustained and extensibility is incorporated. Application developers can easily add functionality to subclasses of any existing class which is not easily accommodated with composition. In many cases, hooks with regards to inheritance are of the form of abstract classes that provide default methods that can be overridden by the child class.

Composition is best used when interfaces and the means of how to use the framework are well defined, whereas *inheritance* provides functionality in cases where the full spectrum of the functionality cannot be anticipated. Moreover, composition forces conformance to specific interfaces and functionality which cannot be easily changed (Adair, 1995).

Hot Spots and Frozen Spots

Hot Spots represent the parts of the framework that require user interaction and are considered as the means of customization. Accordingly, *Frozen Spots* encompass the parts of the framework that are equal to every application and thus do not require any interaction on the client's side. In a 3D

graphics framework, for instance, the actual rendering process of primitives to the screen under consideration of the coordinate system maps to a Frozen Spot as it is hidden from the user and remains the same for every application. Telling the rendering function what primitives to render and where to place them can be achieved by providing this information through appropriate Hook Methods reflecting a Hot Spot. Thereby, each Hot Spot will likely contain several hooks associated with the Hot Spot as they describe how specific changes can be made. In the case of the 3D graphics framework the *rendering* Hot Spot could exist of one hook for communicating which primitives to render and another hook for passing the positions of the primitives within the coordinate system.
In general, three main ways exist in which frameworks can be modified using hooks (Froehlich, Hoover, Liu, & Sorenson, 1997).

1. **Providing pre-built components** within the library that are used directly or passed back to the framework to achieve the desired customization.
2. **Providing parameterized classes or patterns** which are filled in by the user.
3. **Deriving subclasses from existing framework classes** and adding new functionality when it is difficult to anticipate how certain Hot Spots are used.

Concerning the desired degree of a Hot Spot's flexibility, it must be remarked that flexibility and ease of use relate indirect proportionally to each other. This means that an increase in flexibility will result in a decrease in the ease of use. The reason for that is simply that an increased flexibility can only be achieved by further abstracting the framework from its targeted problem domain. But the more abstract the framework becomes, the less concrete functionality for specific problems is available. Thus, the flexibility aspect and the ease of use have to be balanced in a way that promises the highest possible success. This balance, however, is hard to find and can often only be approximated. Another significant aspect of customization performed through Hook Methods is that with regards to the changeability of the framework's behavior at runtime, composition is preferred to inheritance if the runtime behavior of a framework changes. As it is hardly possible to generate code at runtime in a way that is efficient, inheritance in most cases is less applicable to address changing runtime behavior (Froehlich, Hoover, Liu, & Sorenson, 1997).

Framework families

Similar to individual classes being a specialization of more abstract classes, a framework can be a specialization of another more abstract framework. Thereby, the specialized framework can provide more support for special cases within the framework domain. From another perspective, abstract frameworks can be considered as providing basic services and interactions to more specialized frameworks that focus on more specific tasks. While abstract frameworks provide flexibility, specialized frameworks incorporate completeness and ease of use. Thus, framework families are a means of dealing with complexity. Instead of including all possible options within one single, complex framework, related options can be bundled into more specialized frameworks. This eventually not

only lightens up frameworks but also generates higher flexibility resulting in an increased performance as for a specific problem domain irrelevant parts are simply left out. A bottom-up approach for developing framework families is to create a solution for an initial problem first and to then generalize this initial solution. The generalized framework might then again be abstracted into a more abstract framework while this process continues until no more general problem solution is found. It can be problematic to clearly identify what the more general solution for the next level of abstraction would be. Figure 21 summarizes how frameworks can be abstracted into other frameworks and thus form a framework family (Froehlich, Hoover, Liu, & Sorenson, 1997).

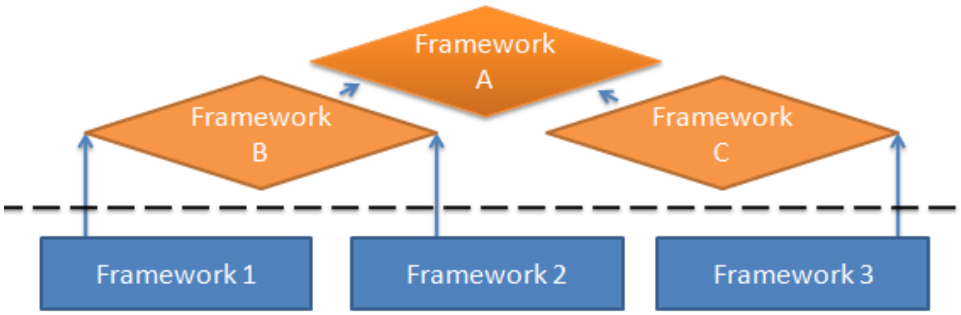

Figure 21: A framework family: Framework A abstracts framework B and C and thus represents the root of the family. Framework B abstracts framework 1 and 2, framework C abstracts framework 3.

Quick Review - General framework development techniques

This section gave an overview of framework development techniques in general. After abstract were distinguished from concrete classes, composition was opposed to inheritance. Composition tends to be easier to use whereby inheritance provides a higher degree of flexibility. Hot Spots were identified as the means of customization whereas Frozen Spots represent the framework invariant parts. Finally, framework families were introduced and it was differentiated between the top-down and the bottom-up approach.

3.5. Specific framework development techniques

While the last section gave an overview of general framework development techniques, this section will highlight a selection of more specific aspects of the framework development process. The here described principles and concepts can provide significant aid to the development as such as well as to the quality of the final product. Covered techniques are **domain analysis**, **software architecture**, **domain-specific software architecture**, **design patterns**, **open implementation**, as well as **contracts**.

Domain analysis

Domain analysis techniques focus on finding requirements for an entire domain rather than just for a single application. Without determining specific designs, domain analysis techniques identify concepts and connections between concepts within a domain. Furthermore, they support the identification of variant and invariant parts of the domain and primary abstractions needed for the framework. One domain analysis technique is the Feature-oriented Domain Analysis, short FODA (Kang, Cohen, Hess, Novak, & Peterson, 1990). FODA tries to abstract generic parts of the domain from existing applications by removing all factors that make each application different. Abstraction is done to the point where the product of the domain analysis covers all of the applications. Differences are factored out by generalizing existing parts of applications and aggregating the variant aspects into more generic constructs. As a result FODA delivers the primary invariants of a framework. Variations within the domain are captured by refining common abstractions through specialization and decomposition. Additionally, parameters which capture the variant aspects are defined for each refinement. Finally, the results of the refinement process can form the basis for the framework's hooks.

Software architecture

Perry and Wolf described in their work that all software systems must have a solid foundation at the level of software architecture (Perry & Wolf, 1992). Software architecture studies higher level issues involved in software design, such as the organization of a system, physical distribution of subsystems, performance, and composition of design elements. Software architecture encompasses components, pieces of software, such as object-oriented classes or components, and connectors enabling the software pieces to communicate with each other. Thereby, components are understood as "abstract units of software and internal state that provide a transformation of data via its interfaces". Connectors are "abstract mechanisms that mediate communication, coordination, or cooperation among components" (Fielding, 2000). Multiple variations of organizing components and connectors that can exist within particular software as well as the rules of how they can be arranged have proven to be reasonably efficient and are referred to as *architectural styles*. Examples for architectural styles include *call/return* or *client/server*. Architectural styles support the application development process by providing a general understanding of what can or cannot be built from the framework (Shaw & Garlan, 1996). Furthermore, they provide general rules for the structure of the application extension.

Design Patterns

Design Patterns capture the expertise of object-oriented software developers by describing a solution to a common design problem that has worked in the past in several different applications. The description of the solution encompasses details about benefits and drawbacks of the pattern and perhaps also names alternatives. With respect to frameworks, Design Patterns are particularly useful for designing Hot Spots, the parts of the framework that determine its flexibility. As Design Patterns

have excelled to be a best practice providing reusable solutions to generic problems, Design Patterns can help enhancing the flexibility and extensibility of a framework's Hot Spots and thus of the whole framework (Gamma, Helm, Johnson, & Vlissides, 1995).

Even though Design Patterns are available for a long time, the credit goes to Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides who, known as the *Gang of Four* (GoF), together have fundamentally shaped our today's understanding of Design Patterns. The GoF classified Design Patterns based on two criteria: The first criterion is referred to as the **purpose** with the dimensions *creational, structural, and behavioral*, whereby the second criterion is named **scope** which is further subclassified into *class and object*. (Gamma, Helm, Johnson, & Vlissides, 1995) The graphic given below shows the Design Pattern space after GoF.

Purpose				
Creational		**Structural**		**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Momento Observer State Strategy Visitor

Figure 22: The classification of Design Patterns after GoF. Creational Patterns talk about Object creation, Structural Patterns deal with Object and Class composition while the Behavioral Patterns are concerned with Object interaction and distribution of responsibilities.

Another classification of Design Patterns was done in Froehlich's work on object-oriented frameworks. He identifies three levels of design patterns: Architectural Patterns, Design Patterns in a more specific sense, and Idioms (Froehlich, Hoover, Liu, & Sorenson, 1997).

Architectural Patterns describe a general high-level architecture for applications and are closely related to architectural styles introduced in section 3.5.3 *Domain-specific software architectures*. Many patterns incorporate a particular style rather than defining an own, distinct style. Exposing expertise in software architecture, they identify specific components and connectors that are used to

describe functional relationships between the components. In an object-oriented paradigm, for instance, objects communicate with each other through object methods which, in this particular case, represent the connectors between the objects.

The next level down, in a more specific sense the term **Design Patterns** refers to patterns that do not form the whole basis of a framework, but rather help to structure specific parts within the framework. Thus, they are especially useful for adding flexibility to Hot Spots as they provide methods that have proven of value to expose specific functionality through Hook Methods. The *Singleton Design Pattern*, for example, is applicable when only one instance of a class is needed (McConnell, 2004). It defines a static pointer to an object of the Singleton class with a private constructor and so ensures that the only instance of the Singleton class is the object within the class itself. Furthermore, a Singleton guarantees that the Singleton object is globally accessible as long as the declaration of the Singleton class is globally known. Specific unique components of a framework (in case of a games engine: a 3D engine, an audio engine, or a networking engine) could be exposed through Singleton classes as those parts must be globally available without actually needing an instance of them. Furthermore, having those components as Singletons would significantly improve the ease of use since there is no effort required for globally announcing and managing instances of those components.

The lowest level patterns are called **Idioms**. Idioms tend to be language specific implementations that focus on specific low-level issues. They describe, for example, how exception handling can be provided in a specific programming language.

Open Implementation

Open Implementation is another technique for supporting flexibility within a framework. Software modules incorporating Open Implementation can adapt their internal implementations to serve the needs of different clients (Kiczales, Lamping, Lopes, Maeda, Mendhekar, & Murphy, 1997). Thereby, the modules themselves support different strategies of implementation while providing a strategy control interface that allows clients to choose the implementation strategy that best serves their needs. A files system, for instance, could support several different caching strategies allowing the setting of a particular caching strategy based on a usage profile. Typically, the strategy selection information is provided during the initialization of the module.

Open Implementation can be applied in particular to the framework library as the above mentioned configuration aspect mostly refers to ready-to-run components. Since classes within the framework core are often abstract, Open Implementation is better applied to the framework library. Incorporating Open Implementation within a framework can help applications to tune a library or module for efficiency as well as increasing the suitability for a wider variety of applications. Four different styles of Open Implementation interfaces are distinguished.

Style A describes an Open Implementation approach where no control interface is provided. Thus, the client cannot tune the implementation which results in only one reasonable scenario, when the only implementation available is appropriate. Due to its lacking flexibility, Style A is somewhat similar to a Blackbox component or module described in chapter 2.

In **Style B** the client provides information about how the component will be used, typically through setting parameter values when initializing the component. The component then is responsible for evaluating the provided information and based on that choosing the best implementation. Clients benefit from Open Implementation in Style B through an enhanced ease of use but are also limited as they are not given much control over the component.

In **Style C** the client specifies the implementation strategy to use from a set of given strategies. The component has no control over the selection which lets this approach appear somewhat similar to providing several interchangeable components. Style C is best used, when components lack the ability to select the appropriate strategy through declarative information as it is the case with Style B.

Lastly, **Style D** describes the most abstract Open Implementation approach where the component simply specifies the interface and the client provides the implementation. Related to a Whitebox model, Style D is used best, when a set of possible implementations cannot be anticipated.

Contracts

One of the difficulties regarding frameworks is to ensure that the framework always receives correct information or produces correct results. With the integration of classes or modules being developed independent from each other it often occurs that, even though the class functioned correctly during the isolated testing, it does not deliver the anticipated results after integration into the framework. However, the key problem for generating unintended behavior actually evolves from improper use of the components by the clients of the framework. Passing arguments that are out of scope to a component's function and thus providing input in a way that was not anticipated by the component designers often results in a wrong behavior. To solve this misunderstanding between component designers and component users, contracts provide a way of determining beforehand whether a class or a component used within a certain context generates a correct result (Beugnard, Jézéquel, Plouzeau, & Watkins, 1999). Contracts represent a policy stating the conditions to which a certain service can be performed. Ideally, contracts consist of specifications that tell what the component does without entering into the details of the component's implementation. The component should provide parameters against which the component can be verified and validated, thus clarifying the use conditions between the user and the component. To address the initial question of how to ensure the correctness of the frameworks inputs and outputs, a contract-aware component is able to inform the user of wrong arguments passed to one of its functions by, for example, throwing an exception or returning a status flag.

The concept of contracts also incorporates that contracts can specialize another and can be composed of other contracts. Beugnard et al. in their work identified four classes of contracts: *basic or syntactic, behavioral, synchronization, and quantitative* contracts (Beugnard, Jézéquel, Plouzeau, & Watkins, 1999).

Basic Contracts are often represented by Interface Definition Languages (IDLs) as well as typed object-based or object-oriented languages. Basic contracts define the operations a component can perform, the input as well as the output parameters each component requires, and the possible exceptions that might be raised during operation. At compile time, static type checking ensures that all clients use the component interface properly whereas dynamic type checking is done at runtime. The specification of the object interfaces, no matter whether in IDL or a native programming language, such as Java or C++, ensures that *client* (another component or the user) and *server* (the serving component) can communicate and additionally that they communicate correctly. Using basic contracts, only methods that are explicitly exposed to the client can be called and only parameters that conform to the specified parameter list can be passed as input to the class or component. The bank account sample below depicts the functioning of basic contracts.

```
class BankAccount
{
        public void withdraw( in Money amount )
        {
                balance -= amount;
        }

private Money balance;

}
```

Figure 23: basic contract the method withdraw incorporates simply defines the input parameter amount. The compiler enforces the contract at compile time by ensuring parameters that are passed to withdraw are of type Money.

Behavioral Contracts are a means of specifying the preconditions required by a method or operation and the postcondition that a method ensures (Beugnard, Jézéquel, Plouzeau, & Watkins, 1999). The Specification and enforcement of conditions can help to cut down on the level of defensive or cautious programming needed within a framework as they help reducing the number of errors application developers make when using a framework. For example, a component within the framework library does not have to handle every possible exception in the input if the preconditions for the component's method are clearly identified and enforced. This helps developers of application extensions as they now can comprehend what the framework does and can handle errors which might be best handled by the application rather than the framework. Furthermore, handling only

exceptions that can really occur within the application instead of handling every possible exception within the framework can increase its performance and make it more attractive. The programming language Eiffel, for instance, supports Behavioral Contracts as an integral part of its language specification (Meyer, 1992).

The specification of Behavioral Contracts within a framework helps capturing and exposing design information that might be needed by application developers. This is achieved by making the required information explicit, so that the user does not need to synthesize it from the source code which might not even be possible if the source code of the framework is not available. With respect to the validation of parameters and thus ensuring that the component is provided correct information, a contract-aware component could signal its client that the passed parameters are incorrect by throwing an exception or returning a status flag. This procedure could also be incorporated in cases where a method call on a component might not be appropriate according to the component's status. The extended bank account example in Figure 24 demonstrates this.

```
class BankAccount
{
        public void withdraw( in Money amount )
        {
                balanceTemp = balance;

                Require ( amount > 0 && amount <= balance + overdraftLimit );
                balance -= amount;
                Ensure (balance = balanceTemp – amount);
        }

private Money balance;
private Money overdraftLimit;

}
```

Figure 24: The behavioral contract within the function withdraw requires that amount cannot be larger than balance plus overdraftLimit. Furthermore, it ensures that balance is updated correctly.

Synchronization Contracts specify the global behavior of objects in terms of synchronizations between method calls. Basic and Behavioral Contracts assume that services are atomic and executed without interruption, which is practically not applicable on multi-threaded operating systems. While one object method is being executed, another object method might run concurrently. If the performing of the one method might influence the behavior of the other method, then the result might be corrupted or undefined. As one particular strategy for managing intra-component

concurrency, the *Mutex* or the *Spin Lock* strategy[3] forces all services requested from a component to be called mutually exclusively, ensuring atomicity. Thus, all function calls on an object that currently executes another but synchronized method are delayed and start running after the synchronized method has finished.

Extending the bank account example to support atomicity of methods calls could look as the following:

```
class BankAccount
{
        public synchronized void withdraw( in Money amount );
}
```

Figure 25: The function withdraw incorporates a synchronization strategy that ensures that there is only one withdraw running at the same time.

Quality-of-service contracts quantify the expected behavior by offering the means to negotiate the result value of a method. Statically, Quality-of-service contracts could be specified by enumerating the features the server object does support. Alternatively, more dynamic solutions that negotiate the conditions of the contract between the object client and its sever are possible. Common Quality-of-service parameters are the precision of the result, the maximum response delay, or the result throughput for multi-object answers.

Quick Review - Specific framework development techniques

This section introduced to a selection of specific framework development techniques. Domain *analysis techniques*, such as FODA, have the target to identify the invariant parts of the framework, whereby variations in the domain are captured by refining common abstractions. *Software Architecture* studies higher level issues in software design, such as software architecture or the distribution of physical subsystems. *Design Patterns* are particularly useful for designing Hot Spots. *Open Implementation* enables software to adapt their internal implementation to certain situations. Finally, *contracts* ensure that the framework always received correct information or produces correct results.

3.6. Framework deployment

Once a framework has been implemented, refined and tested to the point where it is considered stable, it is ready to be deployed by application developers (Froehlich, Hoover, Liu, & Sorenson, 1997). But prior to the final release certain deployment related questions evolve and have to be considered.

[3] For more detail on those strategies, please refer to the book "*Programming Windows - The definitive guide to the Win32 API*" (Petzold, 1998)

"What aids for learning the framework are provided"? Sparks et al. suggest three different approaches: Roll-out-sessions can be held during the development of the framework and gradually introduce possible future users to the framework as it develops. Additionally, example applications demonstrate how to solve specific problems by using the framework. Reference documentations, such as manuals or specifications, describe the purpose of the framework, how to use it, and how it is designed (Sparks, Benner, & Faris, 1996).

Further consideration prior to the release must be dedicated to the question "**Will the framework be distributed as source code or as binaries"?** Binaries on the one hand do not allow the framework community to modify the framework, so that they are dependent on the interfaces provided. This eases future framework upgrading efforts since the interfaces will not change, but only the implementations. On the negative side delivering only binaries means that framework users cannot fix any errors in the framework and are forced to find workarounds. Furthermore, developers are not able to examine the source code and so might lack the possibility to understand the framework. On the other hand, delivering the framework's source code instead of binaries would require from the users to first compile the framework source and then link the binaries to their application extensions. Beyond it, the users might not even have access to additional dependencies of the framework and so might not be able to immediately start working with the framework.

Finally, the question "**How will changes to the framework be handled?"** needs to be answered. What effect will bug fixes or upgrades of the framework have on existing applications? Ideally, only the internal implementation changes, whereas interfaces remain the same. In practice, this is rarely the case as introducing new features or adopting new platforms not only requires changes in the implementation, but also changes to the interfaces. Modifications to the framework that result in mandatory modifications to the code of the application extension are referred to as *code breaking changes. Code b*reaking changes symbolize the aspect that without intervention of the application developer the application will not be able to compile with the new framework or possibly not deliver the correct results.

Quick Review - Framework deployment

This section discussed how roll-out session and sample applications can support the learning process of the framework users. Furthermore, the benefits and challenges of distributing the framework as source code or as binaries were revealed. Lastly, it was shown that in case of the framework adopting changes to the problem domain, ideally, the framework does not change its interfaces, but only its internal implementations.

4. Application of framework concepts to the MFC

This section applies the theoretical concepts that were introduced in the previous sections to the Microsoft Foundation classes and analyzes to which degree the MFC conforms to those concepts. Furthermore, this section will provide the foundation for the next section in which the MFC's characteristics with respect to the framework concepts will be assessed. From that, ultimately, a conclusion about maintainability and extensibility development in MFC will be derived.

4.1. Fundamentals of MFC

The **Microsoft Foundation Classes** comprehend a software framework for developers on the Windows platform that exposes parts of the *Windows Application Programming Interface* (**Windows API**) in an object-oriented manner. The Windows API defines a C interface to the functionality and services that the operating system Windows offers to applications running on it (Petzold, 1998). Thus, the Windows API provides application developers with solutions to fundamental problems, such as *Input/Output*, *memory and process management, connectivity via networks* as well as the *rendering of primitives* and the *support for audio* (for more detail, please refer to the excurse about the Windows API). Considering the variety of solutions to generic problems that the Windows API addresses, it reflects a massive and complex framework for the creation of application extensions, in this case Windows executables, of any kind. Applying the terminology of the framework world, the Windows API can be considered as a large set of hook methods that developers can use to access the functionality the framework Windows offers to its application extensions.

Excurse - Functionality grouping of the Windows API (Petzold, 1998)

Base Services

Provide access to the fundamental resources available to a Windows system. Included are things like file systems, devices, processes and threads, and error handling.

Advanced Services

Provide access to functionality that is an addition on the kernel. Included are things like the Windows registry, the shutdown and restart of the system as well as the starting, stopping, and creating of Windows services.

Graphics Device Interface

Provides the functionality for outputting graphical content to monitors, printers and other output devices.

User Interface

Provides the functionality to create and manage screen windows and most basic controls, such as buttons and scrollbars, receive mouse and keyboard input, and other functionality associated with the GUI part of Windows.

Common Dialog Box Library

Provides applications with standard dialog boxes for opening and saving files, choosing color and font, etc.

Common Control Library

Gives applications access to some advanced controls provided by the operating system. These include things like status bars, progress bars, toolbars and tabs.

Windows Shell

Component of the Windows API that allows applications to access the functionality provided by the operating system shell, as well as change and enhance it.

Network Services

Give access to the various networking capabilities of the operating system. Its sub-components include NetBIOS, Winsock, NetDDE, RPC and many others.

The MFC: The concept of a thin wrapper

The Microsoft Foundation Classes wrap portions of the Windows API while keeping the layer of abstraction as thin as possible (Kuglinski, 2001). As can be found on the Microsoft Developer Network (MSDN), the general design philosophy behind keeping MFC a thin wrapper is depicted in the following summary.

Key considerations behind the "Thin Wrapper" concept

- Significant reduction in the effort to write an application for Windows
- Execution speed comparable to that of the C-language API
- Minimum code size overhead
- Ability to call any Windows C function directly
- Easier conversion of existing C applications to C++
- Ability to leverage from the existing base of C-language Windows programming experience
- Easier use of the Windows API with C++ than with C

- Easier to use yet powerful abstractions of complicated features such as ActiveX controls, database support, printing, toolbars, and status bars
- True Windows API for C++ that effectively uses C++ language features

The overall target of the MFC is to provide a more elegant access to the functionality the operating system Windows offers while ensuring that the performance does not suffer in a measure that is to be evaluated as significant.

The evolution of the MFC

The MFC was introduced in 1992 with Microsoft's C/C++ 7.0 compiler for use with 16-bit versions of Windows as an extremely thin object-oriented C++ wrapper for the Windows API. C++ was just beginning to replace C for the development of commercial application software as predominant way to interface the API. With Windows 95, MFC version 4.0 for the first time was shipped as a *Dynamic Link Library* (DLL) included in the Windows 95 release and thus started being a component of Windows that could easily be upgraded without having to upgrade the whole operating system as well. Later, MFC evolved to version 4.2 and was released with Microsoft's Windows 98 operating system. The latest version of the MFC was launched with Microsoft's Visual Studio 2008 in November 2007 and carries the version number 9.0. Figure 26 provides an overview of the history of the MFC.

Product version	MFC version
Microsoft C/C++ 7.0	MFC 1.0
Visual C++ 1.0	MFC 2.0
Visual C++ 1.5	MFC 2.5
Visual C++ 2.0	MFC 3.0
Visual C++ 2.1	MFC 3.1
Visual C++ 2.2	MFC 3.2
Visual C++ 4.0	MFC 4.0 (mfc40.dll included with Windows 95)
Visual C++ 4.1	MFC 4.1
Visual C++ 4.2	MFC 4.2 (mfc42. dll included with the Windows 98 original release)
eMbedded Visual C++ 3.0	MFC 4.2 (mfc42.dll)
Visual C++ 5.0	MFC 4.21 (mfc42.dll)
Visual C++ 6.0	MFC 6.0 (mfc42.dll)
Visual C++ .NET 2002	MFC 7.0 (mfc70.dll)
Visual C++ .NET 2003	MFC 7.1 (mfc71.dll)
Visual C++ 2005	MFC 8.0 (mfc80.dll)
Visual C++ 2008	MFC 9.0 (mfc90.dll)

Figure 26: The evolution of the Microsoft Foundation Classes (MFC). With version 4.0, MFC started being distributed as a Dynamic Link Library in Windows 95.

Quick Review – Fundamentals of MFC

The MFC comprehends a software framework for developers on the Windows platform that exposes parts of the *Windows Application Programming Interface* in an object-oriented manner. Thereby, the MFC tries to keep the layer of abstraction as thin as possible.

4.2. Domain analysis of the MFC

In section 2.5 *"Framework domains",* depending on the area they are applicable to, frameworks were classified into *application frameworks*, *domain frameworks*, and *support frameworks*. **Application frameworks** encompass functionality that can be applied horizontally across problem domains, **Domain frameworks** offer vertical functionality that is common to problems within one particular problem domain, and **Support frameworks** provide applications and other frameworks with low-level functionality.

Before being able to assign the MFC a membership of one of the three above mentioned framework groups, it is necessary to identify the actual **problem domain** the MFC addresses. To add a structure to the various parts of the MFC's problem domain, further segmentation needs to be done by introducing a first level and a second level problem domain (see figure 27). Since the MFC is a framework that aims at exclusively supporting the application development on the Windows platform, the first level problem domain can be referred to as *"Windows Application Development"* For obvious reasons other operating systems, such as Linux, Unix, or Mac OS are not supported. The second level problem domains are extracted from the excurse about the Windows API ("*Windows API: The functionality grouping*") and arranged below the first level problem domain. Figure 27 depicts condensed and arranged data retrieved from the excurse about the Windows API and identifies the MFC's second level problem domains with respect to the first level problem domain:

1. Level	Windows Application Development	
2. Level	**Low-level services** - File system - Threading - Memory - Windows services	**User interfaces** - Controls, buttons, windows - Mouse, keyboard - Common dialogs - Common controls
	Networking - Internet - Sockets	**Graphics** - Output to monitor, printer, and other devices

Figure 27: The MFC's first level problem domain Windows Application Development represents an abstraction of the second level problem domains low-level services, user interfaces, networking, and graphics.

Even though the second level problem domains could theoretically also be broken down into third level problem domains, the resulting granularity would be small enough to instead consider the entities of the third-level problem domains as concrete problems. Breaking down *"networking"*, for example, would yield internet *connectivity* or *socket management*. For this reason, the following consideration will be limited to the described problem domains above.

Stating the aspect that the MFC is applicable to the development on the Windows platform only and thus implicitly rejects the support of other operating systems, such as Linux, Unix, or Mac OS, the MFC can be clearly identified as a **domain framework**. Thereby, all problems the MFC provides solutions for lie vertically within the first level domain *Windows Application Development* (please refer to Figure 28).

Figure 28: The MFC spans across several problem domains and therefore can be clearly characterized as a domain framework.

In addition to the domain-oriented aspects the MFC incorporates, it also shows **application framework**-related qualities that lie within the *Windows Application Development* domain. As already indicated in Figure 28, the MFC spans across the fields *user interface development*, *graphics programming*, *networking,* and even offers low-level functionality. Figure 29 gives a graphical representation of the in this paragraph mentioned *application framework* property of the MFC.

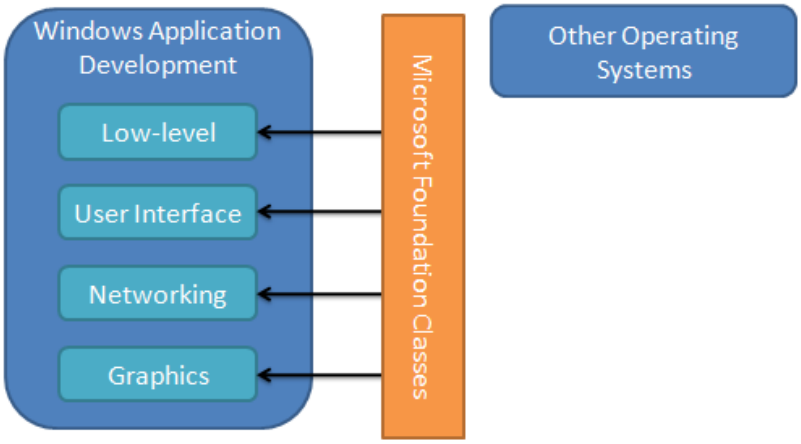

Figure 29: The MFC is limited to the application development on the Windows platform and therefore is also considerable as an application framework.

The third perspective on the MFC that can be argued for is that it ultimately shows aspects of a **support framework** as well. *File system support*, *process handling and threading* as well as *memory management* is functionality that is typically settled within support frameworks. Incorporating such basic functionalities also add the characteristic of being a support framework to the MFC.

As this section tried to show, the Microsoft Foundation Classes, due to their complexity, cannot be clearly categorized as either domain, application, or support framework. The MFC rather shows characteristics of all three framework types.

Quick Review – Domain analysis of the MFC

It was shown that the MFC embodies characteristics of a domain framework, an application framework as well as a support framework.

4.3. Architectural concepts of the MFC

As described earlier in section *2.2 "Framework Concepts"*, a framework consist of a framework core and a framework library. The framework core provides the hooks to customize the framework to fit individual requirements and thus, accordingly, must show a high degree of flexibility. Usually, this flexibility is achieved through abstraction. It was also shown that as the degree of abstraction and therefore the degree of flexibility increases, a framework's degree of completeness decreases. Thereby, the relation between flexibility and completeness is simply stated: A framework that abstracts moves away from a specific solution to a problem and provides a more general solution that can be applied to multiple similar problems. Contrarily, a framework that is complete and thus has a specific solution for multiple problems is tightly bound to these problems without having any possibility to also fit problems for which no specific solution was generated. To solve this flexibility-completeness dilemma, frameworks usually are split into two parts: the *framework core* and the *framework library*. Thereby, the *framework core* serves the flexibility requirements by providing hook methods to the framework's Hot Spots. The framework library on the other hand typically encompasses instantly usable specializations of the framework core's abstractions and thus customizes them to fit more specific and concrete problems.

The documentation of the MFC on the *Microsoft Developer Network (MSDN)* does not explicitly distinguish the terminology of *framework core* and *framework library*, but rather uses both terms as equivalents. From a scientific point of view, this is inadequate and incorporates further confusion when distinguishing **Hot Spots** (places within the framework that can differ amongst applications and therefore require customization) from **Frozen Spots** (places within the framework that are invariant for all applications). Reading the MFC documentation, it is not easy to understand which classes of the MFC are meant to be used as ready-made support components, such as an array or a list container, without requiring any customization and which classes are parts of the MFC's architecture, thus requiring customization. Section 6.2 *Documentation on the MSDN* will discuss this topic in detail

and provide a proposal for how to categorize the classes within the MFC measured on the aspects and terminology that was introduced in section 2.2 “Framework concepts”. It will also show that a documentation following the topology that is based on the terminology introduced in chapter two and three could improve the understanding of the user of what can or cannot be done with a certain class.

The MFC framework core

The MFC was originally designed as a framework that intended to ease the development of document processing application on the Windows platform. Thus, the so called *document/view architecture* with its *Single Document Interface* (SDI) and its *Multi Document Interface* (MDI) not only represents an integral part, but with respect to framework terms instantiates the **framework core** of the MFC. A famous example for an SDI application is the *Windows WordPad* originally written by Jim Springfield. The WordPad is a simple text editor with print, format, and save/load functionality that restricts the user to edit only one document at a time. If the user wants to load a document other than that he is currently working on, he is asked to first save and close the open document before the new document can be displayed. This procedure ensures that the Single Document restriction is reinforced. An example for an MDI application is *Microsoft’s Excel* which covers the domain of spreadsheet and table processing. Here, the user can have multiple spreadsheets open next to each other and even let various spreadsheets interact. This is especially useful when data from one spreadsheet is processed in and written back to another spreadsheet. Figure 30 summarizes the *Document/View* architecture and gives a graphical representation.

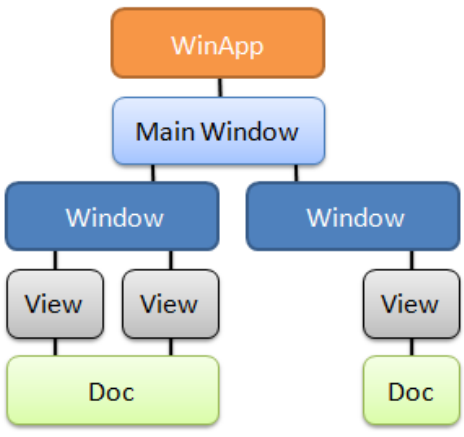

Figure 30: General architecture of an MFC application: Every application has a WinApp and some sort of main window. Attached to this main window several client windows can exist. Client windows (the main window can also be a client window) can be associated with views which present documents.

Document/view applications contain one or more sets of documents, views, and frame windows. Thereby, the *documents* manage an application's data including serialization[4] and printing, *views* display data and accept user input while *frame windows* host one or several views. Additionally, all MFC applications at least have one application object derived from *CWinApp* and one main window object often indirectly derived from a class called *CWnd* through an instance of another *CWnd* incorporating class.

Even though the core of the MFC is based on the document/view architecture, MFC applications are not restricted to it. Other types of applications, such as dialog-based, form-based, or even console-based are also possible with them using only certain parts of the document/view architectural features.

The **MFC core** provides *hooks* to application extensions through classes with virtual overridable methods. An example for this is the class *CWinApp* that is part of every MFC application. In the first step, a new class that extends the CWinApp and overrides its member functions respectively must be created. In order to specify the start-up behavior of an MFC application, clients need to extend the member function *InitInstance*(). However, if no functions are overridden by the client, the default implementations are used. This means that in MFC variation is dealt with by enabling the MFC client application to provide own application-specific functionality by placing code in derivatives of the MFC's virtual functions. Therefore, MFC is not based on instantiating objects from the MFC framework library and passing back the parameterized objects to the MFC accordingly. In section *2.4 "Classification of Frameworks"* the concept of architectural-driven (Whitebox) vs. data-driven (Blackbox) frameworks was introduced. Thereby, the essential quintessence was that Whitebox frameworks deal with variation through inheritance from existing classes of the framework core while Blackbox frameworks use composition of components to achieve the desired customization. From this aspect it seems more appropriate to categorize MFC as an architectural-driven framework rather than a data-driven one. The particular hooks referring to the Hot Spots within the MFC framework are called event handlers and will be covered in section 4.2.3 *"Event processing in MFC"*.

The MFC framework library

As covered in section 2.3 *Characteristics of frameworks*, the framework core usually fulfills the requirements of being flexible enough to serve solutions to as many problems as possible. The framework library furthermore attempts to give the framework a relatively high degree of completeness by providing ready-made solutions to reoccurring problems. While the MFC core shows strong characteristics of a *Whitebox* framework, the MFC framework library fits the role of instantiating the aspect of relative completeness. As typical for a framework core, the completeness aspect within the MFC is also only insufficiently developed. The MFC library therefore represents a collection of solutions to reoccurring problems that are related to the Windows development and do not require customization of any kind anymore. This collection encompasses, for example, *standard*

[4] Serialization is covered in detail later in section 6.8 Design Patterns in MFC

dialogs, such as file dialogs and message boxes, *basic container classes*, such as array, list, or map, *classes for file processing*, *classes for database connectivity*, as well as *Internet and networking classes* (please refer to the MSDN for a detailed listing of all classes).

The MFC Application Wizard

During previous sections the MFC was merely described as an assembly of C/C++ classes that together represent a software framework for development on the Windows platform. In addition to the pure code and binaries that make up the major part of the MFC, Microsoft also provides integration into its development solution Visual Studio. Visual Studio is an *Integrated Development Environment (IDE)* that amongst others manages the file system of an MFC project[5], provides a graphical resource editor for adding dialogs and input forms to the application, and also enables the user to edit the source code of his application extension. When creating a new MFC project, the so called *MFC Application Wizard* asks the user to adjust the standard settings of the application to his needs. Thereby, the user inter alia can choose between creating an SDI or an MDI[6] application, can customize the actual main window that is created, can include database support, and can determine which classes are actually generated. With the generation of classes the IDE supports the developer with the creating new classes and deriving them from the appropriate MFC base classes to finally obtain the skeleton of the MFC application extension. The skeleton now can be used to add new hook methods and customize existing event handlers.

Event processing in MFC

Windows distributes events to Windows applications through so called Windows messages. Typically, events are triggered on application windows that were registered with the operating system during the initialization phase of the application. If a window event, such as a mouse click, is triggered, Windows first fetches this event globally regardless on which application the event has been triggered. Thus, in order to receive notice of an event it does not matter for Windows, whether a mouse click occurred within an application or on the Windows shell. Technically, an application can even poll Windows for messages that are sent to other applications. This process is referred to as Windows API hooking (Richter & Nasarre, 2007). In the second step, Windows identifies the application window the event was triggered on and then forwards the message to the appropriate application that owns the window. The application receives the message in its **message pump**, a non-deterministic repetition instruction that frequently polls the operating system for new events (please see Figure 31 for a working of scheme of the *message pump*). The *message pump* represents the initial point of event processing and is the point where application functionality is connected to events.

[5] MFC application, or more generally Visual Studio applications, consist of a number of files including source files, resource files, and project files.

[6] For an explanation of what SDI/MDI stands for, please refer to section 4.2.1 The MFC framework core.

```
while( message != QUIT )
{
        message = nextMessage();
        process( message );
}
```

Figure 31: Pseudo code of the message pump of a Windows application.

As the last previous paragraph gave an overview over how events are processed in the Windows API, this paragraph can now detail the MFC event processing process. In MFC events are exposed through so called event functions. Event functions are virtual overridable functions that mostly share the same prefix "On…" in their names. The OnIdle() function, for example, is called by the MFC, when the application is in an idle state and does not have anything to do. Typically, clients place their code in the OnIdle() function if they want any kind of calculation done that can run in background. All event functions within MFC relate to one Windows message that is part of the Windows API. Thereby, the name of the Windows message and the name of the respective MFC event function are kept similar (e.g. Windows API: WM_MOUSEMOVE, MFC: OnMouseMove()). When the operating system sends a message to an MFC application, the application's CWinApp class that was introduced earlier, receives this message and dispatches it to the appropriate message target, typically a class. The process of receiving messages takes place within the MFC's *message pump* which is reflected and encapsulated by the CWinApp member function Run(). But how does the MFC *message pump* know which target the message was designated to and how it can communicate with this target? The mechanism that solves this question is referred to as **Message Mapping**. The concept of *Message Mapping* incorporates that each potential target defines its communication interface with the *message pump* as a table of messages that the target can process and further announces the **event handler** the message is connected to. The MFC's message pump then searches for an entry in the message map of the message target that refers to an *event handler* for the corresponding message. *The event handler* (or event function) is declared as a virtual overridable within the declaration of the *Message Map*, so that finally the overridden event function is called in which the client placed its code to achieve a customized response to the message. Figure 32 provides a graphical representation of the event processing mechanism *Message Mapping* in MFC.

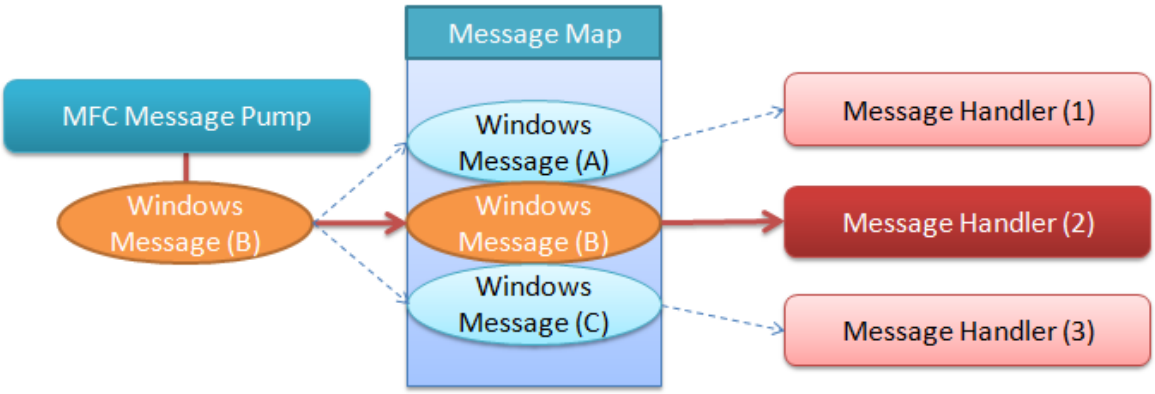

Figure 32: MFC Message Mapping: The MFC message pump receives a Windows Message (B) and searches in the global Message Map for an entry of a Message Handler (2). When found, this message handler is executed.

Conventions in MFC

The Microsoft Foundation Classes abstract certain parts of the Windows API into C++ classes. The general concept of the MFC thereby is to stay a thin wrapper on top of the Windows API while ensuring that the associations the developers connect to the Windows API are also valid within the MFC. The naming of the Windows messages, for example, exactly maps to the naming of the equivalent MFC event functions. While Windows messages start with “WM_” being followed by the actual name of the message, the MFC event functions start with “On” followed by the actual name of the windows message accordingly. This makes it easier for experienced Windows API developers to migrate to and acquaint themselves with the MFC as they can apply the knowledge they have gained on the Windows API to the MFC as well.

Another naming convention in MFC is reflected by the use of **Afx** as the prefix for many functions, macros and the standard pre-compiled header name "*stdafx.h*". During early development what became MFC was called **Application Framework Extensions** and abbreviated **Afx**. The name *Microsoft Foundation Classes* (MFC) was adopted too late in the release cycle to change these references. Afx encourages best coding practices including among other things several types of specialized asserts, memory supervision, etc. The following illustration gives an example for an Afx function in MFC.

```
int AfxMessageBox(
   LPCTSTR lpszText,
   UINT nType = MB_OK,
   UINT nIDHelp = 0
);
```

Figure 33: A function within MFC that shows a simple Message Box dialog. The function name starts with Afx, an old relict from when the early time of MFC, when it was called Application Framework Extensions.

Another quirk about the naming in MFC is that class names are introduced with a **capital C** (*C* stands for *Class*). This especially aims at helping developers to maintain the overview when reading code. It distinguishes MFC classes from other classes and types, such as *int* or *double*, and furthermore makes the information explicit that the class is a part of the MFC and not belonging to any other framework or library.

Quick Review – Architectural concepts of the MFC

It was shown that customization in MFC is achieved through inheritance rather than composition. Furthermore, it was covered that the MFC framework core incorporates the Document/View architecture whereby the MFC framework library provides a certain degree of completeness. Event processing in MFC is based on the Windows API event processing mechanism and exposed in MFC through Message Mapping.

5. Maintainability study of the MFC

5.1. Overview

The previous chapter gave an overview of the MFC, described its relation to the Windows API, and applied the concepts that were introduced earlier in chapter two and three. The MFC was clearly identified as a framework and its properties were categorized depending on their belongingness to *Application*, *Domain*, or *Support framework*. Furthermore, an introduction to the architectural foundation of the MFC was stated as well as a distinction of the MFC into framework core and framework library was conducted.

This last chapter performs a maintainability study of the MFC and therewith in particular addresses the actual main topic of this work. A warrantable question with respect to the overall development cycle of software frameworks is why the maintenance phase is of special interest. A study conducted by the National Bureau of Standards estimated that 60% - 85% of the total software development cost is due to maintenance (Eagle, 1995). Thereby, these numbers are determined mostly by mistakes that were not found during operational testing and thus need to be fixed at the customer's side. Concerning this matter, Capers Jones states in one of his works about software metrics that the later a bug is found, the higher the cost of fixing it (Jones, 2006). Since maintenance amongst other things, such as extension development, also includes bug fixing of products that were already released to the customer, maintenance most widely deals with even those expensive bugs. With this in mind, it seems reasonable to attempt a reduction of potential future maintenance efforts from the very beginning. Referring to this one key aspect certainly is to perform intensive and exhausting testing prior to the release, but even more importantly to design the framework from the very beginning in a way that makes future interventions easier to accomplish. The incorporation of best practices, such as design patterns or object-oriented methodology in general, keeping standards and sticking to guidelines as well as a substantiated documentation can significantly support the creation of better preconditions for future maintenance and extension development projects.

The remainder of this chapter covers, inter alia, how specific architectural decisions that are reflected in the MFC affect the complexity of maintenance and extension development projects. It analyzes how well specific properties and peculiarities of the MFC support efforts that are directed towards maintenance and enhancement. Thereby, regarding the above mentioned, each section covers one particular aspect by first giving an overview over the aspect itself and then drawing the conclusion that evolves from this aspect. In the end, the overall goal of this work is to show that the better the internal structure of the MFC services maintenance and enhancement projects, the less resources will be bound by those projects; a consideration that ultimately impacts the volume of financial investments.

5.2. Documentation on the MSDN

In section 2.3 *Characteristics of Frameworks* the documentation was identified as a success factor that can make a critical contribution to the overall success of the framework. It was shown that *ease of use* can be established by providing a detailed documentation including descriptions of hooks 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 recycle and comprehend the framework's methodology 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 technically zero.

With the *Microsoft Developer Network* (MSDN), Microsoft amongst other things provides a detailed documentation for the MFC framework that is accessible over the internet[7]. For each and every class within the MFC, Microsoft details its purpose and explains its member variables and functions. The documentation of global and member functions includes a description of the functions' parameters, states possible values for flag parameters, and provides information about the return value of the function. Together with remarks to specific aspects of the functions' behavior, the MSDN also illustrates the usage of a function with a paragraph of example code. Inter-class relationships are mentioned and references to relating classes are provided.

A conspicuity regarding the documentation of the MFC on the MSDN is reflected in the fact that it is not exclusively distinguished between framework and library when referring to the MFC. The terms framework and library are rather used as synonyms and at several places the MFC is even referred to as *MFC Class Library*. With respect to the differentiation of both terms that was conducted in section 2.3 *Characteristics of Frameworks*, this represents a gross carelessness. While the flow of control in a framework constantly remains within the framework's range, so that the framework at any time has the power intervene, the flow of control regarding a library is always maintained within the calling program. An example for a real library is the Standard Template Library (STL), since the STL truly contains a set of components and classes that at all times are instantiated with a certain dependency towards the calling code. MFC, however, clearly shows all characteristics that describe a framework, so that referring to the MFC as a library seems inappropriate. Figure 34 depicts the difference between the MFC and the STL:

[7] http://www.msdn.microsoft.com (May 5th, 2008)

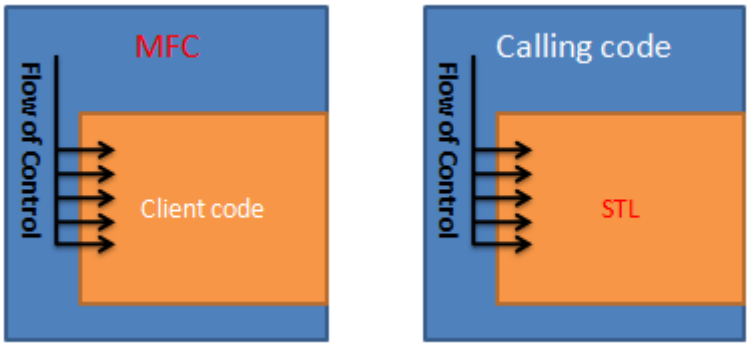

Figure 34: As already shown in chapter 2, the flow of control represents a determining demarcation criterion between a framework and a class library. The MFC, as a framework, permanently maintains the flow of control and interacts with client code through callback methods. The fact that components within the Standard Template Library (STL) are called from outside the STL clearly identifies it as a library.

Concerning the maintainability and extensibility aspect, it was already detailed in previously that a comprehensive and well structured documentation can significantly drive the framework's adoption on the client side. With the MSDN, Microsoft provides a central point of information for the users of the MFC and details almost every aspect that can potentially be of interest. Considering the fact that the MFC is accessible over the internet and hosted on Microsoft servers, Microsoft has the possibility to instantly add or update information and propagate it to the community. In addition to the online version of the MSDN, Microsoft also ships an offline version with every release of its Visual Studio development environment. The big advantage of the online version, however, is that it is permanently maintained and kept up-to-date. Incorporating updates in the offline version is only possible with a new release[8] of the whole Visual Studio.

In addition to the support of the adoption of the MFC on the client side, the MSDN also helps reducing the number of support enquiries from users facing problems. When searching for an answer to a question, users in general choose the fastest option. Since the processing of a support request via email usually takes significantly longer than looking up the information on the MSDN and furthermore the prospect of success due to the complexity of the MSDN is promising, users usually will first try the second option. This positively effects the utilization of the Microsoft support, allows a reduction in resources that need to be allocated for this service, and ultimately saves cost.

5.3. MFC conventions

As described in section 2.3 *Characteristics of frameworks*, 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. Thereby, the individual team members interoperate with each other and it often occurs that one developer works on code that was produced by another developer. 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 comprehended and

[8] A typical release cycle for Visual Studio takes between two and three years.

understood (Cwalina, Abrams, & Ragsdale, 2005). 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 structural pattern. This structural pattern that aims at providing a coding foundation for all developers and thus tries to converge their work-related way of thinking is 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 (Moser, 2003).

Since MFC wraps and exposes certain parts of the Windows API and also includes Windows API data types, it seems reasonable to also take the Windows API naming conventions into consideration. Young revealed in his work the Hungarian Notation as being incorporated across the Windows API (Young, 2003). 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. The following Figure 35 represents an extract of the most common prefixes used within the Hungarian Notation.

Prefix	Interpretation
b or f	A prefix for booleans; f stands for flag.
c	A prefix for characters.
n	A prefix for short integers.
dw	A prefix for double words.
sz	A prefix for null-terminated strings.
P	A prefix for pointers.
lp	A prefix for a long pointer.

Figure 35: An extract from the Hungarian Notation prefixing system.

The Windows API defines its own data types which are usually written with all capital letters and map to the Hungarian Notation scheme (e.g. *LPSTR*, a long pointer to a string or *BOOL*, a Boolean data type). Additionally, Windows constants, such as the Windows message identifier *WM_COMMAND*, always start with a prefix representing the type of the constant. In this case, *WM* stands for "Windows Message" and *COMMAND* identifies the type of the Windows message.

In addition to the Windows API naming conventions that are part of the in MFC used Windows data types MFC also has its own set of naming conventions. Member variables within an MFC class, for example, are preceded with an "*m_*". Event functions that are called on a particular Windows Message carry the prefix "*On*" (e.g. *OnCommand*). In version 9.0 of MFC another convention was introduced. *CMFC* stands for classes within MFC that do not wrap any Windows API, but rather represent independent MFC-specific functionality. The new MS Office Ribbon[9], for example, was

[9] Or officially named Fluent UI is the terminology for a new menu structure in Microsoft's product Office.

completely reimplemented by MFC. Having an independent Ribbon in MFC enables MFC users to develop applications that have the same look and feel as MS Office 2007 and are additionally not bound to the Office Layout restrictions although the Office guidelines are still strongly recommended. To distinguish the new classes that expose MFC-specific functionality, such as the *CMFCRibbonBar*, from classes that merely wrap parts of the Windows API, the new *CMFC* naming convention was added.

With respect to maintenance and extension development of MFC, the used naming conventions certainly help developers to quickly acquaint themselves with existing code. As indicated in the introduction of this section, developers furthermore can make use of experiences and associations they have made during past projects. This not only saves time but also resources and thus makes the maintenance process less complex and cost-intensive.

5.4. Message Mapping vs. Polymorphism

In section 4.2.3 *Event Processing in MFC* an MFC-unique mechanism, referred to as *Message Mapping,* was introduced. *Message Mapping* describes the way MFC sets Windows messages (integer IDs) and Windows message handlers (typically classes) in a relationship, whereby the particular event handlers announce which events they are able to process (please refer to figure 32 in 4.3 for a graphical representation of the MFC *Message Mapping* mechanism). Thereby, the basic idea of *Message Mapping* is that the users of the MFC in their client applications are able to customize the message handling to fit their needs. If no Windows messages were to be processed by the client application, then certainly an easier mechanism that simply assigns an MFC message handler to a Windows message would be more appropriate and also better performing. However, as it is one basic characteristic of every framework to make its functionality customizable in order to fit individual requirements, the mentioned hard-coded mechanism is not applicable. It might, for example, be necessary to route a message that is per default handled by the MFC core to a user-defined message handler. Furthermore does the MFC message pump somehow need to know what to do with a message that is not handled per default by the MFC and where to route it to. Therefore, *Message Mapping* was established as an integral part of the MFC's core activity.

With the introduction of the MFC, which was at that time still known as *Application Framework Extensions (AFX)*, it could evolve as a child of the newly established programming standard C++ and benefit from its object-oriented syntax. One key aspect the object-oriented C++ programming standard enabled was the encapsulation of functionality into classes. As an object-oriented wrapper for the C-based Windows API, this is exactly what MFC does: It summarizes several Windows API function calls that were required to solve a particular problem, such as opening a file, into a class.

Another novelty of the object-oriented programming methodology was *Polymorphism*. Polymorphism describes the concept of having instantiated derivatives (child classes) of base classes addressable through an interface of the base class (typically a pointer or a reference). When a child class derives from a base class which embodies virtual functions, the child class can override the base

class' virtual functions and place its own functionality (for an outline of virtual functions, please refer to *Excurse Abstract Classes* in section 2.1 *Defining Frameworks*). Thereby, a call of a virtual function on an object of a child class through a base class interface results in the corresponding specialization of the child class' virtual function to be invoked. Figure 36 and 37 help to clarify the above explanation:

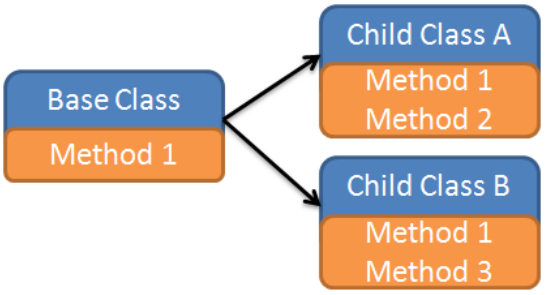

Figure 36: Base Class defines the virtual Method 1. Child Class A and Child Class B derive from the Base Class and thereby override Method 1. In addition to that they define new functions (Method 2, Method 3).

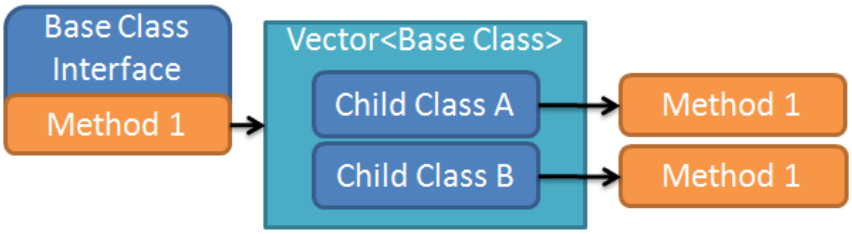

Figure 37: Vector includes objects of Child Class A and Child Class B and addresses them commonly as type Base Class. Through a Base Class Interface call of Method 1 on all objects within Vector the particular specializations of Method 1 are invoked. Thereby, the first call of Method 1 refers to the implementation of Child Class A; the second call to the implementation of Child Class B.

It seems like the concept of *Polymorphism* is able to perform exactly the same the Message Mapping mechanism does. The general idea of Message Mapping, again, is to expose the flexibility to the client application to change the standard routing procedure of the Windows messages to the respective message handlers. The same can be achieved with *Polymorphism*, whereby the MFC message pump would need to automatically attach a certain **virtual message handler** function to a specific Windows message. 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. But what happens to the Windows messages that are not processed by the MFC message pump per default and for which there is hence no virtual message handler declared yet? Those could be accessed by overriding the MFC message pump (CWinApp:Run) and adding a new forward to a message handler function to the already existing *Windows message/message handler* equivalents.

Figure 38 illustrates the described concept:

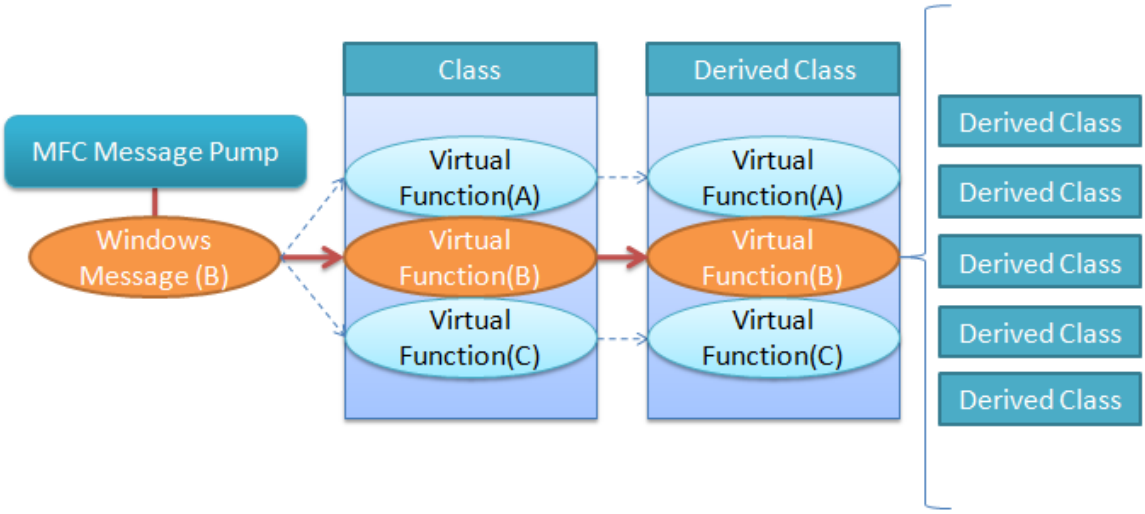

Figure 38: Virtual functions instead of Message Mapping.

Above picture demonstrates an alternative to the MFC mechanism of routing and handling Windows messages that can be achieved by incorporating the concept of *Polymorphism*. The remainder of this section will cover a discussion of possible benefits of a Polymorphism-based approach over the Message Mapping mechanism with impacts of both on the MFC's maintainability will follow next.

Both Message Mapping and the Polymorphism-based alternative are concepts to enable the user to customize the framework's behavior. 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 superfluous 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. In order for the compiler to know which function to invoke on an object method call through a base class interface, it uses a lookup table that is referred to as V-Table (Virtual Function Table). Without keeping track of the actual type of an object that is communicated to through a base class interface the concept of Polymorphism would simply not be realizable. The following Figures 39 and 40 depict how V-Table, classes, and objects relate to each other in *Polymorphism*.

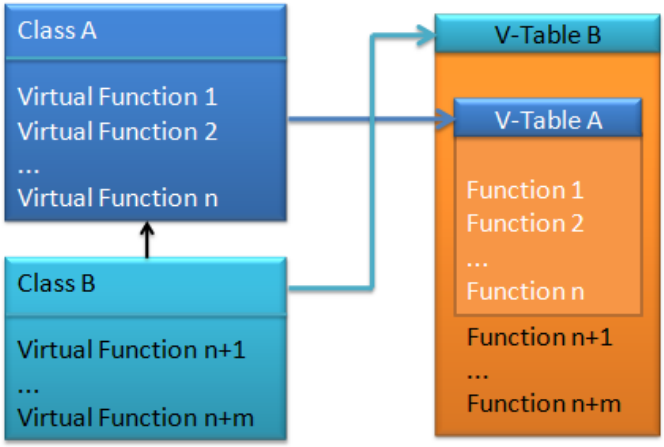

Figure 39: Class A declares a set of virtual functions which are kept for tracking purpose in V-Table A. Class B derives from class A and therewith also inherits class A's V-Table. Additionally, class B declares its own virtual functions which are added to V-Table B.

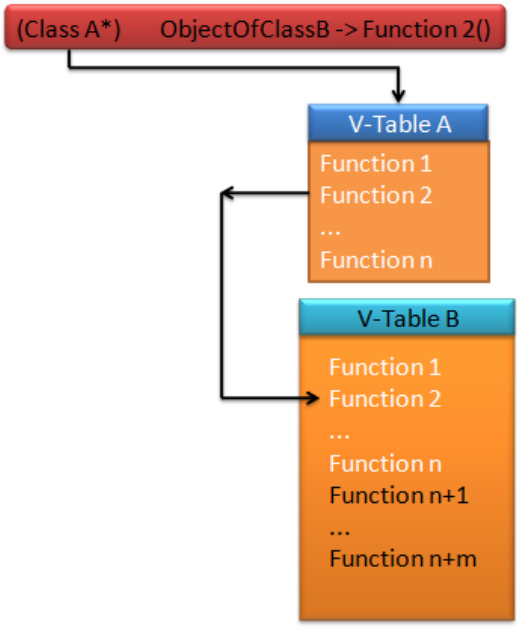

Figure 40: Through the base class interface (Class A*) Function 2 is invoked on an object of Class B. First, the offset of Function 2 within Class A is obtained. This offset is then used to get the address of Function 2 in Class B as both offsets are the same.

Since there are no message map entries to be taken care of, the Polymorphism-based approach would not only improve the usability aspect of the MFC, but also make the code of the application extension 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 would be related to a decrease in resources necessary for maintenance (as there is less code affected) and thus be clearly superior to the Message Mapping mechanism. This also applies to the maintenance of application extensions that are developed from the MFC. Hooking in 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.

Even though *Polymorphism* has the potential to improve the experience of the MFC users, its flipside is a decreased performance. At the time when the initial MFC was developed, the average CPU power was about 100MHz and the average memory was not more than 64 MB. Having huge applications with many classes, the V-Table would quickly gain a size that at this time was considered notable. Furthermore, searching at runtime[10] in a big V-Table for a virtual method's actual implementation could quickly consume a significant amount of CPU power. With the compiler at compile time establishing the link between *Windows message* and *Windows message handler* Message Mapping is absolutely independent of any additional traversing of table entries at runtime and thus potentially performs higher. However, today's computers have gained a level of performance that would easily allow large V-tables without having noticeable impact on the application's performance.

During this section *Polymorphism* was identified as a native concept that could be used to simply hide MFC's message routing mechanism from the MFC users while the compiler ensures that the correct message handlers are called. Thus, using a *Polymorphism*-based approach maintaining and extending applications developed from the MFC as well as the MFC itself could be achieved with considerably less investments.

5.5. Standard conformity of the MFC

In order to realize MFC, Microsoft had to make use of extensions to the C/C++ standard[11] that are specific for the Microsoft compiler. As with many standards, the ISO C/C++ standard also aims at enabling developers to have their product being independent from its base technology (e.g. compiler or operating system). Code that is emancipated from its base technology but is still guaranteed to run on every platform that complies with the standard enables software developers to use their products with a variety of base technology distributions of multiple vendors (Sutter, 2004). In a more concrete sense, this would mean that developers who have written standard C/C++ code are able to compile their code with a standard-conform compiler of their choice (e.g. Microsoft's CL, Intel's ICC, or GNU's G++) and furthermore are able to port their code to any arbitrary platform, such as Windows, Linux, or Unix (Stroustrup & Ellis, 1990).

Since the MFC, by its nature as an object-oriented framework that encapsulates the Windows API, runs on the Windows platform only, it stands to reason that MFC does not support the development of platform-independent code. Furthermore, MFC makes use of language elements that are specific to the Microsoft compiler (e.g. *__declspec*) and not covered by the ISO C/C++ standard. With that, the Microsoft compiler itself is the only compiler that is able to process source code of MFC applications and translate it into machine executable binaries. Other compilers on the Windows

[10] The compiler could also perform the searching of the referring function in the V-Table at compile time and thus reduce the runtime-overhead to a jump to the identified location of the virtual function's implementation.
[11] For an overview of the C/C++ Standard please refer to (Stroustrup & Ellis, 1990)

platform including *Intel's ICC* and *GNU's G++* do not explicitly support MFC applications; a fact that binds the MFC developer community to the Microsoft compiler. With respect to the maintainability and extensibility aspect of the MFC this does not seem to have any impact on the first glance as both the MFC and the Microsoft compiler are products of the same company. In case of novelties in MFC making a modification of the Microsoft compiler necessary, those could be incorporated without any major conflicts. Both the Visual C++ Compiler and MFC are produced by the same team (the Visual C++ Team) and are part of the same product Visual Studio, hence the dependency is well worked out. This would look different, if, for example, the MFC was outsourced and produced by a third-party company, but the Microsoft compiler was still the only compiler being able to compile the MFC. In this theoretical scenario, the third-party company could be limited in its efforts to maintain and extend the MFC due to the Microsoft compiler unit possibly not providing sufficient resources to adopt the changes in their product.

Concerning applications that are developed from the MFC, the maintainability and extensibility aspect is of a more concrete and critical dimension. In case of a specific compiler being superior to competitive products (including Microsoft's compiler) by generating particularly high-performing executables, the MFC application could not benefit from this fact as it is bound to the Microsoft compiler. Furthermore, fictional bugs within the Microsoft compiler could not be by-passed by simply switching to another compiler that is without these bugs. Therefore, from an application developer's perspective, an MFC that works without any Microsoft-specific extensions would be desirable.

This section clearly stated that MFC application developers are dependent on the Microsoft compiler. Even though there are no potential hurdles for other compilers to support MFC, Microsoft provides guarantee only for its in-house compiler to be able to compile the MFC. As long as Microsoft supports and maintains the MFC in its future compiler releases, MFC applications will also benefit from technological novelties. However, if Microsoft in future for any arbitrary reason should decide to discontinue the support of the MFC, application developers would either have to migrate their code to another framework or find workarounds to keep their MFC applications alive. One workaround, for example, could incorporate to concentrate all MFC-related code in one module while translating this module with a compiler version that still supports the MFC and compiling the MFC-independent rest of the code with another newer compiler, which might even be provided by another vendor.

5.6. Default behavior in MFC

As shown in section 2.3 *Characteristics of Frameworks*, one important question with respect to the completeness aspect of a framework is how default behavior is incorporated and exposed. If the user, for example, does not need or want to customize certain abstractions within the MFC, a default implementation that is general enough might save him/her time and trouble. On the other hand, if a default implementation of an abstraction should not be sufficient, he/she could simply override it and provide his/her own customized functionality.

Regarding the MFC framework core, default behavior is incorporated in classes that are intended to be used as base classes (e.g. *CObject::Serialize*) as well as in the virtual hook methods that represent the set of Windows message handlers. Instead of leaving the implementation of the virtual Windows message handlers to the developers of the application extensions, the MFC provides a default implementation for every virtual method. The class *CWnd*, for example, provides the basic functionality of all window classes in the MFC and defines a Windows message handler called *OnQueryEndSession*. When Windows sends the Windows message *WM_QUERYENDSESSION* to an MFC application, the MFC core maps this to the event function *OnQueryEndSession* that indicates that the application was requested to shut down. Per default, the MFC reacts on the *WM_QUERYENDSESSION* with a dialog asking the user to save all open documents before the application is shut down[12].

Not only the MFC framework core, but also the MFC framework library incorporates the concept of completeness by providing default implementations for the abstractions. Furthermore, global functions of the framework library, such as *AfxMessageBox* (a function that invokes a simple dialog) define default arguments in their parameter list where feasible. Besides the text the user has to provide to *AfxMessageBox*, he can optionally specify the visual style of the dialog and another third less frequently used parameter.

Another organ in MFC that incorporates and exposes default behavior is the *MFC Application Wizard* that was introduced in section *4.3*. When creating a new MFC project in Microsoft's Visual Studio, the *Application Wizard* asks the user to adjust the default settings the MFC application will be created with. While without the *Application Wizard* the user would have to trouble with customizing the main window of his application, the Wizard per default defines a window style, but leaves it open to the user to define another style. By default, the Application Wizard generates the classes for an MDI application with a context menu and printer support; all features that the user would have to manually customize his MFC application for.

The decision to which extend default behavior is incorporated in a framework can significantly affect the intensity at which future efforts need to be driven for maintenance and extensibility. Frameworks that merely declare the virtual hook methods but do not define any default implementation obviously require less coding and therefore less resources. Even though such frameworks show an extremely low completeness aspect, they are easier to develop and thus have a drastically shortened development time. However, they are also harder to use and as they do not provide any ready-made solutions for reoccurring situations, the framework users have to spend a significant amount of time on implementing the framework's virtual hooks. It seems even justifiable to say that the time the framework users need to invest for implementing the basic behavior of the

[12] The dialog blocks the application from shutting down as long as the user does not confirm to save the documents. With Windows Vista, a new feature called Restart Manager was introduced. This means that the OnQueryEndSession message must not be blocked anymore and should only be used to indicate the operating system that it understands it is going to be shut down soon. Adoption of this in future MFC versions will be required.

framework's hooks equals the time that the framework developers saved by not providing a default implementation[13].

With respect to the MFC's maintainability and extensibility, the above discussion indicates that maintenance work, such as adding new hooks or modifying existing default behavior, requires an increased amount of resources as compared to a fictional MFC without default implementations. In addition to the actual coding work that is necessary, even more time is to be spent on finding out what kind of implementation is general enough to fit the most common situations while still standing a meaningful default behavior. On the other hand, default behavior exposed as default arguments of functions within MFC can potentially support the introduction of new function parameters without breaking the code of existing applications. A defaulted parameter could thus be added to the parameter list of an existing function, whereby existing applications would simply "ignore" this new parameter as an explicit parameter. The compiler could then ensure that, transparent for the existing application, the default value for the new parameter is passed to the new version of the referring MFC function and, again transparent for the application, either be interpreted to invoke a modified default behavior or withdrawn.

5.7. Contractual behavior in MFC

One of the main problems when working with frameworks, such as MFC, is that, provided the inputs to a function or component, the framework users don't often 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. Section 3.5.6 *Contracts* addressed this topic and introduced a more intuitive way for framework users that helps them to better understand what a component can do and what this component is not capable of doing. It was shown that contracts aim at clarifying the interaction between a component and a user by explicitly stating the conditions to which a certain service can be performed. This section will cover how contracts can help reducing the effort required for maintenance and extension development of software frameworks and in particular for the MFC. It will be discussed to which extend MFC incorporates aspects of the in section 3.5.6 introduced *basic* contracts, *behavioral* contracts, *synchronization* contracts, and *quality-of-service* contracts.

When the first version of MFC was released in 1992, the concept of contracts was still fairly new[14]. This might be one of the reasons, why MFC's contractual behavior is rather limited than sophisticated; however MFC incorporates some **basic contractual behavior**. Basic contracts specify the *operations a component can perform*, the *input/output parameters* each component requires as well as the *possible exceptions* that might be raised during operation. During development, Microsoft's Visual Studio as integral part of the MFC framework provides information about the parameter list of MFC functions are displayed in a small dialog box when hovering the cursor over

[13] This equation only works when the behavior that the framework users implement is somehow similar to the arbitrary default behavior the framework developers did not implement.

[14] The programming language Eiffel as one of the first programming languages having contracts as integrated part of the language was developed 1985 by Bertrand Meyer and his company Interactive System Engineering Inc.

MFC code. The dialog displays the type of the function's return value, the required parameters of the function as well as optional parameters and their default values. At compile time, the compiler checks whether the number and type of the parameters the user has called an MFC function with map to its declaration and thus represents the organ that ensures that the contract is fulfilled. Together with a complete documentation of every individual MFC function, the *Microsoft Developer Network* (MSDN) also provides information about a function's input parameters as well as its output. One important part of the specification of basic contracts, however, is not explicitly stated in the function declarations of the MFC: The specification of possible runtime exceptions a function can throw. Since runtime exception in MFC are often thrown by a set of macros, the in the MFC contained functions themselves do not even declare exceptions that could potentially be thrown. Therefore, in many cases, if an MFC framework method was called with invalid arguments, it will indicate this at the latest possible convenience: at runtime. The MSDN, however, at least offers a little information regarding exceptions in MFC functions: It explicitly states the fact that a function does not throw any exception, if this is the case.

Further basic contractual behavior is achieved through certain compiler macros that check, whether the version of the underlying operating system Windows actually supports the build of the MFC application. Features of newer Windows versions[15], for example, that are also exposed in MFC might not be available on older Windows platforms and thus would also not be accessible through MFC. For that reason above mentioned macros ensure that the MFC features used in the application extension fit the operating system it is built on.

Behavioral contracts were identified in section 3.5.6 as a means of specifying the preconditions required by a method or operation and the postcondition that a method ensures. Thereby, the specification and enforcement of conditions can help to cut down on the level of defensive or cautious programming needed within a framework as they help reducing the number of errors application developers make when using a framework. In a more practical context, this could mean that parameters being passed to a function are checked for validity at runtime. In case of the parameters not complying with the behavioral contract, an error message could be communicated to the user of the application extension. Concerning MFC, this kind of contractual behavior is incorporated by the macros *VERIFY, ENSURE*, and *ASSERT.* Each of these macros thereby checks its parameter for *zero* or *non-zero*, which in a Boolean context maps to *false* or *true*. If the evaluation of the parameter results in a value of zero (or false respectively), the 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. Considering the fact that in many cases MFC exceptions are invoked by *VERIFY*, *ENSURE*, or *ASSERT* from within (virtual) MFC functions, this exception handling mechanisms can trivially be bypassed by just overriding and reimplementing this function accordingly

[15] At the time this document was authored, the latest available Windows distribution carried version number 6, marketing name Vista.

(see Figure 41). Ultimately, this means that child classes that override virtual functions lose the base class functions' binding to their contracts. At this place, a mechanism for not only inheriting functionality or interfaces from base classes but also contracts could help solve this problem and enable a stricter contractual policy in MFC.

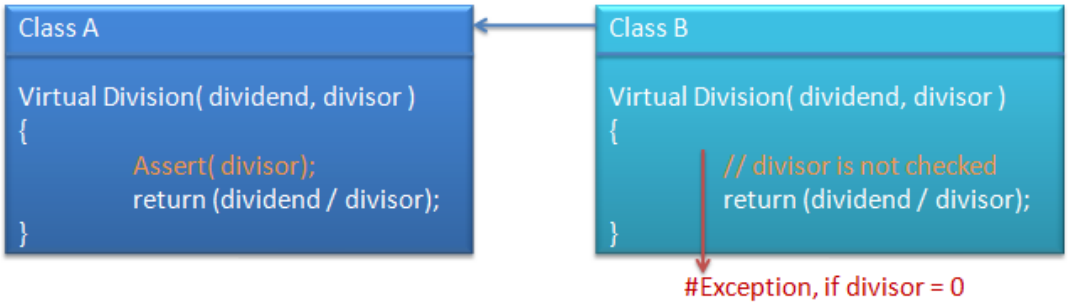

Figure 41: Class A defines a virtual function Division and assigns it a contract which ensures that the divisor is different from zero (Assert). Class B overrides A's Division function, but does not automatically inherit its contract. Since A's contract is circumvented in B, divisor is not checked for zero which might ultimately result in an exception.

Another form of contractual behavior is reflected by the concept of **Synchronization Contracts**. Synchronization contracts, also introduced in section 3.5.6, specify the global behavior of objects in terms of synchronizations between method calls. If two methods operate on one resource (e.g. reading and writing to the same memory block), inconsistencies might occur, when one method reads from the resource while the other method concurrently writes to it. To deal with such problems, MFC provides synchronization functionality in form of a *Critical Section* class (*CCriticalSection*) and a *Mutex class* (*CMutex*). If a function needs to be executed exclusively on a certain resource, means with only one instance of this function running at the same time, the user can achieve this by locking a critical section (*CCriticalSection::Lock*). If now another instance of this function requests access to the same resource, it needs to do this through the critical section object, which will communicate the second function that the resource is currently locked. The second function will then have to wait until the first function unlocks the critical section and thus frees the resource. The following Figure 42 and 43 help clarifying the necessity for synchronization mechanisms and depict how *Synchronization Contracts* are incorporated in MFC using CCriticalSection:

Figure 42: CCustomClass defines a resource and two functions operating in the resource. When two threads simultaneously operate on an object of class CCustomClass, inconsistencies might occur as one thread might be reading while another one is writing.

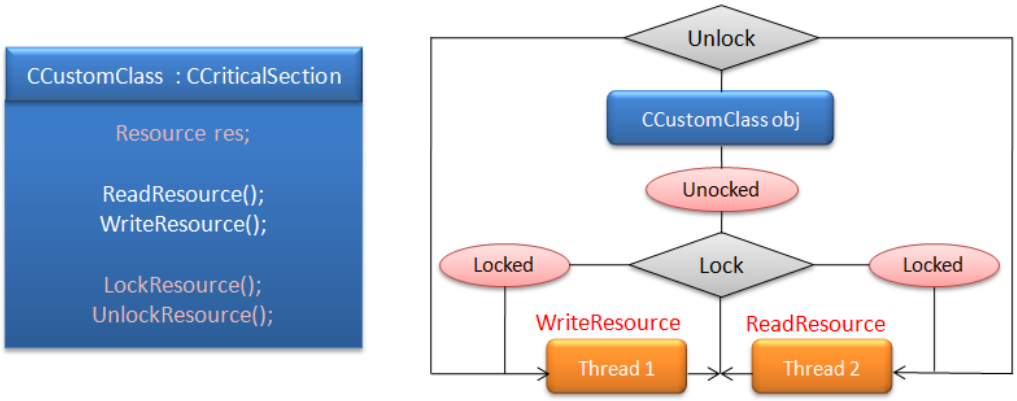

Figure 43: Synchronization contracts in MFC can be achieved by extending custom classes from CCriticalSection. Before a thread (Thread1) can operate on the resource of obj, it has to lock this resource. If the resource is already locked by another thread (Thread2), Thread1 will be kept idle until the resource is unlocked again by Thread2. This ensures that concurrent access operations do not result in inconsistencies.

To summarize MFC's contractual behavior, it is assumable that by providing a more intuitive description of which inputs an MFC function can process and which outputs it will generate, contracts could significantly increase the MFC users' understanding of the framework. Instead of looking up a class' or function's documentation on the MSDN in order to find out which values for a parameter will trigger which behavior, the MFC users could simply read the contract and understand from the source code how to provide the individual parameters to a function. Maintenance and work for fixing hooks that are often used in a wrong way by, for example, providing wrong parameters to a function could be significantly reduced and help MFC to become more stable. By guaranteeing a result given a certain set of inputs, contracts also could increase the trust in the framework and thus reduce support enquiries from users that trouble to use certain parts of the framework. Application extensions built from the MFC could be structured less defensively since MFC would take over the validation of the passed parameters. Furthermore, contracts could help lowering the requirements for documentation and allow a less comprehensive documentation on the MSDN.

5.8. Design Patterns in MFC

Section 3.5 covered how Design Patterns provide solutions to reoccurring software design problems that have proven to work in practice. It was discussed how Design Patterns can support the development of software frameworks by improving the flexibility and enhancability of a framework's Hot Spots and thus ultimately increase its success. Furthermore, section 3.5.4 covered the classification of Design Patterns after GoF into the dimensions *creational, structural, and behavioral*. The remainder of this section will identify an example within MFC for one Design Pattern of each class and is based on an article by T. Kulathu Sarma on *CodeProject*[16].

[16] http://www.codeproject.com (May 5th, 2008)

Creational dimension: Singleton Pattern

The Singleton Design Pattern is used, when only one instance of an object is needed throughout the whole program. Thereby, the *Singleton Pattern* also ensures that this object is accessible from everywhere within the code. Section 4.2.1 has mentioned that every MFC application has one basic application object, an instance of a class derived from *CWinApp*. *CWinApp* encapsulates the central application entry and exit point, the points when the operating system passes over the flow of control to the application and when the application returns the flow of control back to the operating system. Additionally, the *CWinApp*, by receiving and distributing Windows messages, represents the central place of communication between the operating system and the MFC application. As it does only rarely make sense to have more than one application object, a mechanism within MFC that ensures that there is only one application object at a time would be desirable. Therefore, *CWinApp* was designed as a Singleton and ensures that the singularity property is also propagated to derivatives of CWinApp. The Singleton attribute of being globally accessible within an MFC application is incorporated by a global function named *AfxGetApp*. Figure 44 provides a graphical representation of the Singleton Pattern in MFC.

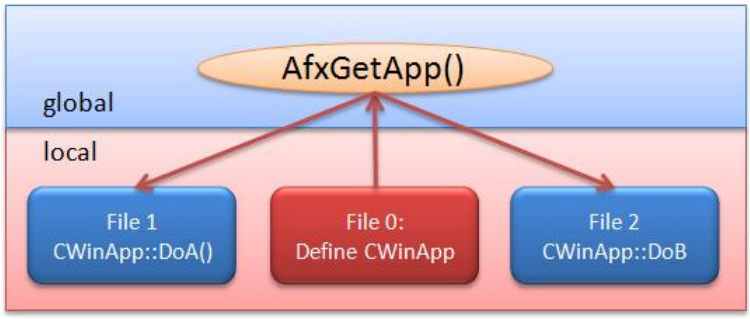

Figure 44: CWinApp is defined locally in File 0 and instantiated only once. Through the global function AfxGetApp() the CWinApp Singleton is available from all other files and modules.

Structural dimension: Bridge Pattern

The *Bridge Pattern* is a Design Pattern which is meant to decouple an abstraction from its implementation, so that the two can vary independently (Gamma, Helm, Johnson, & Vlissides, 1995). When classes including their interfaces and their implementations vary often, the object-oriented programming paradigm becomes especially useful. Changes to the behavior of a program can be easily achieved by extending existing functionality without knowledge about prior implementations being necessary. To also decouple the interface of a class from its implementation, the Bridge Pattern suggests defining the interfacing methodology in one class and the implementation of the functionality that underlies the interface in another class.

To better illustrate the above, the example of a geometric primitive abstraction will be depictured in the following. There are many types of geometric primitives, each with its own properties. One aspect all primitives have in common is that they need to be drawn to the screen which is

implemented by a drawing function within the primitive class. However, drawing graphics to a screen can sometimes be dependent on different graphics adapters and operating systems. For the reason of openness and flexibility across platforms, it might therefore be necessary to enable the primitives to be used across different variations. Having the primitive by itself implement various graphics adapters and operating systems or modifying the class to work with different architectures is not practical and would result in too much interdependent code with respect to future maintenance or extension programming. The bridge helps by allowing the creation of new implementation classes that provide the particular drawing functionality for each operating system or graphics adapter. The abstraction class *primitive* provides methods for getting a primitive's properties, declares the abstractions for the drawing methods, and maintains a reference to the implementations of those abstractions. When a drawing function is called on a primitive, this call is routed to the particular implementation which embodies the system and adapter specific internals. In case of a new operating system or graphics adapter requiring support from the primitive class, then this is simply achieved by adding a new implementation class.

Concerning MFC, the *Bridge Design Pattern* is incorporated in a mechanism to save a document's data to and retrieve it from a persistent storage. This mechanism is referred to as *Serialization* and reflected by the two classes *CArchive* and *CFile*. The *CArchive* class on the one hand provides an interface for writing and reading an object to or from a persistent storage. *CFile* and its subclasses on the other hand provide implementations for different persistent storages, such as disk file (*CFile*), memory (*CMemFile*), or even socket (*CSocket*). Similar to the primitive example, *CArchive* is during its construction configured with an object of base class type *CFile* accordingly. The *CFile* object thereby provides the *CArchive* object with the implementation of the actual read/write mechanisms. Figure 45 illustrates how the Serialization mechanism in MFC incarnates the Bridge Pattern.

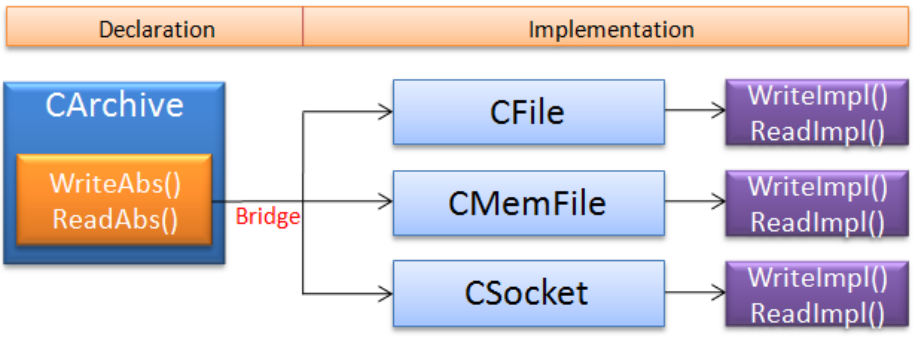

Figure 45: CArchive declares abstractions for the write and read mechanism (WriteAbs/ReadAbs). These abstractions forward to the actual implementations (WriteImpl/ReadImpl) that are kept by the persistent storage classes (CFile, CMemFile, and CSocket). Depending on which persistent storage class the CArchive object was instantiated with, the respective implementation will be utilized.

Behavioral dimension: Observer Pattern

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. 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 merely define a notification function that is called by the subject as soon as a new event occurs. A prominent example for the application of the *Observer Pattern* is a mailing list: Every time an event occurs, a message is sent to the people who subscribed to the list.

MFC embodies the *Observer Pattern* in its *Document/View* architecture (for more detail, please refer to section 4.2 *The MFC framework core*). 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 46) ensures that the modification of a document's data is propagated to all views that were attached to this document.

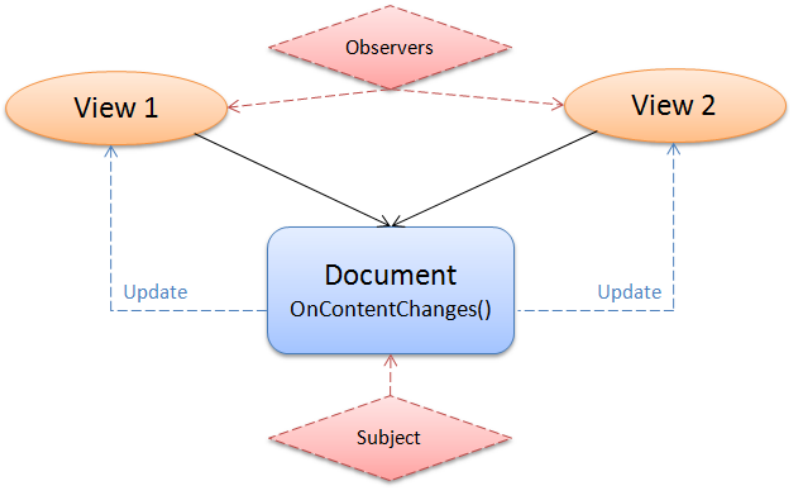

Figure 46: The Observer Pattern in MFC is incorporated by its Document/View architecture. Thereby, the views (observers) register with the document (subject). When the content of a document changes, it will automatically propagate these changes to the registered views and update them.

As already mentioned earlier, Design Patterns represent solutions to reoccurring software design problems that are proven of value in practice. With an enhanced flexibility and enhancability of the framework from the release date on, they decisively can influence later maintenance and extension development efforts. On the one hand, the refinement of Hot Spots becomes significantly easier since components and classes within a framework are better decoupled. On the other hand, an evolving necessity for the correction of bad design decisions becomes less likely, since an effective design decision is already incorporated by the Design Pattern itself. The fact that Design Patterns

were used at various appropriate places in MFC has, with no doubt, contributed to its continuing success as a software development framework on the Windows platform. Certainly, Design Patterns did not play a bit part regarding the reduction of cost and effort required for maintenance and extension development.

5.9. MFC deployment

Once a framework has reached a final state and is ready for release, there is actually only one important aspect left: How will the framework be deployed? Will it be distributed as source code or as binary files? Will there be one big or several small binaries? Considerations related to this topic were already discussed in detail in section 3.6 *Framework deployment*. There, it was shown that both alternatives have certain advantages, but also include certain disadvantages that need to be carefully balanced.

As an integrated part of the *Visual Studio* development environment, MFC is distributed in binary form as a *Static Link Library* (LIB) and a *Dynamic Link Library* (DLL)[17]. In order to make use of MFC, the MFC application needs to include the MFC header files containing the declaration of classes and functions. Furthermore, the application needs to be linked against either the *LIB* at compile time or against the *DLL* at runtime. Even though these two things would be sufficient to build an MFC application, Microsoft also provides the MFC source code with their releases of Visual Studio. This strategy allows Microsoft to straighten out the insufficiencies that are related to the exclusive distribution of binaries and thereby adds the following values to the application of the framework:

1. The developers are able to look into the MFC source code and are not dependent on the MSDN documentation only. Things that are not well enough explained on the MSDN can be reconstructed and understood from the source code. With respect to the maintenance aspect of the MFC, it is imaginable that the number of support enquiries therewith lies below the number of support enquiries of a fictional MFC that does not expose its source code to the public. A possible explanation for this might be that a certain percentage of MFC users try to investigate on the source code first, before approaching the MFC support.

2. In case of the MFC being erroneous, the MFC users are not blocked from fixing the weak spots on their own and do not have to wait for the next release[18]. This allows postponing bug fixes (even for critical bugs) to later releases instead of providing an immediate hot fix. However, the MFC support should at least inform the users how to work around the problem on their own and, if necessary, how to modify and compile the affected parts. Being able to postpone maintenance work provides Microsoft with a higher flexibility regarding the assignment and distribution of resources.

[17] At the time this document was authored, the latest version was version 9.0. In addition to the actual DLL that contains the MFC functionality (*mfc90.dll*), there is also a set of other language DLLs (e.g. *mfc90deu.dll*).
[18] Modified source code, however, is not supported by Microsoft anymore.

Another deployment-related concern that will be discussed in the following is whether the *Dynamic Link Library* should embody the whole MFC or better be split into several smaller *DLLs*. Therefore, a brief understanding of how DLLs work on Windows is necessary. An MFC application that is dynamically linked against a DLL at compile time merely needs to include a header file and a so called *stub library* that contains references to the actual implementations contained in the DLL (Petzold, 1998). When the application is started, a new process is allocated by the operating system and the executable is loaded into the process' address space. In addition to that, the operating system also loads the DLL into the process' address space as it contains the definitions for the referenced functions.

Above explanation could misleadingly result in the conclusion that the operating system does not have the possibility to only load the functionality that is actually used into the process' address space, but instead also loads the functionality that is not used. This would mean that even though only a small part of a DLL is used within an application, the whole DLL of arbitrary size is loaded into the process' address space and therefore into the main memory. Even though this aspect seems to be negligible for appropriately sized DLLs, many instances of the same applications could quickly outnumber the available main memory. To avoid this, Windows was given the ability to determine the start and end offsets in a DLL that contains the actual definition of a function and is referred to as *page*. This mechanism enables Windows to only load the pages that are indeed relevant for the application to be loaded and thus avoid wasting main memory with not used functionality.

Another problem, however, that could not be solved through the above delineated *paging* mechanism, but also relates to a DLL's size is the distribution problem. When bundling executables and DLLs together in one distribution in order to ship the whole package to the clients, there is no way to extract the relevant pages from the DLLs and only include those in the package. Lacking this possibility, there is no other option than providing the whole DLLs together with the executables accepting the fact that the package also contains garbage in form of not needed functionality. However, as the MFC *DLL* at the time this work was authored had a manageable size of about 1.1 MB (in the release version), this consideration only carries authority with respect to smaller application extensions. In case of an MFC application extension, for example, having a size of 9MB, the size of the MFC DLL would only be 10% of the overall size of the package and therefore negligible. This weight increases admittedly in case of the application extension having a size of 1MB only. Then, the MFC would represent 50% of the overall size of the package and thus reserve a considerable amount of storage volume.

A possible solution to this problem could be to split the MFC DLL into several smaller DLLs with each of them grouping a specific functional part of the MFC. This would enable developers of MFC applications to ship their packages with those DLLs only that contain the actually used functionality and thus would significantly reduce the size of the distribution package. On the contrary, having MFC split into several DLLs could easily add confusion to the MFC application developers as they might start mixing DLLs of a newer MFC version with DLLs of older version of the framework. Not only

would this result in not having a consistent MFC as a foundation for their application extensions, but also in conflicts between older and newer DLL versions. One DLL, for example, could be dependent on another DLL, but since the other DLL is of an older version, the dependency could not even be implemented.

Another concern regarding the question whether or not to split up the MFC DLL into several smaller DLLs is related to the migration away from functions and classes that are scheduled to be discontinued in the future towards newer functions and classes. If a new function is added to the MFC that possibly extends an older function and coherently overtakes its role within the framework, Microsoft cannot simply abandon the older function from the next release. Abruptly discontinuing the support of the older function would break the code as well as the binaries of application extensions developed from older versions of the MFC and thus force application developers to adopt their application to the newer framework version. In most cases this is connected to a considerable investment on the application developers' side, even though the application extension might not even benefit from the changes it has to adapt to. A more sensitive approach to discontinue the support of old functionality is to inform the MFC users that an old function will be abandoned within a future release and that they should start to migrate their code towards an alternative. The old function could still be included in the release, but maybe already forward to the function that is meant to be its successor. After the MFC clients have performed the migration towards the newer version, the older version could finally be removed from the release. This scenario would be noticeably harder to realize with MFC being split into several smaller DLLs. What, for example, if MFC was split into two DLLs A and B and one method was deprecated from DLL A? Would the method replacing the old one go to A or B? If it went to B, what would happen if the user was normally deploying A only, would the user now have to deploy both A and B? If it went to the same DLL A, would this make sense even if the functionality grouping per DLL would not be clearly given anymore? To answer those questions and obtain an understanding of the corresponding consequences, future studies will become necessary. Whether or not the MFC will be split into several DLLs will ultimately depend on the demands and feedback from the MFC customer side. Microsoft, however, should always be alert not to precipitously introduce changes that could break the code of their customers or remove an API that is still highly used.

5.10. Summary of investigations

As the introduction already stated, the goal of this maintainability study was to apply framework engineering principles and practices to the MFC framework and inter alia answer the following questions:

1. To which extent does the MFC incorporate the framework engineering principles and practices?

Chapter four and five applied a selection of framework engineering principles and practices to the MFC and discussed how well those were incorporated. It was discussed, for example, that one of the most important success drivers for a framework, the documentation, is provided as highly detailed and always up-to-date online source on the Microsoft Developer Network (MSDN). The MFC incorporates a high pervasion of framework engineering principles and practices including consistent naming conventions, default behavior, Design Patterns, and even a basic contractual behavior (all summarized in the remainder of this section). Addressing above question to which extent the MFC does incorporate framework engineering principles and practices, T. Kulathu Sarma states in his work about Design Patterns: "The success story behind MFC is its class architecture and design principles that were adopted by the developers of the MFC, which makes it one of the most popular and successful frameworks" [19].

2. How can the framework engineering principles and practices help reducing maintenance and extension development cost?

While chapter four gave a general introduction to the MFC, chapter five in particular identified several principles and practices that were included in the MFC's architectural structure. It was shown that the use of coding conventions is well worked out and potentially supports the orientation of developers that have not made any experience with the MFC code base before. The Hungarian Notation consequently pervades the naming of basic data types, such as string, integer, or pointer, while the *Afx* naming convention for functions was used in the earlier versions of MFC. Furthermore, MFC classes that wrap portions of the WinAPI carry the prefix "*C*", classes that represent an independent functionality set within the MFC carry the prefix "*CMFC*". Providing maintenance and extension development teams

[19] http://www.codeproject.com (May 5th, 2008)

with the ability to apply structures and associations they have gained from previous projects to their future tasks certainly eases the process of familiarization with the code and thus saves project time and resources.

Also, the *completeness aspect* that was covered in chapter two is well incorporated by providing default behavior at almost every customizable spot. On top of that, the *Visual Studio IDE* supports the generation of MFC application skeletons, so that a compilable base of an MFC application can be created from a few mouse clicks only. The high degree of completeness the MFC shows saves application developers from implementing abstract Hot Spots that might not even be needed, before they can concentrate their efforts on the actual task-related part of their application. This ultimately allows a higher flexibility regarding the assignment of resources in the project schedule of application developers.

Design Patterns were described in section 3.5 as solutions to reoccurring software design problems that have proven to work in practice. Thereby, Design Patterns can support the development of software frameworks by improving the flexibility and enhancability of a framework's Hot Spots and thus ultimately increases its success. Chapter five identified three Design Patterns in the MFC, namely *Singleton, Bridge, and Observer*. It was shown how those Design Patterns were utilized to realize parts of the MFC's fundamental architecture (e.g. global access of the application object, *Document/View, Serialization*). Above all, the decision to use those Design Patterns has certainly helped reducing the complexity of past and future maintenance projects. Contrarily, the introduction of a home-made and untested design for realizing the *Document/View* architecture or *Serialization* might have been more susceptible to change requests, updates, or refinement, risking larger investments in time and money.

3. Which alternatives exist to the current implementations and what value could they add to the MFC and its customers?

Section 5.4 covered how Windows messages are processed in MFC by a mechanism that is referred to as *Message Mapping*. It was shown that the concept of *Message Mapping* in general tries to achieve what the native C++ language feature *Polymorphism* already incorporates. Since typically the declaration and definition of *Message Maps* are scattered amongst different places in the source code, this mechanism introduces an additional complexity and hence a potential confusion that could be avoided with Polymorphism. Even though a complete replacement of the *Message Mapping* mechanism by a *Polymorphism*-based approach seems to a high

extent unlikely, at least enabling the alternative presented in this work would leave it open to the application developer to decide which option to use. Above all, the *Polymorphism*-based approach would unlock an additional object-oriented linguistical feature of the C/C++ programming standard and add a more modern experience to application developers.

Addressing the question of MFC's standard conformity, it was shown that, even though the linguistical syntax of the MFC fully complies with standard C/C++, the Microsoft compiler (CL) makes use of Microsoft-specific extensions (e.g. __declspec) to compile the MFC code. On the first glance, this seems to result in a dependency of the MFC application developers on Microsoft's CL. As long as Microsoft keeps its compiler up-to-date and exposes novelties of the hardware as well as its operating system Windows, no immediate difficulties for the application developers evolve. If, however, Microsoft for any reason should decide to discontinue the support of the MFC in the future or other compilers should deliver a noticeably better result, MFC application developers could be seriously affected. For this case, section 5.5 introduced a way of how to circumvent the dependency of MFC applications on CL by separating MFC code from code that is independent from CL. So, at least the MFC-independent code could be compiled with any compiler that conforms to standard C/C++.

4. How up-to-date is the MFC and which investments are necessary in order to continue its success?

For a long time, the MFC did not experience any major updates. Not only were novelties to newer Windows versions not accessible from within the MFC, but also MFC applications running on later version, such as Vista, still had the old visual style of Windows 2000/NT. With version 9.0 of Visual C++, Microsoft started to bring the MFC up-to-date and added the Windows Vista look and feel to applications developed from the MFC. An additional investment was done with the acquisition of BCG Soft's Fluent UI equivalent that enabled application developers to add the Office 2007 look and feel to their applications.

One big weak spot that still remains within the MFC is its poor contractual behavior. Section 3.5 and 5.7 in detail discussed how contracts can enhance the communication between the framework and the application developers by stating the conditions to which services or routines in MFC can be performed. Erroneous but legal parameters passed to MFC functions in many cases deliver results that the application developers might not have anticipated. An enhancement in the

contractual behavior of MFC would make the way the framework is used more intuitive and more efficiently add an understanding of what individual parts within the framework can or cannot do. Hence, poorly designed Hot Spots could still maintain a certain degree of usability as long as it is clear how to use them. This could lower the urgency of immediate fixing or make certain maintenance efforts unnecessary at all.

5. How practicable is the introduction of major changes and what effects would they trigger?

Combining the fact that Microsoft's Windows is the widest spread operating system in the desktop area (Decrem, 2003) with the consideration that the MFC was released more than fifteen years ago, might create an understanding for the number and size of software that was produced from the MFC. Introducing changes as long as they do not break the code or the binaries of client applications can be performed without major risk. Conducting critical modifications, however, that would require action on the MFC customer side or might block them from upgrading the framework to a newer version is to be deliberated carefully. Showing a high degree of responsibility for their customers, Microsoft tries to avoid breaking changes wherever possible. Upgrading the MFC to expose new features can be achieved mostly by adding functionality to the existing MFC. In contrast to that, introducing changes to the fundamental architecture or structure of the MFC as a possible outcome of critical bug fixing is often related to breaking changes.

Even though this work discussed a number of modifications to the MFC that would possibly enhance its overall usability and maintainability, the freedom of action to realize those changes is rather limited. The *Polymorphism*-based alternative to the *Message Mapping* mechanism (section 5.4), for example, is imaginable at the utmost as an additional mechanism to *Message Mapping* rather than a substitute. Enhancing the contractual behavior of the MFC (section 5.7) could only be done to the extent that lies within the semantic scope of the language C/C++ and furthermore that does not introduce any breaking changes. Splitting the MFC into several DLLs (section 5.9) could result in confusion regarding the version control of application extensions and additionally increase the maintenance efforts on Microsoft's side as now several DLLs would need to be administered in parallel. The MFC, with its age and maturity is a highly complex framework that due to the number of customers it has gathered over the years has only a limited flexibility with respect to the adoption of innovations and novelties in the framework engineering sector.

6. Conclusion and prospect

While the MFC did not experience any major updates for years, its competitors kept adopting new Windows features and continued to evolve. To make use of novelties in Windows, MFC customers had to incorporate parts of competitive products, such as *CodeProject's WTL* or *Borland's* VCL[20], or even entirely migrated their code away from the MFC towards a more up-to-date foundation. As already indicated in chapter one, Microsoft for a long time concentrated their efforts on the promotion of the *.Net framework*, which was ultimately thought as replacement for the MFC. Referring to that, on the Visual C++ (VC) side, a huge investment was done with the *.Net framework* integration into the C++ syntax. The result, which is commonly referred to as the C++/CLI, allows a mixed development of managed and native code while hiding the conversions between both in the language syntax (Sivakumar, 2007). Even though the C++/CLI was a great success, it should not remain the predominant strategy determinant within Visual C++.

After having recognized that the software world cannot exist of managed code only and that there is still a significant amount of customers outside that build their software in native code, VC's focus has changed once again. With the Visual C++ team redirecting their efforts back to native development, the MFC was given a second spring. Features of the latest Windows version *Vista* are now already partially exposed and customers can add the *Vista* look and feel to their MFC applications by just recompiling their existing sources. After Microsoft had fallen behind Apple regarding style and usability, the effort was strengthened to quickly catch up and generate an attractive and integrated visual frontend that should be similar to all applications being built on the Windows platform. With the Visual C++ 2008 Feature Pack for Visual Studio 2008 released in April 2008, the MFC experienced another major update. The Feature Pack enables the MFC customers to create applications that have the Office 2007 look and feel –the Office Ribbon might be named as a keyword here- and therefore supports Microsoft's initiative to encourage developers to create applications that visually fit each other.

After VC's strategy has changed from managed back to native, the reincarnation of the MFC was finally initiated. The prospect of again becoming an active and highly used framework for the development on the Windows platform makes a reassessment of the MFC's internal architecture and structure inevitable. Therefore, this work conducted a maintainability study trying to evaluate the ease with which the MFC is able to adopt future updates. Many best practices and principles, such as design patterns, conventions, and default behavior were already incorporated or taken into account when the MFC was designed and developed back in the nineties. Furthermore, with the MSDN Microsoft established a gigantic network where developers can find detailed documentation as well as discussion, not only of an MFC-related nature. Albeit the many positive things, there are also certain negative aspects that do not seem to represent the state of the art of framework engineering anymore. With new investments into the MFC framework, Microsoft has the possibility to also

[20] WTL stands for Windows Template Library, VCL stands for Visual Component Library (VCL)

incorporate some of the topics that were outlined in this work: Including a better contractual behavior for new classes, incorporating an alternative for the out-of-date *Message Mapping* mechanism, or maybe even adding alternative ways of deployment could be potential candidates.

Even though Visual C++ contributes only a relatively low part to Microsoft's overall revenue[21], it is an indirect, but important success driver for many Microsoft products. Enabling Windows developers to easier and more efficiently create application extensions for Microsoft software, such as Windows or Office, will support the adoption of new Microsoft products and guide the development of third party software. The more external software is developed from Microsoft technology, the higher the pervasion of those technologies. With the reorientation of VC's efforts towards native development, the future position of the MFC within the Microsoft product family seems to be manifested. Further significant investments are planned with Visual C++ 10; critics that have already attested the death of the MFC will soon have to reposition their opinions. Quoting Bill Dunlap, Business manager of the Visual C++ unit:

"MFC is not dead, it was just a little sick!"
(Bill Dunlap, MS Visual C++)

[21] The biggest branches are Windows and Office (state: 2008).

Table of figures

Bibliography

Adair, D. (1995). *Building Object-Oriented Frameworks.* AIXpert.

Arrango, G., Pietro-Diaz, G., & Pietro-Diaz, R. (1991). *Domain Analysis Concepts and Research Directions.* IEEE Computer Society.

Beugnard, A., Jézéquel, J. M., Plouzeau, N., & Watkins, D. (1999). *Making Components Contract Aware.* IEE Computer Society.

Birrer, A., & Eggenschwiler, T. (1995). *Pattern Generate Architectures: An Experience Report.* Proceeding of ECCOOP 93.

Cwalina, K., Abrams, B., & Ragsdale, S. (2005). *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries.* Addison-Wesley .

Decrem, B. (2003). *DESKTOP LINUX TECHNOLOGY & MARKET OVERVIEW.* Open Source Applications Foundation.

Eagle, D. (1995). *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.

Fielding, R. T. (2000). *Architectural Styles and the design of network-based software architectures.* University of California, Irvine, USA.

Froehlich, G., Hoover, H., Liu, L., & Sorenson, P. (1997). *Designing Object-Oriented Frameworks.* Universtiy of Alberta, Department of Computer Science, Edmonton, Canada.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Gangopadhyay, D., & Mitra, S. (1995). *Understanding Frameworks by Exploration of Examples.* Proceedings of 7th international Workshop on Computer Aided Software Engineering.

Hejlsberg, A., Wiltamuth, S., & Golde, P. (2006). *C# Programming Language.* Addison-Wesley.

Jones, C. (2006). *POSTIVE AND NEGATIVE INNOVATIONS IN SOFTWARE ENGINEERING.*

Kang, K., Cohen, S., Hess, J., Novak, W., & Peterson, A. (1990). *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Carnegie Mellon University, Software Engineering Institute, Pitsburgh, USA .

Kiczales, G., Lamping, J., Lopes, C. V., Maeda, C., Mendhekar, A., & Murphy, G. (1997). *Open Implementation Design Guidelines.* Proceedings of the 19th International Conference on Software Engineering.

Koskimies, K., & Mossenback, H. (1995). *Designing a Framework by Stepwise Generalization.* Proceedings of the 5th European Software Engineering Conference.

Kuglinski, D. J. (2001). *Inside Visual C++.* Microsoft Press.

Lücke, T. (2005). *Frameworks.* University of Hanover, Germany.

McConnell, S. (2004). *Code Complete: A practical Handbook of Software Construction.* Microsoft Press.

McShane, S. L. (2002). *Organisational Behaviour on the Pacific Rim.* McGraw Hill .

Meyer, B. (1992). *Eiffel: The Language.* Prentice-Hall .

Moser, H. (2003). *Auswirkungen von Code Conventions auf Software Wartung und Evolution.*

Nelson, C. (1994). *A Forum for Fitting the Task.* IEE Computer 27.

Perry, D. E., & Wolf, A. L. (1992). Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes.

Petzold, C. (1998). *Programming Windows - The definitive guide to the Win32 API.* Microsoft Press.

Richter, J. M., & Nasarre, C. (2007). *Windows via C/C++.* Microsoft Press.

Shaw, M., & Garlan, D. (1996). *Software Architecture - Perspectives on an Emerging Discipline.* Prentice Hall.

Singh, R. *INTERNATIONAL STANDARD ISO/IEC 12207 SOFTWARE LIFE CYCLE PROCESSES.* Federal Aviation Administration, Washington DC, USA.

Sivakumar, N. (2007). *C++/CLI in Action.* Manning.

Sparks, S., Benner, K., & Faris, C. (1996). *Managing Object-Oriented Framework Reuse.* IEE Computer.

Stroustrup, B., & Ellis, M. A. (1990). *The Annotated C++ Reference Manual.* Addison-Wesley.

Sutter, H. (2004). *C++ Coding Standards: 101 Rules, Guidelines and Best Practices.* Addison-Wesley.

Taligent. (1995). *The Power of Frameworks.* Addison-Wesley.

Werfel, J., Xie, X., & Seung, H. S. *Learning curves for stochastic gradient descent in linear feedforward networks.* Massachusetts Institute of Technology.

Young, P. (2003). *Naming Conventions inWin32 and MFC.* Stanford University.

Statement of autonomy

This is to confirm that this work was created autonomously by me, Robert Neumann, using only the declared sources of information and tools.

Wolmirstedt (Germany), 9th of May 2008 Robert Neumann