Evolution and Composition of Object-Oriented Frameworks

Michael Mattsson

University of Karlskrona/Ronneby
Department of Software Engineering and Computer Science

ISBN 91-628-3856-3

Printed in Sweden
Kaserntryckeriet AB
Karlskrona, 2000

To Nisse, my father-in-law

- who never had the opportunity
to study as much as he
would have liked to

This thesis is submitted to the Faculty of Technology, University of Karlskrona/Ronneby, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Engineering.

Contact Information:

Michael Mattsson
Department of Software Engineering
and Computer Science
University of Karlskrona/Ronneby
Soft Center
SE-372 25 RONNEBY
SWEDEN

Tel.: +46 457 38 50 00
Fax.: +46 457 27 125
Email: Michael.Mattsson@ipd.hk-r.se
URL: http://www.ipd.hk-r.se/rise

Abstract

This thesis comprises studies of evolution and composition of object-oriented frameworks, a certain kind of reusable asset. An object-oriented framework is a set of classes that embodies an abstract design for solutions to a family of related problems. The work presented is based on and has its origin in industrial contexts where object-oriented frameworks have been developed, used, evolved and managed. Thus, the results are based on empirical observations. Both qualitative and quantitative approaches have been used in the studies performed which cover both technical and managerial aspects of object-oriented framework technology.

Historically, object-oriented frameworks are large monolithic assets which require several design iterations and are therefore costly to develop. With the requirement of building larger applications, software engineers have started to compose multiple frameworks, thereby encountering a number of problems. Five common framework composition problems, together with existing solution approaches and the underlying causes for the problems are presented in this thesis. Adopting a reuse technology, such as object-oriented frameworks, in a software development organization causes changes and additions of practices and procedures. We present problems and possible solutions related to these issues. Examples of topics addressed are; domain scoping, business models, verification of the framework's abstract behavior, and when to release a framework. Object-oriented frameworks, as all software, evolve due to changed and new requirements. The evolution of object-oriented framework can be more costly than conventional software since there generally exist several applications based on and forced to evolve with the framework. In our studies, we characterize different views of framework evolution. Aspects investigated are structural and behavioral stability, change and growth rates using historical information and effort distribution of framework development and customization. We also provide an assessment of the methods used for characterizing the evolution against the following management issues; identification of evolution-prone modules, framework deployment, change impact analysis, benchmarking and requirements management. As part of these studies, we have extended and validated two proposed methods for software evolution; one for quantitatively assessing stability of a framework, which has been extended with a set of framework stability indicators, and one for identifying evolution-prone modules based on historical information (adapted for object-orientation). Our studies have validated that these methods are feasible and possible to apply on industrial object-oriented frameworks. In addition, we provide quantitative evidence that

Preface

The research work presented in thesis has been carried out as part-time Ph.D. studies since the summer 1992 when I joined the University of Karlskrona/Ronneby. This thesis includes a collection of six papers and is organized in the following three parts:

- Introduction. The introduction gives a background to the presented papers. Section 1 introduces the notion of object-oriented frameworks and our view of effective utilization of framework technology. The section also discusses the following activities; development & usage, composition and evolution of object-oriented frameworks. Each of the activities is illustrated with one or more typical examples to give the reader an understanding of the phenomena. Section 2 presents an overview of research topics in the framework field. The overview is presented in a triplet form, <background, existing approaches, research questions>, thereby also giving an overview of related work. Section 3 describes the research methods used, the main contributions and a summary of the papers.
- Framework Composition. This part includes two papers, I-II, the first is about problems and experiences of adopting framework technology in an organisation. The second paper lists five problems which may occur when composing two or more frameworks. Except the problems, existing solutions approaches as well as the causes for the problems is presented.

- Framework Evolution. This part includes four papers, III-VI, regarding framework evolution. Paper III presents a method for assessing structural and behavioural stability of an evolving framework and its application. Paper IV presents and applies a method for identifying and characterizing the evolution-proness of a framework, its subsystem and their modules. In paper V the relative distribution of effort per development phase and version for a framework and the relationship between framework development effort and framework instantiation effort is presented. In Paper VI we have compared and assessed the methods presented and used in papers III-V. The comparison is between the methods and the assessment against a set of evolution issues comprising identification of evolution-prone modules, framework deployment, change impact analysis, benchmarking and requirements management. Through these studies we have obtained knowledge and information about the nature of framework evolution that, hopefully, will assist management making well-informed decisions about a framework's evolution, thereby ensuring controlled and predictable evolution of functionality and costs for future framework development efforts.

December 1999
Michael Mattsson

Acknowledgements

The work presented in this thesis was financially supported in parts by Blekinge Forskningsstiftelse and the Foundation for Knowledge and Competence.

I am deeply indebted to many persons who have provided help, support and encouragement throughout my work with this thesis. First of all, I want to express my gratitude to my supervisor, colleague and friend *Prof. Jan Bosch* who always believed in me, for all the work we have done together and for being a friend. Thank you Jan. A special thanks to my friend and colleague *PO (PerOlof) Bengtsson* for fruitful discussions about meaningful and meaningless things. Many thanks to *Lennart Olsson*, Utilia Consult AB, who introduced me to the field of object-oriented frameworks in the early nineties. He is the reason, directly or indirectly, that many swedish companies, a few mentioned below, were early successful adopters of framework technology, which has been beneficial for my research work.

Warm thanks to the co-authors of some publications, *Peter Molin* (now Symbian AB), *Mohamed Fayad*, University of Nebraska, Lincoln, and *Christer Lundberg*, University of Kalmar. I would also thank my other colleagues and friends in the software engineering research group (RiSE) in Ronneby.

Special thanks to the following of our industrial partners for their co-operation; Ericsson Software Technology AB, Axis Communications AB, Althin Medical AB and EC-gruppen AB. A very special thanks to *Mikael Roos*, Ericsson Software Technology AB, for providing me access to the Billing Gateway framework and to *Rolf Svensson* and *Drazen Milicic* (now Wilnor AB) for assistance with source code, documentation and other information.

I would also express my warmest gratitude to my parents, *Bertil* and *Mona*, for always being there and having time to take care of our children during high workloads or long journeys.

Finally, thanks to my beloved wife *Helena* and my children *Max* and

Contents

Introduction 5

1. Object-Oriented Frameworks 6
2. Research Topics in the Framework Field 19
3. Research Results 26

Framework Problems and Experiences 41

1. Introduction 42
2. Object-Oriented Frameworks 44
3. Examples of Application Frameworks 48
4. Problems and Experiences 51
5. Summary 74

Composition Problems, Causes, and Solutions 81

1. Introduction 82
2. Object-Oriented Framework Examples 84
3. Framework Composition Problems 87
4. Underlying Causes 93
5. From Problems to Causes to Solutions 97
6. Summary 104

Observations on the Evolution of an Industrial OO Framework 109

1. Introduction 109
2. The Case Study 111
3. Framework Evolution Observations 113
4. The Evolution of Subsystems 117
5. Related Work 122
6. Conclusion 123

Stability Assessment of Evolving Industrial Object-Oriented Frameworks 127

1. Introduction 128
2. Object-Oriented Frameworks 131
3. The Two Framework Cases 133
4. The Method 138
5. Results and Analysis 141
6. Observations 148
7. Assessment 153
8. Related Work 154
9. Conclusion 155

Effort Distribution in a Six Year Industrial Application Framework Project 159

1. Introduction 160
2. Case Study Description 160
3. Product Metric Values 164
4. Framework Development Effort Data 166
5. Framework Customization Effort Data 169
6. Related Work 174
7. Conclusion 175

Assessment of Three Evaluation Methods for Object-Oriented Framework Evolution 177

1. Introduction 178
2. Object-Oriented Frameworks 182
3. The Methods and Major Results 185
4. Assessment and Comparison 207
5. Related Work 212
6. Conclusion 214

List of Papers

The following six papers are included in the thesis:

[I] Frameworks Problems and Experiences,
Jan Bosch, Peter Molin, Michael Mattsson, PerOlof Bengtsson and Mohamed Fayad
Chapter in “Building Application Frameworks: Object Oriented Foundations of Framework Design", Eds: M. E. Fayad, D. C. Schmidt, R. E. Johnson, Wiley & Sons, ISBN#: 0-471-24875-4, 1999, pp. 55-82

[II] Composition Problems, Causes, & Solutions,
Michael Mattsson and Jan Bosch.
Chapter in "Building Application Frameworks: Object Oriented Foundations of Framework Design", Eds: M. E. Fayad, D. C. Schmidt, R. E. Johnson, Wiley & Sons, ISBN#: 0-471-24875-4, 1999, pp. 467-487

[III] Evolution Observations of an Industrial OO Framework,
Michael Mattsson and Jan Bosch
Proceedings of the International Conference on Software Maintenance, ICSM '99, pp. 139-145, Oxford, England, 1999

[IV] Stability Assessment of Evolving Industrial Object-Oriented Frameworks,
Michael Mattsson and Jan Bosch
Accepted with minor revisions to the Journal of Software Maintenance, 1999

[V] Effort Distribution in a Six Year Industrial Application Framework Project,
Michael Mattsson
Proceedings of the International Conference on Software Maintenance, ICSM '99, pp. 326-333, Oxford, England, 1999

[VI] Assessment of Three Evaluation Methods for Object-Oriented Framework Evolution,
Michael Mattsson and Jan Bosch
Reserach report 1999:20, Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby, Sweden, also submitted to the Journal Information and Software Technology, 1999

Related Publications

The following publications are related but not included in the thesis:

[VII] Framework Integration Problems, Causes and Solutions,
Michael Mattsson, Jan Bosch and Mohamed E. Fayad
Communications of ACM, Vol. 42, No. 10 (Oct. 1999), Pages 80-87,1999

[VIII] Characterizing Stability in Evolving Frameworks,
Michael Mattsson and Jan Bosch
Proceedings of the 29th International Conference on Technology of Object-Oriented Languages and Systems, TOOLS EUROPE '99, pp. 118-130, Nancy, France, June 7-10, 1999

[IX] Evolution Characteristics of an Industrial Application Framework,
Michael Mattsson
Workshop on Object-Oriented Architectural Evolution at the 13th European Conference on Object-Oriented Programming, ECOOP '99, Lisbon, Portugal. 1999

[X] Frameworks as Components: A Classification of Framework Evolution,
Michael Mattsson and Jan Bosch,
Proceedings of NWPER´98, Nordic Workshop on Programming Environment Research, Ronneby, Sweden, August 1998, pp. 163-174

[XI] Object-Oriented Framework-based software Development,
Jan Bosch, Peter Molin, Michael Mattsson and PerOlof Bengtsson
Accepted for Publication in ACM Computing Surveys Symposia on Object-Oriented Application Frameworks, 1998, to appear.

[XII] Framework Composition: Problems, Causes and Solutions,
Michael Mattsson and Jan Bosch
Proceedings of the 23rd International Conference in Technology of Object-Oriented Languages and Systems, TOOLS '97 USA, pp. 203-214, Santa Barbara, California, US, July 28 - August 1, 1997

[XIII] On Using Legacy Software Components with Object-Oriented Frameworks,
C. Lundberg and Michael Mattsson
Proceedings of Systemarkitekturer'96, Borås, Sweden, 1996

[XIV] Object-Oriented Frameworks - A survey of methodological issues,
Michael Mattsson
Licentiate Thesis, Department of Computer Science, Lund University, CODEN: LUTEDX/(TECS-3066)/1-130/(1996), also as Technical Report, LU-CS-TR: 96-167, Department of Computer Science, Lund University, 1996

Introduction

The emergence of evolving or global markets has fundamentally altered competition as many companies have known it. The dynamics of these markets are forcing compression of product development times and expansions of model variety of the products [50].

This is especially true within the information technology field and related areas where we see that an increasing part of industrial development involves development of software. The products developed contain more and more software, or are genuine software products, which need to be enhanced and modified to meet the new market requirements.

This increases the need of methods, tools and techniques for developing and managing the process of creating and evolving software products, i.e. *software engineering* [46].

To meet the market requirements of compressed or reduced product development times *reuse of software* has for several decades been an important issue for the software development organization. Software reuse is the process of creating software systems from existing software rather than building the software from scratch. This visions was introduced 1968 by McIlroy in his seminal paper *Mass Produced Software Components* [31] and software reuse is today an established sub-discipline within software engineering research.

1. Object-Oriented Frameworks

When introducing software reuse in a software development organization this implies changing existing software life cycles into new ones [21]:

- Development-for-reuse and,
- Development-with-reuse

The development-for-reuse life cycle has the focus to develop reusable assets intended to be reused in the development-with-reuse life cycle. In the development-with-reuse life cycle reusable assets are searched for and used, if suitable, in the software development.

Reusable assets in object-orientation can be found in a continuum ranging from objects and classes to design patterns and object-oriented frameworks.

Historically, reusable software components have been procedural and function libraries (the 1960s) and in the late 60's Simula 67 [4] introduced objects, classes and inheritance which resulted in class libraries. Both the procedural and the class libraries are mainly focused on reuse of code. However, reuse of design would be more beneficial in economical terms. This since design is the main intellectual content of software and it is more difficult to create and recreate then programming code.

With the integration of data and operations into objects and classes an increase in reuse was achieved. The classes were packaged together into class libraries. These class libraries often consist of classes for different data structures, such as lists and queues. The class libraries were further structured using inheritance to facilitate specialization of existing library classes. The class libraries delivered software reuse beyond traditional procedural libraries.

The problems with reusing class libraries are that they do not deliver enough software reuse, i.e. they only allow for reuse of relatively small pieces of code. Software developers still have to develop a lot of application code themselves. The striving to increase the degree of reuse and the desire to reuse designs has resulted in *object-oriented frameworks*.

A number of different and similar definitions of the concept of object-oriented frameworks exist. We provide a few to give the reader an understanding of the framework concept:

"a framework binds certain choices about state partitioning and control flow; the (re)user (of the framework) completes or extends the framework to produce an actual application" [9].

In this definition we see that architectural design decision has been taken which future applications has to conform to and that the application will be an extension of the framework.

In [20] the following two definitions are made:

"*a framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact*"

"*a framework is the skeleton of an application that can be customized by an application developer*"

The first of these definitions indicates that a framework do not have to address one application domain but that it is possible to develop frameworks for smaller domains, thereby opening up for composition of frameworks. Another observation to do is the wording "set of abstract classes" which intuitively lead to the conclusion that the extension of the framework has to be done through inheritance. This is not always the case since there exist other ways of extending the framework, which will be elaborated on later. So, the first definition focuses on the structure of the framework whereas the latter definition describes the framework's purpose i.e. to be extended.

A definition that captures, what we consider to be the essential aspects of an object-oriented framework is found in [17]:

"A framework is a set of classes that embodies an abstract design for solutions to a family of related problems. "

Thus, a framework consists of a set of classes (not necessarily abstract), whose instances collaborate (embodies), is intended to be extended, i.e. reused (abstract design), and do not have to address a complete application domain (a family of related problems), i.e. allowing for composition of frameworks. In addition, frameworks are expressed in a programming language, thereby providing reuse of both code and design.

A framework particularly emphasizes those parts of the domain that will remain stable and the relationships and interactions among those parts, i.e. the framework is in charge of the flow of control.

The reuse leverage for object-oriented frameworks comes with inheritance in combination with dynamic binding. We illustrate this with a single class framework [9] for copying files [18].

The single class framework is an abstract class `File` with `read`, `write` and `size` operations. It is possible to extend the framework with subclasses such as `UnixFile` and `NetworkedFile`. Since different kind of files have different algorithms for reading and writing files the `File` class will not implement these operations. However, the `read` and `write` operations can be used to implement other operations, for example `copy`. A file can now copy itself to another file by using the `copy` operation. The `copy` operation is implemented as

```
File::copy()( File aFile)
{
   char buffer[BlockSize];
   for (int i = 1; i++; i <= this->size() )
      {
         this -> read (buffer);
         aFile -> write(buffer);
      }
}
```

using the C++ programming language. A subclass of class `File` that defines the `read`, `write` and `size` operations will now be able to use the `copy` operation. The copying of a `UnixFile` to a `NetworkedFile` will invoke the `read` and `size` operations on the `UnixFile` and the `write` operation on the `NetworkedFile` (given that `UnixFile` do not override the `copy` operation). The point is that an operation in a superclass, the `copy` operation, can call operations in subclasses, i.e. the superclass is controlling the execution flow. Operations like the `copy` operation is often called *template* methods and operations like `read` and write are called *hook* methods [37].

The use of template and hook methods are a distinguishing feature of an object-oriented framework compared to a conventional library, i.e. the framework is designed for dynamic binding. When a conventional library is used by an application routine, calls are made from the application to the library only. In an object-oriented framework, however, calls can go also in the opposite direction (the inversion of

control), see figure 1. This two-way flow of control is made possible by dynamic binding.

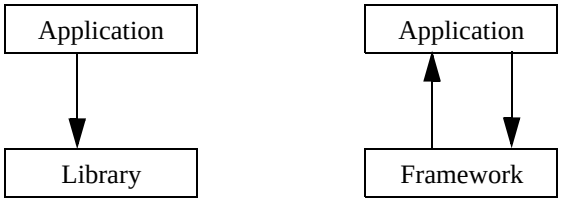

Figure 1. *The two-way flow of control*

The single class framework example illustrated the use of inheritance and dynamic binding for one operation in one class. If we scale up the example to a larger framework consisting of more (abstract) classes, obviously providing more template and hook operations, the framework thereby provides a large number of extension points, for different kinds of functionality, to the framework. Now we can imagine the degree of reuse delivered by a framework, comprising of a set of (abstract) classes which collaborate and own the thread of control. It captures a problem domain, and provides a large number of extension points for applications to extend.

So, the reusability benefits for an object-oriented framework emanate from its extensibility and the inversion of control. The *extensibility* of the framework is provided by a set of hook methods that makes it possible for future applications to extend the framework's interface. The hook methods explicitly divides the framework design into stable interfaces and behavior and the variable parts which are to be customized for the application to be developed. The extensibility techniques used are often used to classify a framework as a white-box, black-box or visual builder framework.

- A *white-box framework* makes heavily use of inheritance and dynamic binding which are available in object-oriented languages. The extensibility in a white-box framework is achieved by first inheriting from framework superclasses (often abstract classes) and secondly overriding the pre-defined hook methods. White-box frameworks are sometimes called architecture-driven or inheritance-focused frameworks, e.g. [48].

- A *black-box framework* is relying on defined component (object) interfaces and object composition where the components are conforming to the interfaces. The extensibility of a black-box framework is achieved by first defining components that conform to a particular interface and secondly integrating these components (objects) into the framework. Black-box frameworks are sometimes referred to as data-driven or composition-focused frameworks, e.g. [48].
- *A visual builder framework* [40]. For some problem domains is it possible to develop visual builder frameworks. To evolve to the visual builder stage the framework must be a black-box framework. An application consists of different parts. First, a script is used that connects the objects of the framework and then makes them active [40]. The other part is the behavior of the individual objects, which is provided by the black-box framework. Thus, framework extension in the black-box case is mainly a script that is similar for all applications based on the framework. One can now develop a separate graphical program, a visual builder, which makes it possible to specify the objects to be included in the application and the interconnection of them. The visual builder generates the code for the application from the specification. The addition of a visual builder to the framework provides a user-friendly graphical interface which make it possible for domain experts to develop the applications by manipulating images on the screen. Examples of frameworks at the visual builder stage are the windows builders that are associated with GUI frameworks. The extensibility technique for a visual builder frameworks is the same as for black-box frameworks but since they are directly developed for domain experts we consider them as a special kind of framework.

Our view of effective utilization of framework technology is based on the following structuring of frameworks. One “backbone” framework that provides hook capabilities for attaching other frameworks for more specialized domains, figure 2. The backbone framework can be viewed as a second-order framework responsible for the whole application’s main behavior, whereas the other frameworks (first-order frameworks) can be considered as flexible components.

Not necessarily, the both kinds of frameworks have to exist. If

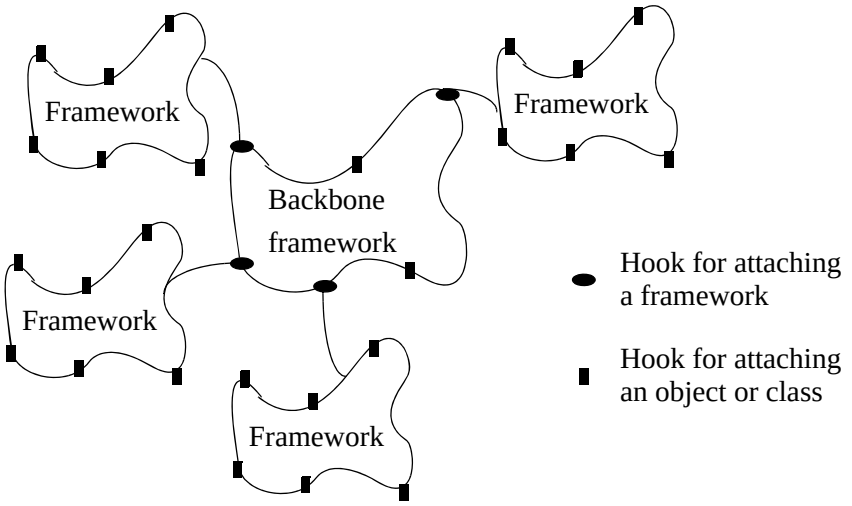

Figure 2. *Framework structuring in an application*

there is only the backbone framework, our structuring approach degenerate to the traditional (large) monolithic framework approach where the framework captures a complete application domain. If the backbone framework is missing we end up with an application architecture that comprises only of components, which are frameworks.

If both kinds of framework exists we have an initial structure for a software product-line architecture. A software product line is a way to organize the development of a set of related software products and consist of a software product-line architecture, a set of reusable components and a number of software products [47]. The role of the *software product-line architecture* is to organize the commonalities and variabilities of the products contained in the software product line and, as such, to provide a common overall structure. A component in the architecture implements a particular domain functionality. The set of software products based on the software product-line architecture is called a *product family.*

Often the situation is that both kind of frameworks exist but that all first-order frameworks have not matured enough to be explicit as white- or black-box frameworks but exist as small domain specific class libraries with vaguely connected classes. This is for example

the case for the studied Billing Gateway (BGW) framework, paper III-VI, in this thesis.

The advantages or promises with the proposed structuring approach are:

- Planned framework maturity *evolution*. The components (the first order frameworks) can evolve smoothly from being immature frameworks, i.e. small domain specific class libraries with vaguely connected classes, to white-box-framework and further on to black-box frameworks. This evolution can be done gradually and in a pace depending on the organisation's business goals and intended use of the component in future products.
- Architecture visibility. The software architecture of the application becomes visible by the second-order framework. Thereby making it easier to independently *evolve* the software architecture and the components.
- Component exchange in product line architecture. It provides for relatively simple exchange of components in a software product line architecture. Thereby making it possible to relatively easily support different platforms or providing for low-end and high-end products. This by changing one or a few flexible components (frameworks), i.e. *evolving* the product family.
- Inter-product line component exchange. It provides for change of components between different software product line architectures, i.e. *evolving* the product family. This requires that we, except for the requirement of designing the framework for composition, as in the case for the fine-grained framework approach, also formulate a requirement of independent composition. With independent composition we mean that the framework should be designed for composition without making any assumption of which other frameworks it is supposed to be composed with.

To summarize, the structuring approach proposed allows for software change, which is a necessity today, but the approach requires an understanding of the nature of framework evolution so controlled and predictable evolution of the framework's functionality and costs can be achieved.

Development and usage of frameworks

The process of developing and using an object-oriented framework comprises of three major activities, problem domain analysis, framework design and instantiation of the framework, figure 3.

- *Problem domain analysis*. Domain analysis can be viewed as a broader and more extensive analysis that tries to capture the requirements of the complete problem domain including future requirements. Depending on the problem domain different kind of information sources are used to obtain knowledge about the domain. Examples of sources are previous experiences of the domain, domain experts and existing standards. If the problem domain is large or covers a complete application domain, specialized domain analysis methods may be used that provide systematic methods steps for performing the analysis. In [44] a comprehensive overview of existing domain analysis methods is presented and compared.
- *Framework design*. The objective with this activity is to end up with a flexible framework. The activity is effort consuming since it difficult to find the right abstractions and identify the stable and variable parts of the framework, which often results in a number of design iterations. To improve the extensibility and flexibility of the framework to meet the future instantiation needs, design patterns [12][38] may be used. The resulting framework can be a white-box, black-box or visual builder framework depending on the domain, effort available and intended users of the framework.
- *Instantiation of the framework*. The instantiation of a framework differs depending on the kind of framework, white-box, black-box or visual builder. We do not discuss the different ways here again but refer to the previous discussion of the different kinds of frameworks. A framework can be instantiated one or many times within the same application, instantiated in different applications and in the cases of one large monolithic framework or a backbone framework with immature or maturing first order frameworks, the instantiation activity results in an application.

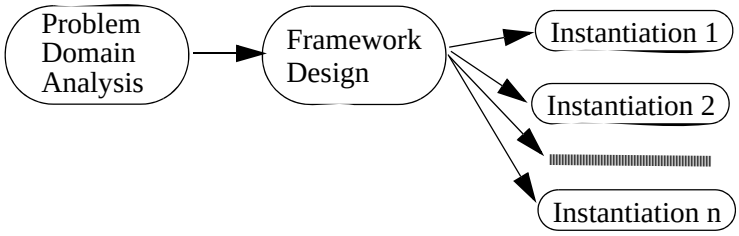

Figure 3. *Development and usage of frameworks*

Composition of frameworks

The view of considering object-oriented frameworks as components opened up for the development of smaller frameworks and for composition of two or more framework. However, the composition of frameworks could be difficult due to that frameworks often are designed based on the assumption that the framework owns the thread of control, and is designed as a monolithic framework designed for extension only (not composition). We have identified five framework composition problems that may occur:

- the composition of framework control,
- the composition of the framework with legacy components,
- the framework gap problem,
- the composition overlap of framework entities,
- the composition of entity functionality.

We elaborate a bit about the composition problem of framework control to give an understanding of the problem and refer to paper II for a more extensive discussion of all five problems.

One of the most distinguishing features of a framework is its ability to make extensive use of dynamic binding. One problem when composing two frameworks is that both frameworks expect to be the controlling entity in the application and in control of the main event loop.

As an example, we use the composition of a measurement system framework [7] and a GUI framework as shown in figure 4. Measure-

ment systems are systems that are located at the beginning or end of production lines to measure some selected features of production items. The hardware of a measurement system consists of a trigger, one or more sensors and one or more actuators. The framework for the software has software representations for these parts, the measurement item and an abstract factory for generating new measurement item objects. A typical measurement cycle for a product starts with the trigger detecting the item and notifying the abstract factory. In response, the abstract factory creates a measurement item and activates it. The measurement item contains a measurement strategy and an actuation strategy and uses them to first read the relevant data from the sensors, then compare this data with the ideal values and subsequently activate the actuators when necessary to remove or mark the product if it did not fulfil the requirements.

Since most readers are familiar with GUI frameworks, such as Microsoft Foundation Classes (MFC) [45], this kind of framework is not described.

The measurement system framework has, from the moment a trigger enters the system, a well defined control loop that has to be performed in real-time and that creates a measurement item, read sensors, computes, activate actuators and stores the necessary historical data. The GUI framework has a similar thread of control, though not real-time, that updates the screen whenever a value in the system changes, e.g. of a sensor, or performs some action when the user invokes a command. These two control loops can easily collide with each other, potentially causing the measurement part to miss its real-time deadlines and causing the GUI to present incorrect data due to race conditions between the activities.

Solving this problem is considerably easier if the frameworks are supplied with source code that makes it possible to adapt the frame-

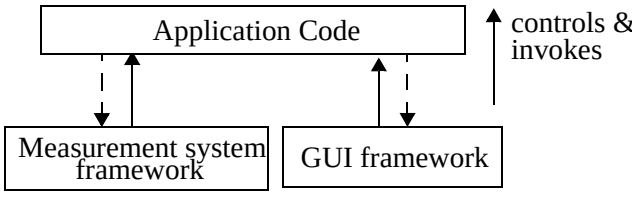

Figure 4. *Problem of framework control composition*

work to handle this problem. However, the control loop in the framework may not be localized in a single entity. Often it is distributed over the framework code, which causes changes to the control to affect considerable parts of the framework.

In addition to merging the framework control loops by adapting the framework code, each framework can be assigned its own thread of control, leading to two or more independently executing control loops. Although this might work in some situations, one has to be aware of, at least, one important drawback. All application objects that can be accessed by both frameworks need to be extended with synchronization code. Since classes often cannot be modularly extended with synchronization code, it requires editing all reused classes that potentially are accessed by both processes.

The five composition problems are primarily caused by the cohesion of involved frameworks, the accuracy of the domain coverage of frameworks, the design intentions of the framework designer and, in some cases, the lack of access to the source code of the framework. The identified problems and their causes are, up to some extent, addressed by existing solutions, such as the adapter design pattern, wrapping or mediating software, but these solutions generally have limitations and either do not solve the problem completely or require considerable implementation efforts. The only serious approach to solve this kind of problems is to intentionally design the frameworks for composition from the very beginning. Attempts and guidelines for this kind of framework design have recently started to emerge e.g. [16] [39].

Framework evolution

Object-oriented frameworks, as other software, evolve over time. Controlled and predictable evolution of framework is of importance since the reusable framework not only evolves as a framework but that it only cause the evolution of the applications developed using the framework. To reduce the costs of updating the applications developed and the cost of evolving the framework, controlled and predictable evolution of the framework is necessary.

The introduction of new or changed application requirements causes the application to evolve. In addition, the requirements, in most cases, affect the frameworks too. As part of our research work

[28] we have identified four major kinds of framework evolution caused by new or changed requirements, which are presented below together with one or more typical actions that are performed for the evolution category.

- *Internal reorganization*: In response to new requirements, the framework may be reorganized internally without affecting the externally visible functionality of the framework. The reason for reorganization is to improve one or more of the quality requirements (non-functional requirements) of the framework, e.g. flexibility, reusability or performance. The following internal reorganizations can be identified: 1) *Hot-spot introduction*. A hot-spot is a place where framework extension takes place (a hook method corresponds to a hot-spot). With the introduction of a new hot-spot in the reuse interface of the framework, an additional way to configuring the framework becomes available. The client of the framework then has a possibility to use this new flexibility option or rely on the original default behaviour. Introduction of a new hot-spot generally improves the flexibility quality attribute of the product. For instance, in a file-system framework, a hot-spot in the file class can be introduced for dealing with access control. The default behaviour of the hot-spot is to perform no access control, but one can specialize this behaviour with an access control algorithm. 2) *Restructuring*: It may be necessary to perform some framework restructuring using refactoring techniques [35] to comply to new requirements. An example of a possible refactoring is transforming a class hierarchy relationship into an aggregate relationship. These kinds of transformations are behaviour preserving but result in a changed reuse interface of the framework. Restructuring is often used for achieving better maintainability of the framework. 3) *Changing control flow.* In cases where the order of actions in response to an event needs to be changed, the flow of control between the objects in a framework needs to be changed. One may argue whether this is a change in functionality, since the same behaviour is executed, although in a different order. Changing the control flow only affects the internal behaviour of the framework, not its structure. 4) *Refining framework concepts*. The new requirements may point out flaws in the framework design such as classes

that has two roles that now have to be refined further. This action will make the framework component more understandable and it may provide new hot-spots to be added to the reuse interface. Concept refining does not affect the internal behaviour but it does change the internal structure of the framework component.

- *Changing functionality*: The functionality of the framework may need to be changed. For instance, due to new underlying hardware the way the framework interacts with the hardware needs to be adapted. The interface and structure of the framework tend to remain constant, but the behaviour in response to events is changed. We present one change functionality action, *changing internal class behaviour.* A behavioural change is localized to only one class in the framework and the class affected has no complex behavioural relationships to other classes. A simple example of this kind of action is the change of a data compression algorithm in a class. The action does not affect the internal structure of the framework but introduces changes in the internal behaviour.

- *Extending functionality*: The most common type of framework evolution is the extension of the framework functionality. Especially when developing new applications, additional functionality may be required from one or more of frameworks. In this category we have identified two kinds of actions: 1) *Functionality introduction.* New requirements may demand the extension of the framework with new functionality. If the requirement only affects one framework, the situation is similar to the traditional monolithic framework development process, which implies iterating the framework design for achieving a new framework version. An example could be the evolution of a read-only file system framework into a read-write file system framework. In principle, only write-storage functionality will be added to the original framework and no fundamentally new functionality was introduced. 2) *Concept introduction*. The introduction of new concepts in the framework causes the introduction of new functionality. New concepts require the reorganization of the framework since new concepts have relations to the existing concepts. Concept introduction often

requires the use of earlier mentioned techniques, e.g. concept refinement, hot-spot introduction, etc. The introduction of authorization in a file-system framework is a typical example of this sub-category.

- *Reducing functionality*: In certain cases, functionality that was developed as an integral part of the framework functionality has to be removed from the framework. For example, the introduction of a new specialized framework requires the functionality it provides to be removed from the other frameworks. We identify two kind of actions: 1) *Enable/disable introduction.* There may be requirements that require the introduction of an enable/disable mechanism in the framework. An example of this situation could be an application architecture comprising authorization functionality realized by a monolithic framework. All instantiations may not require the authorization functionality. Then, a suitable solution is to provide the framework with an enable/disable mechanism for this functionality. 2) *Cutting functionality.* If new requirements require the use of an additional framework that has to collaborate with the existing framework component, conflicts due to of domain overlap or implicit assumptions about ownership of the thread of control may occur. This will make it necessary to cut functionality out of the existing framework since it may not be possible to make necessary changes in the new framework, for instance, due to lack of access of source code or the complexity of the framework.

Unfortunately, it is very hard to observe and investigate framework evolution on the detailed level described above. In practice, we then have to observe and investigate the evolution on the level of complete frameworks. In our research we have studied different evolution aspects of an industrial object-oriented framework, the Billing GateWay framework, during four major versions. In particular we have studied the stability aspect of evolving frameworks, where the stability is characterized by a set framework stability indicators [Paper III], change and growth rates for the framework measured in terms of changed and added classes [Paper IV] and the effort distribution of framework development and customization [Paper V]. In addition, we have compared and assessed the methods used in Paper

III-V, [Paper VI]. The comparison is between the methods and the assessment against a set of evolution issues comprising identification of evolution-prone modules, framework deployment, change impact analysis, benchmarking and requirements management. Through these studies we have obtained knowledge and information about the nature of framework evolution that, hopefully, will assist management making well-informed decisions about a framework's evolution, thereby ensuring controlled and predictable evolution of functionality and costs for future framework development efforts.

2. Research Topics in the Framework Field

Object-oriented framework technology bears the promises, and have in some cases demonstrated them, to deliver improved quality software and reduced development costs. But still, there remain topics that deserve further attention for fully exploiting the technology and we identify, at least, the following important aspects to investigate in the framework field.

2.1 Framework design for composition

Background

The common approach today when using two or more frameworks for the same application is just to try to integrate them with each other. Thus, resulting in a number of composition problems [26][27] since the framework are not designed for composition. The only serious approach to solve this kind of problems is to intentionally design the frameworks for composition from the very beginning.

Existing approach(es)

In [39] the concept of *framelet* is introduced as a way for solving the composition problems. A framelet is 1) small in size (<10 classes), 2) do not assume the main control of the application and 3) have a clearly defined simple interface. Thus, framelets clearly differ from the main characteristics of a large monolithic framework. Framelets

are intended to be specialized by subclassing and composition. A framelet exist in the context where a software system is viewed as a set of service interfaces and their implementation. A interface that belongs to the framelet without its implementation is part of the specialization interface of the framelet and an interface that belongs to the framelet with its implementation is part of the framelet's service interface. This modularization into framelets of a large monolithic framework may solve the composition problems. The idea of framelets has been applied in an effort of rearchitecting a legacy systems which appears to have been successful. Thus, framelets have been identified through an analysis of the legacy systems. However, no evidence is presented that demonstrate that it is possible to develop a set of framelets (forward engineering) intentionally and easily compose them with each other.

Research question(s)

Still, [39] is promising work, more and other work is needed. The long-term objective must be some kind of framework *integration standard*. The existence of a standard will assure that the exchange of frameworks in an application or a product line architecture will be possible. An intermediate step is development of framework *design guidelines* that eliminate, or alleviate, the composition problems ,e.g. [16].

2.2 Documentation of frameworks

Background

Accurate and sometimes also comprehensible software documentation is in general difficult to obtain. The documentation issue becomes even more important in the context of reusable assets. The documentation of a framework must be described on different levels of abstraction since it must address the needs of developers with varying levels of experience. There are different needs for an experienced user of the framework, a first-time user and a framework maintainer. The framework documentation must encompass [19]:

- the purpose of the framework

- how to use the underlying framework
- the purpose of the application examples
- the design of the framework

Existing approach(es)

Regarding documentation of frameworks a few different approaches can be identified [24];

- Cookbook Approaches
- Pattern Approaches

Cookbook Approaches. Most framework users are not interested in the details of a framework but are looking for documentation that describes how to use the framework. They are looking for some kind of cookbook, one that guides them in how to use the framework. A few different approaches that use the cookbook idea exist [1][22][24].

The cookbook approaches are all good for addressing the purpose of the framework and present examples. The aspect of how to use the framework is described informally through natural language. The detailed design of the frameworks is nearly neglected in these approaches, but can be presented with some standard design notation, e.g. [42]. Only some discussions about a few classes and class hierarchies in the frameworks exist in the cookbook approaches.

Pattern Approaches. Patterns exist in a number of shapes and some of them are suitable for the documentation of frameworks. Object-oriented design patterns [12] describe parts of an object-oriented design. This doesn't make them suitable for presenting the complete design of an object-oriented framework in one view. A class in a framework can have two or more different roles, depending on how many design pattern's it is participating in. This means that it is not possible to identify the borderlines between different design patterns in a framework.

However, object-oriented design patterns are useful in the context of describing parts of a framework design through the diagrams for the design patterns and the vocabulary introduced. In some cases where the framework offers the users few entries to adapt the framework, design patterns can be a suitable approach.

The object-oriented design pattern approach does not address the purpose of the framework aspect. The aspect of describing the detailed design of the framework is addressed by the object-oriented design pattern approach since they capture the intricacies of the framework design.

Since a framework is the result of many design iterations, the real world concepts have often been lost in the design. This implies a higher learning curve for the framework, which may be reduced if the framework is documented with design patterns and the framework reuser masters the set of patterns used.

Research question(s)

The documentation issue is often neglected in general but is crucial for object-oriented frameworks since the reuse of the framework will not take place if it is not possible to understand the frameworks and further research on framework documentation is necessary.

To summarize the use of the different approaches, the cookbook approaches seem to be used by companies developing commercial frameworks intended for a mass-market [1], whereas companies developing proprietary frameworks are using the design pattern approach for documenting their software, e.g. [6]. However, no information exist about which kind of documentation is most *cost-effective* for the different kinds of framework users.

In addition, the production of documentation is costly so there is a need of *documentation tools* that, through some kind of reverse engineering tool, provide accurate and usable documentation for the framework reuser.

2.3 Framework development and usage processes

Background

Today, the argued process for developing a framework is to draw from experiences from three or more applications from the same domain. This process of generalizing from existing systems and experiences is to costly so there is a need for a more cost-effective

development process.

Existing approach(es)

Currently, we only see two possible approaches for achieving cost-effectiveness. One, to draw on others experiences and utilize all available knowledge from existing successful frameworks projects. Two, the decomposition of the application into a set of smaller frameworks intended for composition. Hopefully, the development of smaller frameworks together with the integration activity will result in a less costly process than the traditional approach.

Research question(s)

The process of reusing one or more framework is still rather ad hoc. The main issue, we believe, is the documentation of the framework. If an appropriate documentation is available the reuse of the framework will be successful. Thus, the problem of successfully reusing a framework is a *documentation* and *training* problem.

If the approach of developing smaller frameworks is selected the problem of reusing a framework returns back to the problem of framework *composition*.

2.4 Framework evolution

Background

Evolution of object-oriented framework is an important research issue since frameworks can be used in a number of different context; as a large monolithic framework, as a component to be integrated with other frameworks, as a component in a back-bone framework, as a back-bone framework with or without the components (frameworks), and in one or many software product line architectures using frameworks as components. Depending on the context, one may expect the framework evolve in a specific way.

Existing approach(es)

Currently, little knowledge is available how object-oriented frameworks evolve. In [40], the maturity evolution of object-oriented frameworks is discussed. The major stages in the evolution is that a frameworks starts as white-box frameworks and gradually evolve into black-box framework, then a visual configuration and code generation environment for the applicator development is developed, a so called visual builder.

In [10], a metrics-based approach that supports incremental development of a monolithic framework is described. By measuring a set of metrics an assessment of the design quality of the framework can be performed. If the metric data is within acceptable limits, the framework is considered to be good enough from a metrics perspective otherwise another design iteration has to done. The approach is intended to be used during the design iterations, before a framework is released and does not consider long term evolution of a framework. But the basic idea may be used for investigating framework evolution.

In addition, a study, mainly quantitative, of four major versions of a monolithic framework within the telecommunication domain is reported in [29], which also can be found in this thesis, Paper IV.

Research question(s)

To summarize, little is known about framework evolution, especially evolution of smaller frameworks that act as components in larger applications or software product line architectures. Knowledge of the evolution nature of framework is important to attain so an organization can have *controlled and predictable evolution* of the framework, the application or the product line architecture to avoid unexpected technical or economical problems.

2.5 Economical aspects

Background

When should an organization decide to develop a framework or not? How many times must a framework be reused to be economical ben-

eficial? What factors are important to consider? These economical matters, and others, are of concern for any organization developing object-oriented software. Economical aspects in software development, software economics, is still an under-developed area in software engineering, especially for small and medium sized enterprises. In the field of framework economics very little is known.

Existing approach(es)

To the best of our knowledge, we are only aware of one source related to framework economics. In [25] effort data for framework development and usage of a proprietary framework is presented. This work is presented in this thesis, paper V.

Research question(s)

Research issues related to framework economics are [11]:

- *Framework cost metrics*. The issue here is to find a way to measure the cost reduction of reusing frameworks compared to developing the application without using framework technology. Currently, we are not aware of any results in this field. The challenge in this area is to find a convincing argumentation that the use of frameworks has delivered cost reduction since no organisation is willing to develop the same application twice, with and without using frameworks.
- *Cost estimation*. There is a need of developing cost models for predicting the cost of both developing and customizing a particular framework. Since the development of cost models requires empirical data to be calibrated it is necessary to collect effort data. Regarding effort data for framework development and usage we are only aware of one source, i.e. [25]. It is not enough with only effort data to construct the cost model but it is necessary to identify which factors, so called cost drivers, which affect the development and customization costs. Today, we are aware of no sources that discuss these issues but relevant information can, hopefully, be found in the software reuse community. For example, [23], which presents and compares seventeen existing reuse models.

- *Investment analysis*. What is the return of investment of using framework technology in an organisation? Today, no one can give an accurate answer, the application of framework technology is more intuitive than based on justifying economical analyses. So, there is a need of investment analysis models.

3. Research Results

3.1 Research method

It's still a debate within the research community about how software engineering research should be performed, e.g. [51][14][49]. This is not surprising since software engineering has a relatively short history and still is maturing as an research area. Software engineering has emerged from computer science as a complement, but as a more applied discipline. Both computer science research and, to a lesser extent, software engineering, has a large degree of analytical (formal) reasoning as the main starting point. This maybe for historical reasons, since they have their origins in mathematics. However, research in software engineering comprises both computer science, human and organizational issues thereby causing a need of additional research approaches, for example from the social sciences.

A classification of research methods in software engineering was proposed at the Dagstuhl workshop [1], which has been further discussed in [14][51]. The following four categories of research methods were identified, quoted from [1]:

- *The scientific method*. "Observe the world, propose a model or theory of behavior, measure and analyze, validate hypotheses of the model or theory, and if possible repeat".
- *The engineering method*. "Observe existing solutions, propose better solutions, build or develop, measure and analyze, repeat until no further improvements are possible".
- *The empirical method*. "Propose a model, develop statistical or other methods, apply to case studies, measure and analyze, validate the model, repeat". Empirical methods are often further divided into the following three categories [41]:

- *Experiment*: Experimentation is a research strategy involving the assignment of study subjects to different conditions, manipulation of one or more independent variables by the experimenter, measurement of the effects of the manipulation on one or more dependant variables, and control of all other variables. A possible experiment could investigate the effectiveness of two different requirement inspection techniques.
- *Survey*: A survey is a research strategy which comprises the collection of a small amount of data in standardized form from a relatively large number of individuals, groups or organizations and the selection of samples from the individuals, groups or organizations from known populations. An example of a survey could be a questionnaire to a set of organizations about their perceived problems and benefits of applying framework technology.
- *Case study*: A case study is a research strategy which involves an empirical investigation of a particularly phenomenon (the case) within its real life context. The case study strategy is a very flexible approach for performing research and a number of approaches for performing case studies exist. In fact, case studies are inherently multi-method, i.e. involves a mix of observation, interview and analysis techniques, and both quantitative and qualitative data is collected and analyzed. An example of a possible case study could be to study the introduction of framework technology into an organization with respect to software quality and reduced development effort.

- *The analytical method.* “ Propose a formal theory or set of axioms, develop a theory, derive results and if possible compare with empirical observations”.

Paper I and II is based on the authors own experiences of framework technology and open-ended interviews and discussions with software engineers developing and using frameworks. Thus, paper I and II represent case study research within the empirical research method.

The remaining papers, paper III-VI, are all part of a larger case study. Paper III and IV are studies where quantitative data has been collected and analyzed. According to the more detailed classification in [41] these studies can be categorized as using historical observational methods, i.e. a kind of archive analysis. In an archive analysis the data is collected from some kind of archive and the data has been collected for a different purpose than the one used in the current study. In our study, the archive data was not available but existed inherent in the different software versions available and we intentionally created the archive, through applying a metrics tool, with a specific purpose in mind. Paper V is similar to the studies in the previous two papers but the difference is that there now existed a historical archive where the data has been collected for a different purpose. An additional difference is that in paper V we are studying development process data, whereas paper III and IV are studying product data. This distinction is made in [51] where the research method concerning only product data is named *static analysis* and the one that also includes development process data is named *legacy data*.

The focus of paper VI is to present a qualitative comparison between the three methods in paper III-V together with a qualitative assessment of the methods with respect to a set of framework evolution issues. Thus, paper VI can be considered as a large case study comprising the individual studies in paper III-V.

3.2 Research focus

The research presented in this thesis is carried out within the discipline of software engineering, concerned with methods, tools and techniques for developing and managing the process of creating and evolving software products [46]. To meet the requirements of reduced product development effort *software reuse* has established itself as a sub-discipline within in software engineering. Software reuse is the process of creating software systems from existing software rather than building the software from scratch. This change of focus, from development of one single product, to development and usage of many reusable (software) assets, cause changes to the methods, tools and techniques used for creating and evolving the assets.

A reusable asset could be a lot of different things, for example, a sorting routine, a math library, a test suit, a specification document

etc. We present in this thesis research where the reusable assets are object-oriented frameworks. An object-oriented framework is a set of abstract classes that embodies an abstract design for solutions to a family of related problems [17]. For a more elaborated discussion of the framework concept, see section 1. In particular, we have studied

- Problems, anticipated and unanticipated, which may occur when applying framework technology in an organization together with possible solution approaches. The identified problems are related to the development, usage and management of object-oriented frameworks [Paper I]. The issue of composing two or more frameworks has attained additional attention and which problems that may occur in that case, together with existing solution approaches and underlying causes for the composition problems has been studied in [Paper II].
- Different evolution aspects of an industrial object-oriented framework, the Billing Gateway (BGW) framework, during four major versions. In particular we have studied the stability aspect of evolving frameworks, the BGW framework and a graphical user interface framework, Microsoft's Foundation Classes (MFC) framework [Paper III], change and growth rates for the framework [Paper IV] and effort distribution of framework development and customization [Paper V]. In [Paper VI], the papers III-V, are put together in their larger context and a comparison and assessment of the three methods for studying framework evolution is presented. The comparison is between the methods presented in paper III-V and the assessment against a set of evolution issues comprising identification of evolution-prone modules, framework deployment, change impact analysis, benchmarking and requirements management.

3.3 Main contributions

The main results of the presented work include findings related to the application of framework technology within an organization, the usage and composition of multiple frameworks, as well as a set of different methods which can provide management with information for making well-informed decisions about the framework's evolu-

tion, thereby ensuring controlled and predictable evolution of the framework's functionality and costs. All work is based on or has its origin in industrial contexts where object-oriented frameworks have been developed, used, evolved and managed. Thus, the results is based on empirical evidences. This section summaries the main contributions and gives a guide to the reader on the contents of each included paper.

- *Identification of problems and possible solutions when adopting framework technology.* When using new technology it is always the case that some unforeseen problems occur. If the problems are not identified and carefully addressed this may cause extraordinary costs or closing down of projects. Our research work provides an extensive, but not complete, list of issues that have to handled by the organization together with possible solution approaches. Topics addressed by the research cover development, usage, evolution and management of a framework. Examples are domain scooping of the framework, business models, verification of the framework's abstract behavior, when to release a framework, management of application development, software error correcting strategies, and maintenance strategies.

- *Identification of problems, causes and solutions when composing object-oriented framework.* We have identified five framework composition problems encountered in application development; composition of framework control, composition with legacy components, framework gap, overlap of framework entities and composition of entity functionality. The primary causes for these problems are related to; the cohesion between classes inside each framework, the domain coverage of the frameworks, the design intentions of the framework designers and the potential lack of access to the source code of the frameworks. The problems and their causes identified can be, up to some extent, addressed by existing solutions, such as the adapter design pattern, wrapping or mediating software, but these solutions generally have limitations and either do not solve the problem completely or require considerable implementation effort.

- *Characterization of framework evolution*. A characterization of the evolution of an industrial application framework is given. The characterization of the evolution is mainly quantitative and covers four consecutive versions of the framework. A set of framework stability indicators, defining a set of threshold values for the metrics used, is provided which characterizes the stability of an evolving framework. The threshold values are based on analysis of two additional frameworks, besides the BGW framework. In addition, a method for identifying evolution-prone modules in a framework as well as effort data which shows the distribution of development effort for the four versions is presented as other aspects of framework evolution. To summarize, a characterization of the evolution of an industrial application framework is given from different views, using different methods, which hopefully will provide a better understanding of the nature of framework evolution and thereby assist management to perform a controlled and predictable evolution of the framework's functionality and costs.

- *Extension and validation of two proposed methods for software evolution*. The research work covers the extension/adaptation and application of two proposed methods for software evolution; one method for quantitatively assessing stability of a framework [3], which have been extended with an aggregated metric as well as a set of framework stability indicators defining threshold values for the metric set used that characterizes the stability of a framework, the method have been applied on three different frameworks, another method for identifying evolution-prone modules based on historical information [13], the method have been adapted to suit object-oriented software. The result of our application of the two extended/adapted methods are the first one is possible to apply and use for assessing the stability of a framework, the latter method was adapted and successfully applied on a proprietary framework. Thus, we have validated that the two extended/adapted proposed methods are usable and possible to apply on industrial object-oriented frameworks.

- *Evidence for reduced application development effort when using framework technology.* It has for a long time been claimed that object-oriented framework technology delivers reduced time to market and reduced application development efforts but no quantitative evidence has been shown. Our research work contains quantitative support, based on more than 30 instantiations of an industrial framework, that framework technology delivers reduced application development efforts. Our work shows that the average application development effort is less than 2% of the framework development effort. In fact, 75 percent of the framework customizations requires an effort less than 2,5 percent of the framework development effort. Thus, we have provided, to the best of our knowledge, the first quantitative evidence that framework technology reduces application development effort.

Summary of papers

The abstracts of each included paper are provided below as a quick index for the reader.

Paper I: Frameworks Problems and Experiences
Jan Bosch, Peter Molin, Michael Mattsson, PerOlof Bengtsson and Mohamed Fayad
Chapter in "*Building Application Frameworks: Object Oriented Foundations of Framework Design*" Eds. M. E. Fayad, D. C. Schmidt, R. E. Johnson, Wiley & Sons, ISBN#: 0-471-24875-4, pp. 55-82, 1999

Reuse of software has been one of the main goals of software engineering for decades. Reusing software is not simple and most efforts resulted in small reusable, black-box components. With the emergence of the object-oriented paradigm, the enabling technology for reuse of larger components became available and resulted in the definition of object-oriented frameworks. Frameworks attracted attention from many researchers and software engineers and frameworks have been defined for a large variety of domains. The claimed advantages of frameworks are, among others, increased reusability and reduced time to market for applications. Although several exam-

ples have shown these advantages to exist, there are problems and hindrances associated with frameworks that may not appear before they are used in real projects. The authors of this chapter have been involved in the design, maintenance and use of several object-oriented frameworks, and based on the experiences from these projects, they describe a number of problems related to frameworks. The problems are organized according to four categories: framework development, usage, composition, and maintenance. For each category, the authors present the most relevant problems and experiences. The goal of Chapter 3 is to help software engineers avoid the described problems and to suggest topics for future research.

Paper II: Composition Problems, Causes, & Solutions
Michael Mattsson and Jan Bosch.
Chapter in "*Building Application Frameworks: Object Oriented Foundations of Framework Design*" Eds. M. E. Fayad, D. C. Schmidt, R. E. Johnson, Wiley & Sons, ISBN#: 0-471-24875-4, pp. 467- 487, 1999

Whereas framework-based application development initially included a single framework, increasingly often multiple frameworks are used in application development. This chapter provides a significant collection of common framework composition problems encountered in application development together with an analytical discussion of the underlying causes for the problems. Based on the composition problems and the identified primary causes the authors describe and analyze existing solution approaches and their limitations.

The composition problems discussed are (1) composition of framework control, (2) composition with legacy components, (3) framework gap, (4) overlap of framework entities and (5) composition of entity functionality. The primary causes for these composition problems are related to (1) the cohesion between classes inside each framework, (2) the domain coverage of the frameworks, (3) the design intentions of the framework designers and (4) the potential lack of access to the source code of the frameworks. The identified problems and their causes are, up to some extent, addressed by existing solutions, such as the adapter design pattern, wrapping or mediating software, but these solutions generally have limitations and either

do not solve the problem completely or require considerable implementation efforts.

Paper III: Evolution Observations of an Industrial OO Framework
Michael Mattsson and Jan Bosch
Proceedings of the *International Conference on Software Maintenance*, ICSM '99, pp. 139-145, Oxford, England, 1999

Recently an approach for identifying potential modules for restructuring in large software systems using product release history was presented [13]. In this study we have adapted the original approach to better suit object-oriented frameworks and applied it to an industrial black-box framework product in the telecommunication domain. Our study shows that using historical information as a way of identifying structural shortcomings in an object-oriented system is viable and useful. The study thereby strengthens the suggested approach in [13] and demonstrates that the approach is adaptable and useful for object-oriented systems. The usefulness of the original approach has been validated through this study too.

Paper IV: Stability Assessment of Evolving Industrial Object-Oriented Frameworks
Michael Mattsson and Jan Bosch
Accepted with minor revisions to the *Journal of Software Maintenance*, 1999

Object-oriented framework technology has become a common reuse technology in software development. As with all software, frameworks evolve over time. Once the framework has been deployed, new versions of a framework potentially cause high maintenance cost for the products built with the framework. This fact in combination with the high costs of developing and evolving a framework make it important for organizations to achieve controlled and predictable evolution of the framework's functionality and costs. We present a metrics-based framework stability assessment method, which have been applied on two industrial frameworks, from the telecommunication and graphical user interface domains. First, we discuss the framework concept and the frameworks studied. Then, the stability assessment method is presented including the metrics

used. The results from applying the method as well as an analysis of each of the frameworks are described. We continue with a set of observations regarding the method, including framework differences that seem to be invariant to the method. A set of framework stability indicators based on the results is then presented. Finally, we assess the method against issues related to management and evolution of frameworks, framework deployment, change impact analysis, and benchmarking.

Paper V: Effort Distribution in a Six Year Industrial Application Framework Project
Michael Mattsson
Proceedings of the *International Conference on Software Maintenance*, ICSM '99, pp. 326-333, Oxford, England, 1999

It has for a long time been claimed that object-oriented framework technology delivers reduced application development efforts but no quantitative evidence has been shown. In this study we present quantitative data from a six year object-oriented application framework project in the telecommunication domain. During six years four major versions of the application framework have been developed and over 30 installations at customer sites has been done. We present effort data both for the framework development and the installations as well as relative effort per phase for the framework versions. The effort data presented gives quantitative support for the claim that framework technology delivers reduced application development efforts. In fact, the effort data shows that the average application development effort is less than 2% of the framework development effort.

Paper VI: Assessment of Three Evaluation Methods for Object-Oriented Framework Evolution
Michael Mattsson and Jan Bosch
Reserach report 1999:20, Department of Software Engineering, University of Karlskrona/Ronneby, Sweden, also submitted to the *Journal Information and Software Technology*, 1999

Object-oriented framework technology has become a common reuse technology in object-oriented software development. As with

all software, frameworks tend to evolve. Once the framework has been deployed, new versions of a framework cause high maintenance cost for the products built with the framework. This fact in combination with the high costs of developing and evolving an object-oriented framework make it important to have controlled and predictable evolution of the framework's functionality and costs. We present three methods 1) Evolution Identification Using Historical Information, 2) Stability Assessment and 3) Distribution of Development Effort which have been applied to between one to three different frameworks, both in the proprietary and commercial domain. The methods provide management with information which will make it possible to make well-informed decisions about the framework's evolution, especially with respect to the following issues; identification of evolution-prone modules, framework deployment, change impact analysis, benchmarking and requirements management. Finally, the methods are compared to each other with respect to costs and benefits.

References

[1] Adrion, W. R., Research Methodology in Software Engineering, in "Summary of the Dagstuhl Workshop on Future Directions in Software Engineering", Eds. Tichy, Habermann, Prechelt, *ACM Software Engineering Notes*, SIGSoft, 18(1), pp. 36-37, 1993

[2] Apple Computer Inc., *MacAppII Programmer's Guide*, 1989

[3] Bansiya, J., Evaluating Application Framework Architecture Structural and Functional Stability, chapter in *Building Application Frameworks: Object Oriented Foundations of Framework Design*, Eds. M. E. Fayad, D. C. Schmidt, R. E. Johnson, Wiley & Sons, ISBN#: 0-471-24875-4, 1999, pp. 599-616

[4] Birtwhistle, G. M., Dahl, O-J., Myrhaugh, B., Nygaard, K., *Simula Begin*, Petrocelli/Charter, 1973

[5] Beck, K., Johnson, R., Patterns Generate Architectures, In *Proceedings of the 8th European Conference on Object-Oriented Programming*, Bologna, Italy, 1994, pp. 139-149

[6] Beck, K., Crocker, R., Meszaros, G., Vlissides, J., Coplien, J., Dominick, L., Paulisch, F. Industrial experience with design patterns, *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, 1996, pp. 103-114

[7] Bosch, J., Measurement Systems Framework, chapter in *Domain-Specific Application Frameworks: Frameworks Experience by Industry*, M. E. Fayad, D. C. Schmidt, R. E. Johnson, Wiley and Sons, 1999, pp. 117-206

[8] Cotter, S., Potel, M., *Inside Taligent Technology*, Addison-Wesley, 1995

[9] Deutsch, L. P., Design Reuse and Frameworks in the Smalltalk-80 system, In Biggerstaff, T. J., Perlis, A. J., editors, *Software Reusability*, Vol II, ACM Press, 1989, pp. 57-71

[10] Erni, K., Lewerentz, C., Applying design metrics to object-oriented frameworks, *Proceedings of the Metrics Symposium, 1996*, pp. 64-74

[11] Fayad, M., Schmidt, D. C., Object-oriented application frameworks, *Communications of the ACM*, 40 (10), Oct. 1997, Pages 32 - 38

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

[13] Gall, H., Jazayeri, M., Klösch, R. G., Trausmuth, G., Software Evolution Observations Based on Product Release History, *Proceedings of the Conference on Software Maintenance - 1997*, 1997, pp. 160-166

[14] Glass, R. L., The Software Research Crisis, *IEEE Software*, November 1994, pp. 42-47, 1994

[15] Goldberg, A., *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, 1984

[16] van Gurp, J., Bosch, J., Design, Implementation and Evolution of Object Oriented Frameworks: Concepts & Guidelines, *submitted* to the journal Theory and Practice of Object Systems, 1999

[17] Johnson, R. E., Foote, B., Designing Reusable Classes, *Journal of Object-Oriented Programming*, Vol. 1, No. 2, June 1988, pp. 23-35

[18] Johnson, R. E., *Reusing Object-Oriented Design*, University of Illinois, Technical Report UIUCDCS 91-1696, 1991

[19] Johnson, R. E., Documenting Frameworks with Patterns, *Proceedings of the 7th Conference on Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, 1992, pp. 63-76

[20] Johnson, R. E., Frameworks = (Components + Patterns), *Communications of the ACM*, 40 (10), Oct. 1997, pp. 39 - 42, 1997

[21] Karlsson, E-A, Editor, *Software Reuse - a Holistic Approach*, John Wiley & Sons, 1995

[22] Krasner, G. E., Pope, T. S., A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, *Journal of Object-Oriented Programming*, Vol. 1, No. 3, August-September 1988, pp. 26-49

[23] Lim, W., Reuse economics: A comparison of seventeen models and directions for future research. *Proceedings of the Fourth International Conference on Software Reuse*, pp 41-50, 1996

[24] Mattsson, M., *Object-Oriented Frameworks - A survey of methodological issues*, Licentiate Thesis, Department of Computer Science, Lund University, CODEN: LUTEDX/(TECS-3066)/1-130/(1996), also as Technical Report, LU-CS-TR: 96-167, Department of Computer Science, Lund University, 1996

[25] Mattsson, M., Effort Distribution in a Six Year Industrial Application Framework Project, *Proceedings of the International Conference on Software Maintenance*, ICSM '99, pp. 326-333, Oxford, England, 1999

[26] Mattsson, M., Bosch, J., Composition Problems, Causes, & Solutions, chapter in "*Building Application Frameworks: Object Oriented Foundations of Framework Design*" Eds. M. E. Fayad, D. C. Schmidt, R. E. Johnson, Wiley & Sons, ISBN#: 0-471-24875-4, 1999, pp. pp. 467- 487

[27] Mattsson M., Bosch J., Framework Composition: Problems, Causes and Solutions, *Proceedings of the 23rd International Conference in Technology of Object-Oriented Languages and Systems*, TOOLS '97 USA, Santa Barbara, California, July 28 - August 1, 1997, pp. 203-214

[28] Mattsson, M., J. Bosch, J., Frameworks as Components: A Classification of Framework Evolution, *Proceedings of Nordic Workshop on Programming Environment Research*, NWPER´98, Ronneby, Sweden, August 1998, pp. 163-174

[29] Mattsson, M., Bosch, J., Assessment of Three Evaluation Methods for Object-Oriented Framework Evolution, *submitted* to the Journal of Information and Software Technology, 1999

[30] Mattsson, M., Bosch, J., and Fayad, M. E., Framework Integration Problems, Causes and Solutions, *Communications of ACM*, Vol. 42, No. 10 (Oct. 1999), 1999, pp. 80-87

[31] McIlroy, M. D., Mass produced software components, in Software engineering; *report on a conference by the Nato Science Committee*, (Garmisch, Germany). Naur, P. and Randell, B., Eds. NATO Scientific Affairs Division, Brusssels, Belgium, pp 138-159, 1968

[32] Molin, P., Ohlsson, L. , Points & Deviations -A pattern language for fire alarm systems, *Proceedings of the 3rd International Conference on Pattern Languages for Programming*, Monticello IL, USA, 1996, pp. 431-445

[33] Ohlsson, L., The Next Generation of OOD, *Object Magazine*, May-June 1993

[34] Opdyke, W. F., Johnson, R. E., Refactoring: An aid in designing object-oriented application frameworks, *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990

[35] Opdyke, W. F., *Refactoring Object-Oriented Frameworks*, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992

[36] Lundberg, C., Mattsson, M., On Using Legacy Software Components with Object-Oriented Frameworks, *Proceedings of Systemarkitekturer'96*, Borås, Sweden, 1996

[37] Pree, W., Meta Patterns - A means for capturing the essential of reusable object-oriented design, *Proceedings of the 8th European Conference on Object-Oriented Programming*, Bologna, Italy, 1994, pp. 150-163

[38] Pree, W., *Design Patterns for Object-Oriented Software Development*, Addison-Wesley ACM Books, 1994

[39] Pree, W., Koskimies, K., Rearchitecting Legacy systems - Concepts and Case study, *First Working IFIP Conference on Software Architecture* (WICSA '99), pp 51-61, San Antonio, Texas, 22-24 February 1999

[40] Roberts, D., Johnson, R., Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks, *Proceedings of the Third Conference on Pattern Languages and Programming*, Montecillio, Illinois, 1996, pp. 471-486

[41] Robson, C. *Real World Research*, Blackwell, 1993

[42] Rumbaugh J., Jacobson I., Booch G., *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998

[43] Russo, V. F., *An Object-Oriented Operating System*, Ph.D. thesis, University of Illinois at Urbana-Champaign, October 1990

[44] Schäfer, W., Prieto-Diaz, Matsumoto, R., *Software Reusability*, Ellis-Horwood Ltd., 1994

[45] Shepherd, G., Wingo, S., *MFC Internals: Inside the MFC Architecture*, Addison-Wesley, 1994

[46] Sommerville, I. , *Software Engineering*, 5th edition, Addison-Wesley, 1996.

[47] Svahnberg, M., Bosch J., Evolution in Software Product Lines: Two Cases, to appear in *Journal of Software Maintenance*, 1999.

[48] Taligent Inc., *Building Object-Oriented Frameworks*, A Taligent White Paper, 1994

[49] Tichy, W. F. Should Computer Scientists Experiment More?, *IEEE Computer*, Vol. 31, No. 5, pp. 32-40, May 1998

[50] Walsh Sanderson, S, Uzumeri, M, *The innovation imperative - Strategies for managing product models and families*, Irwin Professional Publishing, 1997

[51] Zelkowitz, M. V., Wallace, D. R., Experimental Models for Validating Computer Technology, *IEEE Computer*, May 1998, pp. 23-31, 1998

I

PAPER I

Framework Problems and Experiences

Jan Bosch, Peter Molin, Michael Mattsson, PerOlof Bengtsson, and Mohamed E. Fayad

Chapter in “Building Application Frameworks: Object Oriented Foundations of Framework Design" Eds: M. E. Fayad, D. C. Schmidt, R. E. Johnson, Wiley & Sons, ISBN#: 0-471-24875-4, 1999, pp. 55-82

Abstract

Reuse of software has been one of the main goals of software engineering for decades. Reusing software is not simple and most efforts resulted in small reusable, black-box components. With the emergence of the object-oriented paradigm, the enabling technology for reuse of larger components became available and resulted in the definition of object-oriented frameworks. Frameworks attracted attention from many researchers and software engineers and frameworks have been defined for a large variety of domains. The claimed advantages of frameworks are, among others, increased reusability and reduced time to market for applications. Although several examples have shown these advantages to exist, there are problems and hindrances associated with frameworks that may not appear before they are used in real projects. The authors of this chapter have been involved in the design, maintenance and use of several object-oriented frameworks, and based on the experiences from these projects, they describe a number of problems related to frameworks. The problems are organized according to four categories: framework development, usage, composition, and maintenance. For each category, the authors present

the most relevant problems and experiences. The goal of Chapter 3 is to help software engineers avoid the described problems and to suggest topics for future research.

1. Introduction

Reuse of software has been a goal in software engineering for almost as long as the existence of the field itself. Several research efforts have aimed at providing reuse. During the 1970s, the basics of module-based programming were defined and software engineers understood that modules could be used as reusable components in new systems. Modules, however, only provided as-is reuse, and adaptation of modules had to be done either by editing the code or by importing the component and changing those aspects unsuitable for the system at hand. During the 1980s, the object-oriented languages increased in popularity, since their proponents claimed increased reuse of object-oriented code through inheritance. Inheritance, different from importing or wrapping, provides a much more powerful means for adapting code.

However, all these efforts only provided reuse at the level of individual, often small-scale, components that could be used as the building blocks of new applications. The much harder problem of reuse at the level of large components that may make up the larger part of a system, and of which many aspects can be adapted, was not addressed by the object-oriented paradigm in itself. This understanding led to the development of object-oriented frameworks (large, abstract applications in a particular domain that can be tailored for individual applications). A framework consists of a large structure that can be reused as a whole for the construction of a new system.

Since its conception at the end of the 1980s, the appealing concept of object-oriented frameworks has attracted attention from many researchers and software engineers. Frameworks have been defined for a large variety of domains, such as user interfaces and operating systems within the computer science domain and financial systems, fire-alarm systems, and process control systems within particular application domains. Large research and development projects were started within software development companies, but also at universi-

ties and even at the governmental level. For instance, the European Union-sponsored Esprit project REBOOT [25] had a considerable impact on the object-oriented thinking and development in the organizations involved in the project and later caused the development of a number of object-oriented frameworks, for example, see [10].

In addition to the intuitive appeal of the framework concept and its simplicity from an abstract perspective, experience has shown that framework projects can indeed result in increased reusability and decreased development effort; see, for example, [38]. However, next to the advantages related to object-oriented frameworks, there exist problems and difficulties that do not appear before actual use in real projects. The authors of this chapter have been involved in the design, maintenance, and usage of a number of object-oriented frameworks. During these framework-related projects several obstacles were identified that complicated the use of frameworks or diminished their benefits.

The topic of this chapter is an overview and discussion of the obstacles identified during these framework projects. The obstacles have been organized into four categories. The first category, framework development, describes issues related to the initial framework design, in other words, the framework until it is released and used in the first real application. The second category is related to the instantiation of a framework and application development based on a framework. Here, issues such as verification, testing, and debugging are discussed. The composition of multiple frameworks into an application or system and the composition of legacy code with a framework is the third category of concern. The final category is concerned with the evolution of a framework over time, starting with the initial framework design and continuing with the subsequent framework versions.

This chapter provides a solid analysis of obstacles in object-oriented framework technology. It presents a large collection of the most relevant problems and provides a categorization of these obstacles. This chapter is of relevance to practitioners who make use of frameworks or design them, as well as to researchers, since it may provide topics to be addressed in future research efforts.

The remainder of this chapter is organized as follows. The next section describes the history of object-oriented frameworks and defines a consistent terminology. Subsequently, Section 3 discusses

some examples of object-oriented frameworks in which one or more of the authors were involved, either as designers or as users. Section 4 discusses the obstacles in object-oriented framework development and usage that were identified, organized in the four aforementioned categories. Section 5 summarizes the chapter.

2. Object-Oriented Frameworks

Although object-oriented frameworks present a well-known concept in the reuse community, no widely accepted terminology is available. After discussing a brief history of object-oriented frameworks, this section presents the definitions used in the remainder of the chapter.

2.1 History of frameworks

Early examples of the framework concept can be found in literature that has its origins in the Smalltalk environment, such as [17] and Apple Inc. [46]. The Smalltalk-80 user interface framework, Model-View-Controller (MVC), was perhaps the first widely used framework. Apple Inc. developed the MacApp user interface framework, which was designed for supporting the implementation of Macintosh applications. Frameworks attained more interest when the InterViews [30] and ET++ [50] user interface frameworks were developed and became available. Frameworks are not limited to user interface frameworks but have been defined for many other domains as well, such as operating systems [45] and fire alarm systems [36], [37]. With the formation of Taligent in 1992, frameworks attained interest in larger communities. Taligent set out to develop a completely object-oriented operating system based on the framework concept. The company delivered a set of tools for rapid application development under the name CommonPoint that consists of more than a hundred object-oriented frameworks [1], [9]. The Taligent approach made a shift in focus to many fine-grained integrated frameworks and away from large monolithic frameworks.

Many object-oriented frameworks exist that capture a domain well, but relatively little work has been done on general framework issues such as methods for framework usage and testing of frameworks. Regarding documentation, patterns have been used for docu-

mentation of frameworks [19], [21] and for describing the rationale behind design decisions for a framework [3]. Other interesting work of a general framework nature is that concerning the restructuring (refactoring) of frameworks. Since frameworks often undergo several iterations before even the first version is released, the framework design and code change frequently. In [39], a set of behavior-preserving transformations-refactorings-are defined that help to remove multiple copies of similar code without changing the behavior. Refactoring can be used for restructuring inheritance hierarchies and component hierarchies [23].

[43] describe the evolution of a framework as starting from a whitebox framework, a framework that is reused mostly by subclassing, and developing into a blackbox framework, a framework that mostly is reused through parameterization. The evolution is presented as a pattern language describing the process from the initial design of a framework as a whitebox framework to a blackbox framework. The resulting blackbox framework has an associated visual builder that will generate the application's code. The visual builder allows the software engineer to connect the framework objects and activate them. In addition, the builder supports the specification of the behavior of application-specific objects.

2.2 Definition of concepts

Most authors agree that an object-oriented framework is a reusable software architecture comprising both design and code, but no generally accepted definition of a framework and its constituent parts exists. Probably the most referenced definition of a framework is found in [22]:

A framework is a set of classes that embodies an abstract design for solutions to a family of related problems.

In other words, a framework is a partial design and implementation for an application in a given problem domain. When discussing the framework concept, terminological difficulties may arise due to the fact that a common framework definition does not exist and because it is difficult to distinguish between framework-specific and application-specific aspects. In the remainder of the chapter, the following concepts are used: core framework design, framework internal increment, application-specific increment, object-oriented

framework, and application. For more debate about the framework definition, please refer to Chapter 1 and [12], [14].

The *core framework design* comprises both abstract and concrete classes in the domain. The concrete classes in the framework are intended to be invisible to the framework user (for example, a basic data storage class). An abstract class is either intended to be invisible to the *framework user* or intended to be subclassed by the framework user. The latter classes are also referred to as *hot spots* [42]. The core framework design describes the typical software architecture for applications in the domain.

However, the core framework design has to be accompanied by additional classes to be more usable. These additional classes form a number of class libraries, referred to as *framework internal increments*, to avoid confusion with the more general class library concept. These internal increments consist of classes that capture common implementations of the core framework design. Two common categories of internal increments that may be associated with a core framework design are the following:

Subclasses representing common realizations of the concepts captured by the superclasses. For example, an abstract superclass Device may have a number of concrete subclasses that represent real-world devices commonly used in the domain captured by the framework.

A collection of (sub)classes representing the specifications for a complete instantiation of the framework in a particular context. For example, a graphical user interface framework may provide a collection of classes for a framework instantiation in the context provided by Windows 95.

At the object level we talk about the *core implementation*, which comprises the objects belonging to the classes in the *core framework design* and *increment implementation*, which consists of the objects belonging to the classes defined in the internal increments. Thus, an *object-oriented framework* consists of a core framework design and its associated internal increments (if any) with accompanying implementations. Different from [43], where the distinction between the framework and a component library is made, our interpretation of a framework includes class libraries.

Some authors categorize frameworks into *whitebox* and *black-box* frameworks (for example, [22]), or *calling* and *called* frameworks (for example, [49]). In a whitebox (inheritance-based) framework,

the framework user is supposed to customize the framework behavior through subclassing of framework classes. As identified by [Roberts-Johnson 1996], a framework often is inheritance-based in the beginning of its life cycle, since the application domain is not sufficiently well understood to make it possible to parameterize the behavior. A blackbox (parameterized) framework is based on composition. The behavior of the framework is customized by using different combinations of classes. A parameterized framework requires deep understanding of the stable and flexible aspects of the domain. Due to its predefined flexibility, a blackbox framework is often more rigid in the domain it supports. A calling framework is an active entity, proactively invoking other parts of the application, whereas a called framework is a passive entity that can be invoked by other parts of the application. However, in practice, no framework is a pure whitebox or blackbox framework or a pure calling or called framework. In general, a framework has parts that can be parameterized and parts that need to be customized through subclassing. Also, virtually each framework is called by some part of the application and calls some (other) part of the application.

An *application* is composed of one or more core framework designs, each framework' s internal increments (if any), and an application-specific increment, comprising application-specific classes and objects. The application may reuse only parts of the object-oriented framework or it may require adaptation of the core framework design and the internal increments for achieving its requirements.

The presence of reusable frameworks influences the development process for the application. The following phases are identifiable in framework-centered software development:

The framework development phase, often the most effort-consuming phase, is aimed at producing a reusable design in a domain. Major results of this phase are the domain analysis model, a core framework design, and a number of framework internal increments, as depicted in Table 1. The framework development phase is described in more detail in Section 4.1.

The framework usage phase is sometimes also referred to as the *framework instantiation phase* or the *application development phase*. The main result of this phase is an application developed reusing one or more frameworks. Here, the framework user must include the core framework designs or part of them, depending on the application

requirements. After the application design is finished, the software engineer has to decide which internal increments to include. For those parts of the application design not covered by reusable classes, new classes need to be developed to fulfill the actual application requirements. We refer to these new classes as the *application-specific increment*. The problem of composing frameworks is further discussed in Section 4.3, and a model for framework usage is outlined in Section 4.2.

The framework evolution and maintenance phase, as all software, will be subject to change. Causes for these changes can be errors reported from shipped applications, identification of new abstractions due to changes in the problem domain, changes in the business domain, and so on. These kinds of issues will be discussed in Section 4.4.

Table 1. Framework development activities, software engineer roles, and artifacts

Activity	The Software Engineer's Role	Software Artifacts Involved
Framework development	Framework developer	Core framework design Framework internal increments
Framework usage[a]	Framework user[b]	Core framework design Framework Internal increments Application-specific increment
Framework maintenance and evolution	Framework maintainer	Core framework design Framework increments

a.Framework usage is sometimes referred to as framework instantiation or application development.

b.By some, the framework user is denoted as framework reuser or application developer.

3. Examples of Application Frameworks

The problems and lessons learned that are presented in the next section are based on extensive experience with industry-strength object-oriented frameworks. In this section, the five most prominent frameworks are briefly presented. Several of the frameworks are used in product development, whereas others have not reached that stage yet.

3.1 Fire alarm systems

TeleLarm AB, a Swedish security company, develops and markets a family of fire alarm systems ranging from small office systems to large distributed systems for multibuilding plants. An object-oriented framework was designed as a part of a major architectural redesign effort. The framework provides abstract classes for devices and communication drivers, as well as application abstractions such as input points and output points representing sensors and actuators. System status is represented by a set of deviations that are available on all nodes in the distributed system. The notion of periodic objects is introduced, giving the system large-grain concurrency without the problems associated with asynchronous fine-grain concurrency. Two production instantiations from the framework have been released so far. For more information, refer to [35], [37].

3.2 Measurement systems

In cooperation with EC-Gruppen, we were involved in the design of a framework for measurement systems. Measurement systems are systems that are located at the beginning or end of production lines to measure some selected features of production items. The hardware of a measurement system consists of a trigger, one or more sensors, and one or more actuators. The framework for the software has software representations for these parts, the measurement item, and an abstract factory for generating new measurement item objects. A typical measurement cycle for a product starts with the trigger detecting the item and notifying the abstract factory. In response, the abstract factory creates a measurement item and activates it. The measurement item contains a measurement strategy and an actuation strategy and uses

them first to read the relevant data from the sensors, then to compare this data with the ideal values, and subsequently to activate the actuators when necessary to remove or mark the product if it did not fulfill the requirements. Refer to [7] for a more detailed description of the framework.

3.3 Gateway billing systems

Ericsson Software Technology AB started to develop the billing gateway framework as an experiment and it is now used for developing products within the company. The billing gateway framework acts as a mediation device in telecommunication management networks between the network elements generating billing information and the billing systems responsible for charging customers. The framework supports several network communications protocols, both for inbound and for outbound communication. The gateway provides services for processing the billing information (for example, conversion from different billing information formats), filtering of billing information, and content-sensitive distribution of the information. The performance of applications developed based on the framework is of great importance and multithreading is used heavily. Refer to [31] for a more detailed overview of these issues.

3.4 Resource allocation

The resource allocation framework was part of a large student project (10,000 hours) in cooperation with Ericsson Software Technology AB as a prototype development. The project goal was to develop a framework for resource allocation systems, since a need to develop several similar systems in the domain was identified by the customer. The framework consists of three subframeworks-the core resource allocation framework and two supporting frameworks-handling user interface and persistence through relational databases, respectively. The main part of the generic behavior consists of the general scheme of allocations, in other words, one or several resources could be allocated by one allocator represented by an allocation. The framework user specifies the concrete resources for the application and implements some specific behavior such as editing, storing, and display-

ing. The framework handles the general business rules, for example, not allowing a resource to be allocated by more than one allocator for overlapping time periods.

3.5 Process operation

In cooperation with researchers from a chemical technology department, one of the authors was involved in the definition of an object-oriented framework for process operation in chemical plants. The goal of the framework was to provide a general frame of reference for all (most) activities and entities in a chemical plant, including sales, planning, process operation, and maintenance. The framework consists of seven concept hierarchies organized into three categories: real-world, coordination, and model structures. The real-world structures are the order structure, material structure, and equipment structure. The model structures are the operation step structure, instrumentation structure, and control structure. The only coordination structure is the equipment coordination structure. These structures describe a considerable part of the domain knowledge for chemical plants as available in the field of chemical technology. For a concrete plant, the structure hierarchies have to be instantiated based on the actual equipment, orders, materials, and so forth, available in the plant. Due to the size of the framework, parts of it have been instantiated, rather than the complete framework. One experiment has been performed on a batch process application. We refer to [4] for more details concerning the framework.

4. Problems and Experiences

As described earlier, three phases can be identified in framework-centered software development: framework development, framework usage, and framework evolution. Each phase is discussed in the subsequent sections. Framework usage, being the most extensive phase, is discussed in two sections, that is, framework usage and framework composition. Framework composition is treated separately since it leads to a number of specific problems.

4.1 Framework development

The development of a framework is somewhat different from the development of a standard application. The important distinction is that the framework has to cover all relevant concepts in a domain, whereas an application is concerned only with those concepts mentioned in the application requirements. To set the context for the problems experienced and identified in framework development, we outline the following activities as parts of a simple framework development model:

Domain analysis [47] aims at describing the domain that is to be covered by the framework. To capture the requirements and identification of concepts, the developer may refer to previously developed applications in the domain, to domain experts, and to existing standards for the domain. The result of the activity is a domain analysis model, containing the requirements of the domain, the domain concepts, and the relations between those concepts.

Architectural design takes the domain analysis model as input. The designer has to decide on a suitable architectural style to underlie the framework. Based on the selected style, among other considerations, the top level of the framework is designed. Examples of architectural styles or patterns can be found in [48] and [8].

Framework design is the stage in which the top-level framework design is refined and additional classes are designed. Results from this activity are the functionality scope given by the framework design, the framework's reuse interface (similar to the *external framework interface* described in [11]), design rules based on architectural decisions that must be obeyed, and a design history document describing the design problems encountered and the solutions selected, with an argumentation.

Framework implementation is concerned with the coding of the abstract and concrete framework classes.

Framework testing is performed to determine whether the framework provides the intended functionality, but also to evaluate the usability of the framework. It is, however, far from trivial to decide whether an entity is usable or not. [24] conclude that the only way to find out if software is reusable is to reuse it. For frameworks, this requires the development of applications that use the framework.

To evaluate the usability of the framework, the test application generation activity is concerned with the development of test applications based on the framework. Depending on the kind of application, the developer can test different aspects of the framework. Based on the developed applications, application testing aims at deciding whether the framework needs to be redesigned or that it is sufficiently mature for release.

Documentation is one of the most important activities in framework development, although its importance is not always recognized. Without a clear, complete, and correct documentation that describes how to use the framework, a user manual, and a design document that describes how the framework works, the framework will be nearly impossible to use by software engineers not involved in the framework design.

The remainder of this section discusses a number of problems encountered during framework development. These problems are related to the domain scope, framework documentation, business models for framework domains, and framework development, as well as to framework testing and releasing.

Domain scope

When deciding to develop a framework for a particular domain, there is the problem of determining the right size for the domain. On one hand, if the domain is too large, the development team does not have enough experience from the enlarged domain. In addition, it may be difficult to demonstrate the usefulness and applicability of a large domain framework. Also, the (financial) investment in a large framework may be so high that it becomes very difficult to obtain the necessary resources. Finally, the duration of the project may be such that the intended reuse benefits cannot be achieved in a reasonable amount of time.

On the other hand, a smaller domain uses the known experiences in a much more efficient way. One problem is, however, that the resulting framework tends to be sensitive to domain changes. For instance, with a narrow framework, an application may easily expand over the framework boundaries, requiring more application-specific changes to the framework in order to be useful than when the framework would have covered the complete application.

From the preceding, one can deduce that defining the scope of the domain is a decision that must be carefully balanced. One problem is that, because the future is unpredictable, it is very difficult to set clear boundaries for the framework. A second problem is that it is a very natural, human tendency to increase the size of the framework during (especially early) design because a framework that includes yet another aspect seems more useful than one that lacks that aspect (see, for example, [7]).

In addition to these problems, selecting the right domain is a difficult issue. A framework must not be so general that it constitutes overkill for the intended application [12]. A one-size-fits-all approach introduces unnecessary risk. The most suitable frameworks for a particular domain will reflect a mature, yet narrow, focus on a particular problem or group of problems [12].

Framework documentation

The purpose of framework documentation is twofold. First, information about the framework design and other related information need to be communicated during the framework development. Second, information on how to use the framework needs to be transmitted to the framework user.

In the first case, it is convenient to rely on informal communication to spread the knowledge about the developed framework. The often small design group uses different ad hoc techniques to exchange information among the individuals in the team. The problem is that these techniques are not very efficient and do not always assure that the correct information is spread.

The second case is more important, since the informal information channels are not available and all information has to be communicated in a way that is possible to deliver with the framework. The current way to do this is by means of different kinds of manuals, using different media. The documentation is usually ad hoc, making it hard to understand and compare. [21] argues that framework documentation should contain the purpose of the framework, information on how to use the framework, the purpose of the application examples, and the actual design of the framework.

For a framework, whitebox or blackbox, users need to understand the underlying principles or basic architecture of the framework.

Otherwise, detailed rules and constraints defined by the framework developers make no sense and the framework will probably not be used as intended. Examples of these rules and constraints are cardinality of framework objects, creation and destruction of static and dynamic framework objects, instantiation order, synchronization, and performance issues. These rules and constraints are often implicitly hidden or missing in existing documentation, and they need to be elucidated and documented. The problem is how to convey this information in a concise form to framework users.

A number of methods have been proposed for documenting frameworks. A first example is the cookbook approach, such as in [27] and [2], which are example based, and the metapatterns approach proposed by [42]. The pattern approaches provide a second example, in which patterns are used to describe the architecture of the framework [3], [19] or the use of the framework [21], [28]. Finally, a framework description language (FDL) [51] can be used that more formally describes, for example, what classes to override and subclass. All approaches address some of the documentation needs, but, we believe, no approach covers all the aforementioned needs for framework documentation. Most of these methods address the issues of purpose, how to use the framework, and the purpose of the example application, but none of these methods address how to communicate the framework design to the user.

There are two more serious problems with framework documentation:

The cost of providing suitable framework documentation is very high. Defining and documenting application frameworks is neither quick nor cheap. The costs of documenting application frameworks can be a high risk. However, it is impossible to extend or customize application frameworks or build new applications without using well-documented frameworks. Defining suitable methods for framework documentation is an important first step. We have found that combining several documentation approaches, such as patterns, hooks, hot spots, examples, and architectural illustrations, was very effective.

A framework documentation standard doesn't exist. Mapping framework documentation approaches to software documentation standards can be difficult. The difficulty is usually traced to the term

standard, implying that it is "applicable for all frameworks except mine" [13].

Business models

Even though it may be feasible from a technological perspective to develop a framework for a particular domain, it is not necessarily advantageous from a business perspective. That is, the investments necessary for the framework development may be larger than the benefits in terms of reduced development effort for applications built using the framework. The return on investment from a developed framework may come from selling the framework to other companies, but it often, to a large extent, relies on future savings in development effort within the company itself, such as higher software quality and shorter lead times. One of the main problems for the management of software development organizations or departments is that, to the best of our knowledge, no reliable business models for framework development exist.

A formulation of a business model is given by [26], but this has never been tested in an industrial setting. Other general reuse business models have been proposed by several authors, but none satisfy all relevant requirements as shown in [29]. Since no reliable investment models exist, decisions on framework development are often based on gut feeling, with the downside that many framework designs never come to fruition since the technical staff is unable to convince management that framework development would be economically viable.

Another major problem is related to the business culture issues. Business culture can be categorized into several nested and difficult cultures. [13] discuss several tough issues for transition management to object-oriented technology that maps well to application framework and component-based technology. They also state that this culture change will be one of the hardest and longest lasting parts of the transition to a new technology. Once the initial transition is complete, the change will be a continuing part of the business [13].

Framework development methods

Existing development methods do not sufficiently support development of frameworks. Framework design introduces some new concepts that need to be covered by the methods, and some existing concepts need much more emphasis. The domain analysis presents problems unlike those of normal application development, for example, behavior and attributes are very likely to change in several cases. The analysis of such behavior and attributes needs to be emphasized in the development model, since it is the nature of frameworks to provide reusable design and implementation that cover the variations-that is, hot spots-in the domain.

During development of the framework, an architecture must be selected or developed. The criteria for a framework architecture are dependent on the domain and the domain variations. The architecture must provide solutions to problems in the domain without blocking possible variations or different solutions to other domain problems. During framework architecture design, decisions need to be made on whether some parts of the framework should be designed as a subframework or if everything should be designed as a single, monolithic framework. Also, interoperability requirements need to be established, for example, whether the framework should cooperate with other frameworks, such as user interface frameworks or persistence frameworks.

In addition to the existing object-oriented design methods [5], [20], [44], which most often support only inheritance, aggregation, and associations, more emphasis is needed for abstract classes, dynamic binding, type parameterization, hot spots and pre- and postconditions. Abstract classes and dynamic binding represent the way to define and implement abstract behavior that is not emphasized in existing design models. Type parameterization needs support in the methods since this is another useful way of supplying predefined abstract behavior that the framework user then uses by supplying the application-specific parts as parameters. Hot spots are needed to show where the intended framework extensions or application specifics go, and support in the design methods for expressing and designing these are very important. Pre- and postconditions provide the framework developer with the possibility of specifying the restrictions for usage of parts of the framework.

There are other serious issues related to framework development methods:

Application frameworks should be stable over time. Framework stability over time is an important issue. A good framework reflects the concise understanding of the enduring qualities of the business through enduring business themes (EBTs; see Sidebar 6 and [12]). EBTs capture abstractions that do not change over time.

Business objects and workflow management and the issue of stability must be considered. The workflow management metaphor provides the necessary modeling capabilities for constructing enduring business processes (EBPs) [12] or workflows. Well-designed frameworks should capture the workflows that do not change over time [12].

Achieving portability is not a trivial task. Platform independence and portability ensure that the framework supports all the platforms in a given organization. Failing to provide platform independence will diminish the scope of use and acceptance of the framework [12].

Verifying abstract behavior

When developing a framework, the result must be verified. The current practice is to develop test applications using the framework and then test the resulting application. In this way most of the core framework design and the framework internal increments can be tested using conventional methods. However, the versatility of the framework cannot be tested by only one application, especially if the application is for test purposes only. Since a framework can be used in many different, unknown ways, it may simply not be feasible to test all relevant aspects of the framework.

Furthermore, testing the framework for errors using traditional methods, such as executing parts of the code in well-defined cases, will not work for the whole framework-for example, using the core framework design. At best, parts of the framework can be tested this way, but since the framework relies on parts implemented by the users it is not possible to completely test the framework before it is released. This is also referred to as the problem of verifying abstract behavior. Standard testing procedure does not allow for testing of abstract implementations, such as abstract base classes.

Framework release problem

Releasing a framework for application development has to be based on some release criteria. The problem is to define and ensure these criteria for the framework. The framework must be reusable, reasonably stable within the domain, and well documented. As for reusability, several authors have addressed this issue and in [41] several of the proposed approaches are compared. The conclusion of the research was that no general reusability metrics exist. The problem of determining the stability within the domain is twofold. First, the domain is not stable in itself, but evolves constantly. Second, the issue of domain scope and boundaries, discussed earlier in this section, also complicates the decision on framework release. Deciding whether the framework is sufficiently well documented is hard, since no generally accepted documentation method exists that covers all aspects of the framework, but also because it is difficult to determine whether the documentation is understandable for the intended users.

Releasing an immature framework may have severe consequences in the maintenance and usage of the framework and the instantiated applications. This issue is discussed in more detail in Section 4.1.

4.2 Framework usage

Although we have seen that it often is feasible to produce complex applications based on a framework with rather modest development effort, we have also experienced a number of problems associated with the usage of a framework. Before discussing these problems, we believe it is important to provide an outline of how a framework is used. The main purpose of this development method is to relate problems to various activities during the development of an application based on a framework. The method is an adaptation of [33] and consists of the following activities:

- *Requirements analysis* aims at collecting and analyzing the requirements of the application that is to be built using one or more framework(s).

- Based on the results from the requirements of the analysis, a *conceptual architecture* for the application is defined. This includes the association of functionality to components, the specification of the relationships between them, and the organization of collaboration between them.
- Based on the application architecture, one or more frameworks are selected based on the functionality to be offered by the application as well as the nonfunctional requirements. If more than one framework is selected, the software engineer may experience a number of framework composition problems. (These problems are the subject of the next section.)
- When it is decided what framework(s) to reuse, the developer can deduce the application specific that needs to be defined; in other words, the developer specifies the required *application-specific increments* (ASIs).
- After the specification of the ASIs, they need to be *designed* and *implemented*. During these activities, it is crucial to conform to the design rules stated in the framework documentation.
- Before the ASIs and the framework are put together into the final application, the ASIs need to be *tested* individually to simplify debugging and fault analysis in the resulting application.
- Finally, the complete application is verified and tested according to the application's original requirements.

The remainder of this section discusses the problems that we identified and experienced during framework-based application development. The problems are related to framework training and learning, the management of the development process, applicability, estimations, understanding, verification, and debugging.

Training and learning

Learning to use an object-oriented framework effectively can take long time due to the complexity of the framework. Experiences from the use of commercial GUI frameworks such as Microsoft Founda-

tion Classes (MFCs) and MacApp indicate that it takes 6 to 12 months to be productive with the framework [14]. A major reason for this is that a framework is most useful for a developer who understands it in detail. For an in-house developed framework, the learning time can be reduced if one of the framework developers is participating in the application development project and can give hands-on mentoring to the other project members.

Two of the most common training approaches to learning a framework are mentoring and training courses. An advantage with mentoring is that the mentor, one of the framework developers or an experienced user of the framework, is easily available most of the time and can provide information about how to use the framework for particular tasks. However, the problem with the mentor approach is that that kind of highly skilled staff almost always is needed in other projects of higher priority. Regarding training courses, it is extremely important that the application developers be taught concrete application examples, beginning with simple applications and continuing with more and more complex applications that make use of more powerful framework concepts-in other words, a kind of iterative and incremental approach that explains some of the framework's possibilities, uses the framework, explains a little more, uses the framework again, and so on. Thus, this approach illustrates the different possibilities with the framework. A useful approach to organizing the training material is to first describe how to use the framework, to make the framework useful quickly, and not to describe the detailed design of the framework from the beginning. This avoids bothering the framework user with a lot of unnecessary detailed information that is not immediately required for using the framework.

Managing the development process

The most important problem of framework-based application development is how to manage the application development process. Development processes for standard applications are well known and have undergone an evolution from ad hoc approaches and waterfall methods to more elaborate models such as spiral models and evolutionary models. We believe that the current state of the art in frame-

work-based application development is still in the ad hoc stage, where an exploratory style of development is used.

The traditional approach for a development process is to start with a specification or description of the work to be done. That specification is used as a base for estimating costs and resources and is also used as an input to design, implementation, and testing. This information is, of course, also needed when a framework-based application is developed, but in the framework case it is much more difficult to explicitly write such a specification. Examples include specifying the new classes that must be implemented and the classes that require adaptation by subclassing.

Applicability of the framework

One initial difficulty is to understand the intended domain of the framework and its appropriateness for the application under consideration. The challenge here is to be able to make the choice of applying the framework for the application, modifying the framework, if that is an option, or constructing the application from scratch. From an abstract perspective, the question is whether the application matches the domain of the framework. In practice, the question of domain can be rather complex and contains several dimensions. The most obvious dimension of a domain is the functional dimension that corresponds to the way people generally interpret the domain. This dimension defines the functional boundaries of the domain. Another dimension is the underlying hardware architecture. This dimension addresses such aspects as multiprocessor systems versus single processor systems or distribution aspects. A third dimension is the interoperability, under which aspects such as coexistence with other applications are discussed. Issues in this dimension are, for example, whether a particular database engine can or must be used or whether there are any restrictions on the graphical user interface. A fourth dimension incorporates nonfunctional aspects such as performance, fault-tolerance, and capacity. Examples are a framework intended for batch processing and not for real-time data processing and a framework that has an upper limit on the number of items it can handle.

The problem here is to be able to determine, with a reasonable degree of confidence, whether a specific application can be built based on a specific framework and what resources are needed to

implement the application-specific increment. It is clear that all dimensions of the domain must be examined. The problem is even more complex if the framework does not support a certain aspect of the application. In that case, the problem is to determine whether it is possible to use the framework at all and, second, to determine the amount of work needed to implement the required application.

Estimations of the increment

The problem of using frameworks compared to traditional application development is that most of the time spent in such a project could well be spent on understanding the framework, with very little spent on actual coding. From a productivity point of view this may indicate that a framework-based application is slow compared to a traditional approach. On the other hand, complex applications can be built very fast so that real productivity is high. The consequences of traditional estimation techniques based on the number of produced lines of code is inadequate in the framework case. [38] discuss these problems in more detail.

Another problem is the sensitivity of estimates of the amount of work required for a specific application. Our experience indicates that this is also a well-known effect of tool usage that we refer to as the *90 percent/10 percent* rule, implying that 10 percent of the development time is spent on 90 percent of the application, and 90 percent is spent on the remaining difficult 10 percent of the application. The difference depends on whether a specific feature is supported by the tool (framework). Furthermore, it can be difficult to foresee if a specific requirement is completely supported by the framework. If it is fully supported, the implementation will be fast. On the other hand, if it is not at all supported, a potential implementation can mismatch the intentions of the framework designers and, in that case, the resulting effort can be even larger than with a traditional approach. The conclusion is that the required implementation effort is very sensitive to features close to the domain boundary and the framework boundary.

There is a noteworthy catch when an estimation of the required effort is performed. For an accurate estimation, it is necessary to thoroughly investigate the framework and the application, in order to determine what needs to be done. On the other hand, such an investigation could be the major task of the application development! The

same could be true even for traditional application development, but it is much more striking in the framework case.

Understanding the framework

When a whitebox framework is used, it is necessary to understand the concepts and architectural style of the framework in order to develop applications that conform to the framework. The framework understanding may be a prerequisite for evaluating the applicability of a framework and the amount of required adaptation, and how the adaptation can be carried out. One example of such an important concept could be concurrency strategies adopted by the framework. For example, if the framework is intended for multithreading, then any adaptations must adhere to rules on resource locking and shared variable protection defined by the framework. On the other hand, if a single-thread solution is chosen, the adaptation code must conform to certain timing constraints. Other examples are dynamic memory allocation strategies where it is important that the increment follow the same strategy. Especially in real-time frameworks such as the measurement system or the fire alarm system, it is absolutely necessary that the users understand how external events are processed; otherwise, it becomes very difficult to guarantee any application response time. Many errors can be avoided and the application can be constructed more efficiently if the framework user understands these strategies and styles. One problem is how to obtain this information from, for example, the framework documentation, discussed earlier in the chapter.

An alternative that avoids the understanding approach, or serves as a complement to it, is to explicitly specify the constraints that the framework put on the application. This approach has been proposed by [33], [35].

Verification of the application-specific increment

An application based on a complex framework or based on several frameworks can prove difficult to test. Therefore, it could be advisable to verify the application-specific increment beforehand. For large-scale applications, the increment should be locally certifiable; that is, it should be tested or verified locally without executing the

entire application. Such verification must be based on a specification of what requirements the increment needs to fulfill. There are two aspects of verification. The first is a functional verification that verifies that the increment implements the expected behavior of the resulting application. The second aspect is the verification that the increment conforms to the architectural style and design rules imposed by the framework. The trade-off in this case is the amount of effort spent on increment verification compared to the amount of debugging time if problems are detected during the test of the complete application. In cases where straightforward application testing is not sufficient, a more formal increment verification could be useful. These problems have been investigated by [35] and a partial solution has been proposed that suggests verification of conformance to the framework, but not to verify the functionality provided by the increment.

Debugging the application

Traditional debuggers have problems when debugging programs using libraries. Using single-step methods is impossible, and library calls must be skipped in some way. Normally, there is no automatic support for this distinction of code source. The debugger must manually define which part should be skipped and which routines should be followed through. In some cases, there is an exception raised in the library part, either as a result of bad usage of the library or due to a bug in the library code, and it may be difficult to determine the actual reason for the exception.

Frameworks, and especially blackbox frameworks, have the same problem as libraries. Furthermore, since frameworks often are based on the Hollywood principle, the problems are even more difficult. It can be very difficult to follow a thread of execution that mostly is buried under framework code.

One solution approach, based on the design-by-contract ideas, is to exactly define the interface between the framework and the increment as pre- and postconditions or behavioral interaction constraints [18], [28], [34]. Precondition violation causes exceptions to be raised and it can be guaranteed that the framework will never crash as a result of illegal use.

4.3 Framework composition

Traditionally, object-oriented framework-based development of applications takes the approach that the application development is started from the framework and the application is constructed in terms of extensions to the framework. However, this perspective on framework-based development represents only the simplest solution. Often, a framework that is to be reused needs to be composed with other frameworks or with reusable legacy components. Composing frameworks, however, may lead to a number of problems since frameworks generally are designed based on the traditional perspective in which the framework is in full control.

This section discusses the problems related to the composition of reusable components and frameworks with an object-oriented framework. Note that we are concerned with the composition of reusable components and frameworks that were not intended to be composed during their design. This is different from the approach taken in Taligent where mutually compatible frameworks are available that are easy to compose. The composition of reused components that were not designed to work together is much more challenging and it is these kind of problems that we discuss here.

Architectural mismatch

The composition of two or more frameworks that seem to match at first may prove to be much more difficult than expected. The reason for that can often be found in a mismatch in the underlying framework architectures. In [16], this is referred to as *architectural mismatch*, in other words, the architectural styles, based on which the frameworks are designed, are different, complicating composition so much that it may be impossible to compose. [8], [48] identify several architectural styles (or architectural patterns) that can be used to construct an application. For example, if one of the frameworks is based on a blackboard architecture and the other on a layered architecture, then the composition of the two frameworks may require either substantial amounts of glue code or the redesign of parts of one or both of the frameworks. Lately, an increasing interest in the domain of software architecture is apparent. Explicitly specifying the architec-

tural style underlying a framework design seems to be the first step toward a solution to this problem.

Examples are the resource allocation and the gateway billing frameworks. The resource allocation framework is based on a layered architecture, the typical three-tier structure, whereas the gateway billing framework uses a pipe-filter architecture as its primary decomposition structure. A designer may want to compose the two frameworks to obtain a more flexible gateway billing system where billing applications can dynamically allocate filtering and forwarding resources. However, this will prove to be all but trivial due to the mismatch in the underlying architectural styles.

Overlap of framework entities

When constructing an application from reusable components, the situation may occur that two (or more) frameworks are used that, combined, cover the application requirements. In this case, a real-world entity may be represented by both frameworks-for instance, as a class-but modeled from their respective perspectives. Consequently, the representations by the frameworks may have modeled both overlapping and exclusive properties. In the application, however, the real-world entity should be modeled by a single object and the two representations should be integrated into one.

The situation is thus that two frameworks F_1 and F_2 exist that both contain representations of a real-world entity ^{r} in the form of a class C^r_{F1} and C^r_{F2}. The real-world entity has a virtually infinite set of properties P_r, of which a subset is represented by each of the framework classes, that is, $P_{C^r_{F1}} \subset P_r$ and $P_{C^r_{F2}} \subset P_r$, where $P_{C^r_{F1}} \neq P_{C^r_{F2}}$. Integrating these representations can, from a naive perspective, be done using multiple inheritance, that is, the application class C^r_A inherits from both classes C^r_{F1} and C^r_{F2}, thereby combining the properties from these classes, that is, $P_{C^r_A} = P_{C^r_{F1}} \cup P_{C^r_{F2}}$.

In practice, however, the composition of properties from the frameworks classes is not as straightforward as presented here, because the properties are not mutually independent. One can identify at least three situations where more advanced composition efforts are required.

Both framework classes represent a state property of the real-world entity but represent it in different ways. For instance, most sensors in the fire alarm framework contain a Boolean state for their value, whereas measurement systems sensors use more complex domains, for example, temperature or pressure. When these sensors are composed into an integrated sensor, this requires every state update in one class to be extended with the conversion and update code for the state in the other class.

Both framework classes represent a property of the real-world entity, but one class represents it as a state and the other class as a method. The method indirectly determines what the value of the particular property is. For instance, an actuator is available both in the fire alarm and the measurement system frameworks. In an application, the software engineer may want to compose both actuator representations into a single entity. One actuator class may store as a state whether it is currently active or not, whereas the other class may indirectly deduce this from its other state variables. In the application representation, the property representation has to be solved in such a way that reused behavior from both classes can deal with it.

The execution of an operation in one framework class requires state changes in the other framework class. Using the example of the actuator mentioned earlier, an activate message to one actuator implementation may require that the active state of the other actuator class be updated accordingly. When the software engineer combines the actuator classes from the two frameworks, this aspect of the composition has to be explicitly implemented in the glue code.

Composition of entity functionality

A typical example of this problem can be found in the three-tier application architecture. A software engineer constructing an application in this domain may want to compose the application from a user interface framework, a framework covering the application domain concepts, and a framework providing database functionality. A problem analogous to the one discussed in the previous section now appears. The real-world entity is now represented in the application domain framework. However, aspects of the entity have to be presented in some user interface and the entity has to be made persistent and suited for transactions. Constructing an application class by

composing instances from the three frameworks or by multiply inheriting from classes in the three frameworks will not produce the desired result. State changes caused by messages to the application domain part of the resulting object will not automatically affect user interface and database functionality of the object.

In a way, the behavior from the user interface and database framework classes needs to be superimposed on the object [6]. However, since this type of composition is not available in object-oriented languages, the software engineer is required to extend the application domain class with behavior for notifying the user interface and database classes, for example, using the Observer design pattern [15]. One could argue that the application domain class should have been extended with such behavior during design, but, as mentioned earlier, most frameworks are not designed to be composed with other frameworks but to be extended with application-specific code written specifically for the application at hand.

This problem occurred in the fire alarm framework, where several entities had to be persistent and were stored in nonvolatile memory - an EEPROM. To deal with this, each entity was implemented by two objects: one application object and one persistence object. These two objects were obviously tightly coupled and had frequent interactions, because they both represented parts of one entity.

Hollywood principle

In [49], the distinction is made between calling and called frameworks. Calling frameworks are the active entities in an application that call the other parts, whereas called frameworks are passive entities that can be called by other parts of the application. One of the problems when composing two calling frameworks is that both expect to be the controlling entity in the application and in control of the main event loop.

For example, both the fire alarm and the measurement system framework are calling frameworks that contain a control loop triggering the iterative framework behavior. If we would compose both frameworks in an application, the two control loops would conflict, leading to incorrect behavior. A calling framework is based on assumptions of execution, which may lead to problems of extensibility and composability. Adding calls in the control loop might be

impossible since the inverse calling principle has led designers to assume total control of the execution flow, leaving no means to modify it.

One may argue that a solution could be to give each framework its own thread of control, leading to two or more independently executing control loops. Although this might work in some situations, there are at least two important drawbacks. The first is that all application objects that can be accessed by both frameworks need to be extended with synchronization code. Since a class often cannot be modularly extended with synchronization code, all reused classes must be edited to add this. A second drawback is that often one framework needs to be informed about an event that occurred in the other framework because the event has application-wide relevance. This requires that the control loops of the frameworks become much more integrated than two concurrent threads.

Integrating legacy components

A framework presents, among other things, a design for an application in a particular domain. Based on this design, the software engineer may construct a concrete application using framework classes, either directly or indirectly, by inheriting from them. When the framework class contains behavior only for internal framework functionality and not much of the behavior required for the application at hand, the software engineer may want to include existing (legacy) classes in the application that need to be integrated with the framework. It is, however, far from trivial to integrate a legacy class in the application, since the framework depends on the subclassing mechanism. Since the legacy component will not be a subclass of the framework class, typing conflicts result.

To integrate a legacy component, the software engineer is required to create some form of adaptation or bridging. For instance, a class could be defined, inheriting from both the framework class and the legacy class, forwarding all calls matching the framework class interface to corresponding methods in the legacy component. This problem is studied in more detail in [32] and, as a solution, they propose the use of templates. Their solution, however, requires that the framework be designed using their approach, which is unlikely in the general case.

4.4 Framework evolution and maintenance

Development of a framework must be seen as a long-term investment and, as such, it must be treated as a product that needs to be maintained. Early in the framework development life cycle, several design iterations are often necessary. An important reason for the iteration is, according to [24], that a framework is supposed to be reusable and the only way to prove this is to reuse the framework and identify the shortcomings. In [39], a set of behavior-preserving transformations, or refactorings, has been identified that characterizes the code and low-level design changes that may occur in the iterations. Changes or refactorings related to inheritance hierarchies [40] and component hierarchies [23] are especially of relevance for framework iteration just before releasing the first version of the framework. This iteration between the phases is very frequent and it involves a considerable amount of simple but tedious work that would benefit from tool support.

The correction of an error in the framework is not as simple as one may think. The error should, obviously, be corrected for the current application developed with the framework, but how should the error be handled in the existing applications developed with the framework? The framework development organization has to decide either to split the framework into two separate frameworks that will need to be maintained or to implement a work-around for the current application and live with the maintenance problems for the application.

As described earlier, a framework aims at representing a domain-specific architecture, but, in many cases, it is difficult to know the exact domain boundary that has to be captured. One problem is that business changes in the organization supporting the framework may require that the domain be adapted accordingly. Presently, it is hard to predict how these kinds of business domain changes affect the existing framework.

In the following sections, we discuss the problems of framework change, selection of maintenance strategies, business domain changes, and design iterations.

Framework change

Imagine the situation in which a fault is found in an application using a framework. Where should the error be corrected? It could be either in the framework or in the application-specific code. If the error is located in the application-specific code, it will be relatively simple to correct. However, if the error is located in the framework, it may be impossible for the framework user to fix the error because of the complexity of the framework (in other words, it is difficult to understand how the framework works) or lack of access to the source code of the framework. Often, the organization has full control of the framework, including the source code, has the ability to correct the error, and does so accordingly. The application will, after the error correction, work as intended, but all existing applications using the framework have not been taken care of in this way. One may wonder whether all applications should be upgraded with the corrected version of the framework, since all are functioning correctly and the error has not (yet) caused any problems. In addition, if one decides to correct the error in the previous applications, there is a potential risk of introducing new faults in these applications. Finally, the cost of upgrading all existing applications is high.

On the first occasion that a real application based on the framework is developed, a number of problems will be found that require changes of the framework. These problems are most easily solved by direct support to the application developers from the framework team [49]. The framework developers then collect all the experiences from the first application and iterate the framework design once more to mature the framework.

However, the identification of errors will not stop after the first application development, and a problem is how to deal with subsequent application developments. It is infeasible to support the application team and redesign the framework over and over again, since the intended reuse benefits will then not be achieved. If problems are encountered when reusing the framework, they are the result of one of two causes. First, the domain covered by the framework is incomplete; in other words, we have failed to capture the domain in our domain analysis. Second, the application does not match the intended framework domain; in other words, we have decided to develop an application whose requirements constitute a bad fit with the domain

covered by the framework. In both cases, the framework will generally be changed, forcing the organization to face the problem of selecting maintenance strategies. This is further described in the following section.

Choice of maintenance strategy

Given the situation that the framework has changed, either because the domain covered by the framework was incomplete or the current application to be developed is a bad fit with the framework domain, it is necessary to decide whether to redesign the framework or do a work-around for this specific application to overcome the problems.

In the case where we decide to redesign the framework, the current application under development will need to be delayed until a new version of the framework is available. In addition, there will be additional cost for the redesign of the framework that has to be accounted for through more extensive reuse of the framework. A third consequence is that the organization is forced to maintain two versions of the framework: the original, since there may exist applications based on this version, and the redesign, for future applications to be developed. The length of the period during which two maintenance lines need to be supported depends on the expected lifetime of the developed applications and the expected number of applications developed for the original framework version.

In the case of a work-around in the application, there will be no additional maintenance for the framework since there is only a single framework. The maintenance problem will instead occur for the developed application. This maintenance strategy will not be suitable if the application under development will have a long expected lifetime. However, this may be an acceptable situation if it is expected that no similar applications will be developed in the foreseeable future.

In conclusion, the problem of deciding the maintenance strategy is dependent on factors that include time pressure, estimated lifetime for the software involved, and existing and expected future applications. Unfortunately, no clear guidelines exist that support the decision.

Business domain change

The framework is developed in a domain that is closely related to the organization's business domain. Unfortunately, the business domain is often weakly defined and not stable over time. Especially when the organization's business domain changes frequently, the framework will be more difficult to reuse and, if not maintained, will be completely useless after a rather short time span. Thus, since the business domain changes, the domain captured by the framework has to be adapted to follow this change, and this affects the existing framework. The probability of business domain change is an important risk factor that must be considered in the investment of the framework development effort.

There are, in principle, three approaches to attacking the problem of business domain change. The developer may define the original framework domain much wider than is currently relevant, assuming that it will capture most future new domain changes. As discussed earlier, there are obvious problems in developing a framework for a large domain. For example, it often is unclear what functionality should be included in the domain to incorporate an existing adjacent business domain or to incorporate a future business domain. Other problems are obtaining funding for a larger framework effort, finding domain expertise for the domain, and verifying framework functionality.

Another approach is to handle the business domain change problem by redesigning the framework such that it covers both the original domain and the new domain. The problem with this solution is that the organization has to support two frameworks with accompanying additional support, maintenance, and costs. Otherwise, the revised framework must be used for new versions of earlier applications, potentially causing major and expensive updates.

A third approach is to reuse ideas from the original framework and develop a framework for the new business domain. One problem with this approach is that the return on investment of the existing framework effort will be less than expected, since most new applications will most likely be in the new business domain. Another problem is that, again, a new framework has to be developed with all its possibilities and, especially, its problems. The main advantage this

time around is that developers have (hopefully) learned many lessons from the first framework development.

Design iterations

It is generally accepted that framework development is an iterative process. Traditional software development may also require this, but iterations are more important and explicit when designing frameworks. The underlying reason, we believe, is that frameworks primarily deal with abstractions, and abstractions are very difficult to evaluate. Therefore, the abstractions need to be made concrete in the form of test applications before they can be evaluated.

There are, in principle, two important problems with the iterations. First, it is expensive and difficult to predict the amount of resources needed for a framework project. The second problem, closely related to the first problem, is that it is difficult to stop iterating and decide that the framework is ready for release. An analogy is possible between these problems and the problems related to software testing. However, some important differences exist, including the fact that design iterations in frameworks are much more expensive than traditional testing. For example, a test case corresponds to an instantiation of a test application. The cost of correcting an error in the traditional system testing depends on where the error was introduced-a coding error could easily be corrected, but design errors or requirements errors are much more expensive. In the framework case, on the other hand, detected errors-that is, situations where the framework was not suitable for a specific test application-may require a redesign of the framework. Not only is such a redesign expensive, but it may also invalidate existing test applications that then need to be modified accordingly.

We have identified three issues that are open for improvement: first, quantitative and qualitative guidelines for when to stop framework design; second, methods that can predict the number of iterations and the corresponding effort involved; and, finally, the need to investigate whether it is possible to evaluate a framework for a specific application without implementing a complete test application.

5. Summary

Object-oriented frameworks provide an important step forward in the development of large reusable components, when compared to the traditional approaches. In our work with the frameworks described in this chapter, we have experienced the advantages of framework-based development as compared to the traditional approach of starting from scratch. In addition to our own positive experiences, others, such as [38], have identified that the use of object-oriented frameworks reduces the amount of effort required for application development and can be a productive investment.

However, as we report in this chapter, a number of problems and hindrances that complicate the development and use of object-oriented frameworks still exist. These problems can be divided into four categories:

Framework development. Problems in the development of a framework are related to the domain scope, framework documentation, business investment models for framework domains, and framework development methods, as well as framework testing and releasing.

Framework usage. The user of a framework has problems related to managing the framework development process, deciding whether a framework is applicable, estimating the development time and size of application-specific code, understanding the framework, verifying the application-specific code, and debugging.

Framework composition. In case the application requires multiple frameworks to be composed, the software engineer may experience problems with respect to mismatches in the architectural styles underlying the frameworks, the overlap of framework entities, the composition of entity functionality, possible collisions between the control flows in calling frameworks, and the integration of legacy components.

Framework evolution and maintenance. Being a long-lived entity, frameworks evolve over time and need to be maintained. The framework development team may experience problems with handling changes to the framework, choosing the maintenance strategy, changes of the business domain, and handling design iterations.

The experiences and problems reported in this chapter are relevant for both software engineers and researchers on object-oriented reuse.

Based on this chapter, practitioners should be able to avoid the problems that some of their predecessors have experienced. In addition, this chapter could be used as an agenda for researchers on object-oriented software development or on software reuse in general.

References

[1] Andert, G. Object frameworks in the Taligent OS. Proceedings of Compcon 1994. Los Alamitos, CA: IEEE CS Press, 1994.

[2] Apple Computer Inc. MacAppII Programmer's Guide. 1989.

[3] Beck, K., and R. Johnson. Patterns generate architectures. Proceedings of the 8th European Conference on Object-Oriented Programming, Bologna, Italy, July,1994.

[4] Betlem, B.H.L., R.M. van Aggele, J. Bosch, and J.E. Rijnsdorp. An Object-Oriented Framework for Process Operation. Technical report, Department of Chemical Technology, University of Twente, 1995.

[5] Booch, G. Object Oriented Analysis and Design with Applications, 2nd ed. Redwood City, CA: Benjamin/Cummings, 1994.

[6] Bosch, J. Composition through superimposition. In Object-Oriented Technology-ECOOP1997 Workshop Reader, J. Bosch, S. Mitchell, eds., LNCS 1357. Springer-Verlag, 1997.

[7] Bosch, J. Measurement systems framework. In Domain-Specific Application Frameworks, M. Fayad andR. Johnson, editors. New York: John Wiley & Sons, 1999.

[8] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stahl. Pattern-Oriented Software Architecture: A System of Patterns. New York: John Wiley & Sons, 1996.

[9] Cotter, S., and M. Potel. Inside Taligent Technology. Reading, MA: Addison-Wesley, 1995.

[10] Dagermo, P., and J. Knuttson. Development of an Object-Oriented Framework for Vessel Control Systems. Technical Report ESPRIT III/ESSI/ DOVER Project #10496, 1996.

[11] Deutsch, L.P. Design reuse and frameworks in the Smalltalk-80 system. In Software Reusability, Volume II, T.J. Biggerstaff and A.J. Perlis, editors. Reading, MA: ACM Press/Addison-Wesley, 1989.

[12] Fayad, M.E., and D. Hamu. Object-oriented enterprise frameworks. Submitted for publication to IEEE Computer, 1999.

[13] Fayad, M.E., and M. Laitinen. Transition to Object-Oriented Software Development. New York: John Wiley & Sons, 1998.

[14] Fayad, M.E., and D. Schmidt. Object-oriented application frameworks. Communications of the ACM, October 1997:32-38

[15] Gamma, E., R. Helm, R. Johnson, and J.O. Vlissides. Design Patterns: Elements of Reusable OO Software, Reading, MA: Addison-Wesley, 1995.

[16] Garlan, D., R. Allen, and J. Ockerbloom. Architectural mismatch or why it's so hard to build systems out of existing parts. Proceedings of the 17th International Conference on Software Engineering, Seattle, WA, April 1995.

[17] Goldberg, A., and D. Robson. Smalltalk-80: The Language. Reading, MA: Addison-Wesley, 1989.

[18] Helm, R., I.M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. Proceedings of ECOOP/ OOPSLA 1990, Ottawa, Canada, October 1990.

[19] Huni, H., R. Johnson, and R. Engel. A framework for network protocol software. Proceedings of the 10th Conference on OOPSLA, Austin, TX, July 1995.

[20] Jacobson, I., M. Christerson, P. Jonsson, and G. Övergaard. Object-Oriented Software Engineering: A Use Case Approach. Reading, MA: Addison-Wesley, 1992.

[21] Johnson, R.E. Documenting frameworks with patterns. Proceedings of the 7th Conference on Object-Oriented Programming Systems, Languages and Applications, Vancouver, Canada, October 1992.

[22] Johnson, R., and B. Foote. Designing reusable classes. Journal of Object-Oriented Programming 1(2), June 1988.

[23] Johnson R.E., and W.F. Opdyke. Refactoring and aggregation. Proceedings of ISOTAS 1993: International Symposium on Object Technologies for Advanced Software, 1993

[24] Johnson, R.E., and V.F. Russo. Reusing Object-Oriented Design. Technical Report UIUCDCS 91-1696, University of Illinois, 1991.

[25] Karlsson, E-A., ed. Software Reuse: A Holistic Approach. New York: John Wiley & Sons, 1995.

[26] Kihl, M., and P. Ströberg. The business value of software development with object-oriented frameworks. Master's thesis, Department of Computer Science and Business Administration, University of Karlskrona/Ronneby, Sweden, May 1995 (in Swedish).

[27] Krasner, G.E., and S.T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming 1(3), August-September 1988.

[28] Lajoie, R., and R.K. Keller. Design and reuse in object-oriented frameworks: Patterns, contracts and motifs in concert. Proceedings of the 62nd Congress of the Association Canadienne Francaise pour l'Avancement des Sciences, Montreal, Canada, May 1994.

[29] Lim, Wayne. Reuse economics: A comparison of seventeen models and directions for future research., Proceedings of the International Conference on Software Reuse, Orlando, FL, April 1996.

[30] Linton, M.A., J.M. Vlissides, and P.R. Calder. Composing user interfaces with interViews. IEEE Computer 22(2), February 1989.

[31] Lundberg, L. Multiprocessor performance evaluation of billing gateway systems for telecommunication applications. Proceedings of the ICSA Conference on Parallel and Distributed Computing Systems, pp. 225-237, September 1996.

[32] Lundberg, C., and M. Mattsson. Using legacy components with object-oriented frameworks. Proceedings of Systemarkitekturer 1996, Borås, Sweden, 1996.

[33] Mattsson, M. Object-oriented frameworks--A survey of methodological issues. Licentiate thesis, LU-CS-TR: 96-167, Department of Computer Science, Lund University, 1996.

[34] Meyer, B. Applying design by contract. IEEE Computer, October 1992.

[35] Molin, Peter. Verifying Framework-Based Applications by Conformance and Composability Constraints. Research Report 18/96, University of Karlskrona/ Ronneby, 1996.

[36] Molin, Peter. Experiences from Applying the Object-Oriented Framework Technique to a Family of Embedded Systems. Research Report 19/96, University of Karlskrona/Ronneby, 1996.

[37] Molin, Peter, and Lennart Ohlsson. Points & deviations: A pattern language for fire alarm systems. Proceedings of the 3rd International Conference on Pattern Languages for Programming, Paper 6.6, Monticello IL, September 1996.

[38] Moser S., and O. Nierstrasz. The effect of object-oriented frameworks on developer productivity. IEEE Computer Theme Issue on Managing Object-Oriented Software Development, Mohamed E. Fayad and Marshall Cline, editors.,September 1996:45-51.

[39] Opdyke, W.F. Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992

[40] Opdyke, W.F., and R.E Johnson. Creating abstract superclasses by refactoring. Proceedings of CSC 1993: The ACM 1993 Computer Science Conference, Indianapolis, Indiana, February 1993.

[41] Poulin, Jeff. Measuring software reusability. Proceedings International Conference on Software Reuse, Rio de Janeiro, November 1994.

[42] Pree, W. Meta patterns--A means for capturing the essentials of reusable object-oriented design. Proceedings of the 8th European Conference on Object-Oriented Programming, Bologna, Italy, July,1994.

[43] Roberts, D., and R. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks, Proceedings of the Third Conference on Pattern Languages and Programming. Allerton Park, IL, September1996.

[44] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. Object-Oriented Modeling and Design. Englewood Cliffs, NJ: Prentice Hall, 1991.

[45] Russo, V.F. An object-oriented operating system. Ph.D. thesis, University of Illinois at Urbana-Champaign, October 1990.

[46] Schmucker, K.J. Object-Oriented Programming for the Macintosh. Hayden Book Company, 1986.

[47] Schäfer, W., R. Prieto-Diaz, and M. Matsumoto. Software Reusability. Ellis-Horwood Ltd., 1994.

[48] Shaw, M., and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Upper Saddle River, NJ: Prentice Hall, 1996.

[49] Sparks, S., K. Benner, and C. Faris. Managing object-oriented framework reuse. IEEE Computer Theme Issue on Managing Object-Oriented Software Development, Mohamed E. Fayad and Marshall Cline, editors., September 1996:53-61.

[50] Weinand, A., E. Gamma, and R. Marty. Design and implementation of ET++, a seamless object-oriented application framework. Structured Programming 10(2), July 1989.

[51] Wilson, D.A., and S.D. Wilson. Writing frameworks--Capturing your expertise about a problem domain. Tutorial notes, 8th Conference on Object-Oriented Programming Systems, Languages and Applications, Washington, DC, October 1993.

PAPER II

Composition Problems, Causes, and Solutions

II

Michael Mattsson and Jan Bosch

Chapter in "Building Application Frameworks: Object Oriented Foundations of Framework Design" Eds: M. E. Fayad, D. C. Schmidt, R. E. Johnson, Wiley & Sons,
ISBN#: 0-471-24875-4, 1999, pp. 467-487

Abstract

Whereas framework-based application development initially included a single framework, increasingly often multiple frameworks are used in application development. This chapter provides a significant collection of common framework composition problems encountered in application development together with an analytical discussion of the underlying causes for the problems. Based on the composition problems and the identified primary causes the authors describe and analyze existing solution approaches and their limitations.

The composition problems discussed are (1) composition of framework control, (2) composition with legacy components, (3) framework gap, (4) overlap of framework entities and (5) composition of entity functionality. The primary causes for these composition problems are related to (1) the cohesion between classes inside each framework, (2) the domain coverage of the frameworks, (3) the design intentions of the framework designers and (4) the potential lack of access to the source code of the frameworks. The identified problems and their causes are, up to some extent, addressed by existing solutions, such as the adapter design pattern, wrapping or mediating software, but these solutions generally have limitations and either

do not solve the problem completely or require considerable implementation efforts.

1. Introduction

Reuse of software has been one of the main goals of software engineering for decades. From the early age of computer science on, the approach of constructing a system by putting together reusable components was a clear goal of software engineers. However, the early approaches-for example, function and procedure libraries-only provided reuse of small, building-block components. With the emergence of the object-oriented paradigm, an important enabling technology for reuse of larger components became available and resulted in the definition of object-oriented frameworks. An object-oriented framework can be defined as a set of classes that embodies an abstract design for solutions to a family of related problems [8]. In other words, a framework is an abstract design and implementation for an application in a given problem domain.

The presence of a framework influences the development process for the application. In [11], we define the following phases in framework-centered software development:

The framework development phase. Normally the most effort-consuming phase, the framework development phase is aimed at producing a reusable design and related implementation in a domain. Major results of this phase are the domain analysis model and the object-oriented framework.

The framework usage phase. This phase is sometimes also referred to as the framework instantiation phase, framework customization phase, or application development phase. The result of this phase is an application developed reusing a framework. Here the software engineer has to adapt the existing framework. In the case of a whitebox framework, the software engineer makes changes and extensions to framework internals using mechanisms such as redefinition and polymorphic substitution. If the existing framework is a blackbox framework (that is, the user does not have to know about the framework internals), the customization is achieved through modification of the configuration interface using mechanisms such as

predefined options (for example, objects) and switching mechanisms. For those parts of the application design not covered by reusable classes, new classes need to be developed to fulfill the application's requirements.

The framework evolution and maintenance phase. The framework, like all software, will be subject to change. Causes for these changes can be errors reported from shipped applications, identification of new abstractions due to changes in the problem domain, and so on.

The advantage of framework-based development is the high level of reuse in the application development. Although this advantage has been claimed by proponents of the technology, recently scientific evidence supporting this claim has also started to appear, for example, [13]. Although the development of an object-oriented framework requires considerable effort, the corresponding benefits during application development generally justify the initial effort.

Traditional framework-based application development assumes that the application is based on a single framework that is extended with application-specific code. Recently, however, we have identified development in which software engineers make use of multiple frameworks that are composed to fulfill the application requirements. Experiences from the composition of two or more frameworks clearly show that frameworks generally are developed for reuse by extension with newly written application-specific code and not for composition with other software components. This focus on reuse through extension causes a number of problems when software engineers try to compose frameworks.

In this chapter, we study the problems that the software engineer may experience while composing frameworks. The identified problems are based on our experiences from developing frameworks for, among others, fire alarm systems [12], measurement systems [4], gateway billing systems [10], resource allocation and process operation [1], as well as existing literature about frameworks, for example, [3],[11],[16]. The composition problems that we identified are related to the composition of framework control, composition with legacy components, framework gap, overlap of framework entities, and composition of entity functionality. Our investigations into these problems have led us to the conclusion that the main causes for the problems are due to the cohesion between the classes in the frame-

work, the domain coverage of the framework, the design intention of the framework designer, and the potential lack of access to the source code of the framework. In addition, we discuss the available solution approaches for addressing the identified problems and their causes, and the limitations of these solutions.

We believe the contribution of this chapter to be the following. First, it provides software engineers employing object-oriented frameworks with an understanding of the problems they may experience during application development, as well as an understanding of the primary causes and solution approaches. Second, it provides topics to researchers in object-oriented software engineering that need to be addressed by future research.

The remainder of this chapter is organized as follows. In the next section, four example frameworks are introduced that are used to illustrate the problems later in the chapter. Then, the framework composition problems that we identified are described. The subsequent section discusses the primary causes underlying the identified composition problems. Next, the various solution approaches for the identified problems are discussed, as well as their limitations, and the chapter is concluded in the succeeding section.

2. Object-Oriented Framework Examples

In this section, we discuss four frameworks that, in the remainder of the chapter, are used to exemplify and illustrate the discussion. These frameworks have been selected for their illustrative ability rather than for their industrial relevance. During the discussion, we assume that it is the intention to reuse two or more frameworks in one application, requiring the frameworks to be composed. The frameworks are a measurement systems framework, a graphical user interface (GUI) framework, a fire alarm framework, and a framework for statistical analysis. The following describes the involved frameworks. However, since most readers are familiar with GUI frameworks, this framework is not described.

In cooperation with EC-Gruppen AB, we have been involved in the design of a framework for measurement systems, Figure 1. Measurement systems are systems that are located at the beginning or end of production lines to measure some selected features of production

items. The hardware of a measurement system consists of a trigger, one or more sensors, and one or more actuators. The framework for the software has software representations for these parts, the measurement item, and an abstract factory for generating new measurement item objects. A typical measurement cycle for a product starts with the trigger detecting the item and notifying the abstract factory [5]. In response, the abstract factory creates a measurement item and activates it. The measurement item contains a measurement strategy and an actuation strategy and uses them to first read the relevant data from the sensors. After that the measurement item compares this data with the ideal values and subsequently activates the actuators when necessary to remove or mark the product if it did not fulfill the requirements. Refer to [4] for a more detailed description of the framework.

The statistical analysis framework, in this context, will be used for the statistical analysis of the data gathered by the measurement system. Although measurement systems generally measure every production item, it is often beneficial to be able to collect measurement data and analyze it statistically to condense the amount of data and to increase the information value. Examples of statistical analysis on measurement item data are the correlation between measurements on items and the equipment that manufactured the items, and the distribution of measured items in various quality categories. The results of the analysis may be used online to adjust the settings for manufacturing equipment. In the remainder of this chapter, we will discuss two versions of the framework, one version providing only the core functionality for statistical analysis and a second version containing, in addition to the core functionality, functionality for the graphical representation of statistical data, for example, histograms.

TeleLarm AB, a Swedish security company, develops and markets a family of fire alarm systems ranging from small office systems to large distributed systems for multibuilding plants. An object-oriented framework was designed as a part of a major architectural redesign effort. The framework provides abstract classes for devices and communication drivers as well as application abstractions such as input points and output points representing sensors and actuators. System status is represented by a set of deviations that are available on all nodes in the distributed system. The notion of periodic objects is introduced, giving the system large-grain concurrency without the

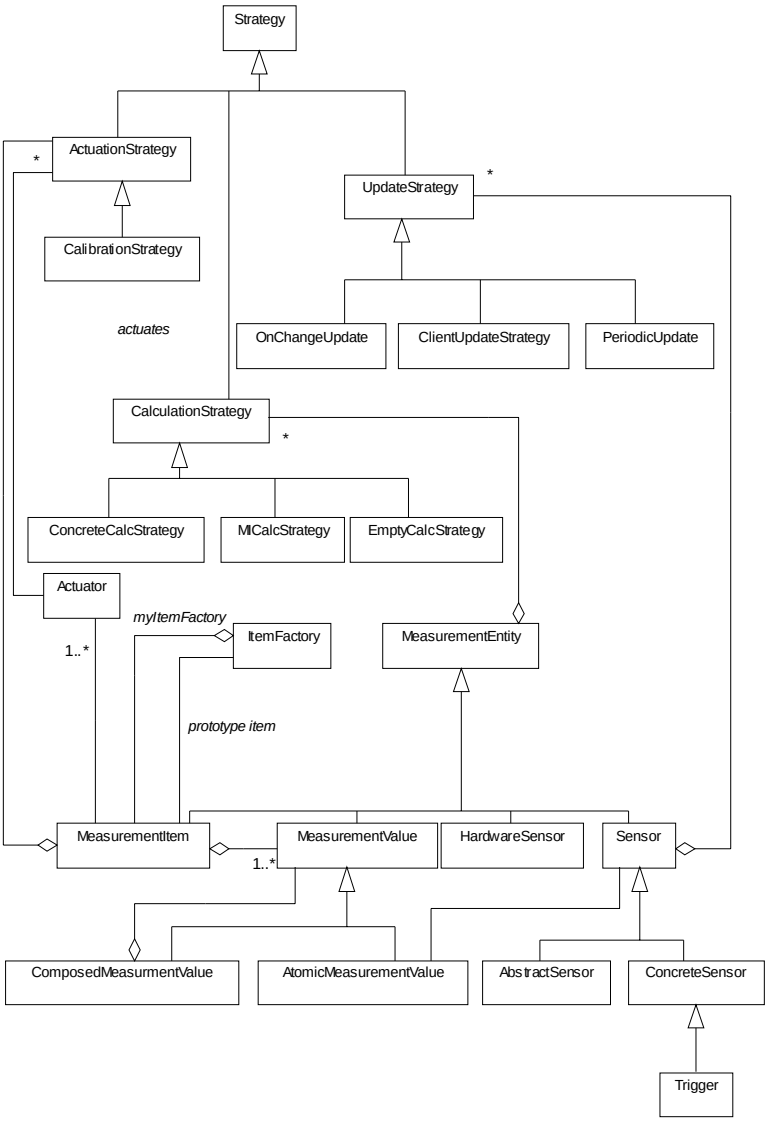

Figure 1. *Measurement system*

problems associated with asynchronous fine-grain concurrency. Two production instantiations from the framework have been released so

far. The fire alarm framework has been described as a pattern language; see [12] for more information. Figure 2 presents a schematic system view of the framework.

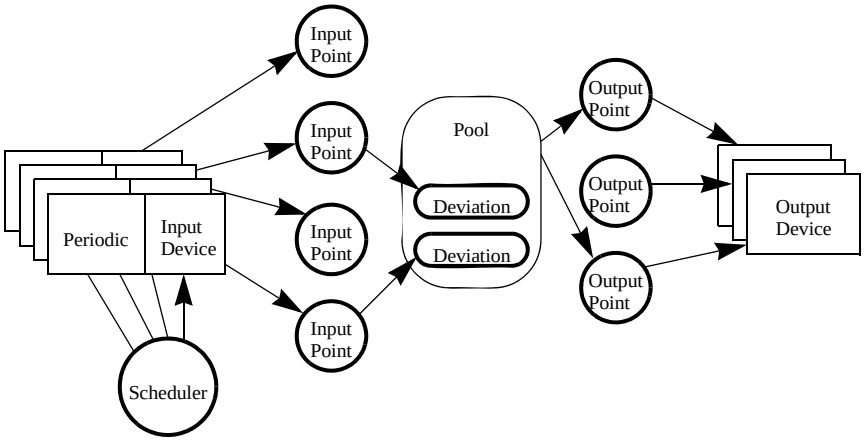

Figure 2. *System view of the fire alarm framework.*

3. Framework Composition Problems

Several of the companies that we cooperate with are moving away from the traditional single framework-based application development to application development based on multiple frameworks that need to be composed with each other, with class libraries, and with existing legacy components. In this section, five composition problems are described that may occur when composing a framework with another software artifact. Although software engineers may encounter additional problems, we believe this list to cover the primary problems.

3.1 Composition of framework control

One of the most distinguishing features of a framework is its ability to make extensive use of dynamic binding. In traditional class or procedure libraries, the application code invokes routines in the library and it is the application code that is in control. For object-oriented frameworks, the situation is inverted and it is often the framework

code that has the thread of control and calls the application code when appropriate. This inversion of control is often referred to as the *Hollywood principle*: "Don't call us-we'll call you." Figure 3 illustrates this inversion.

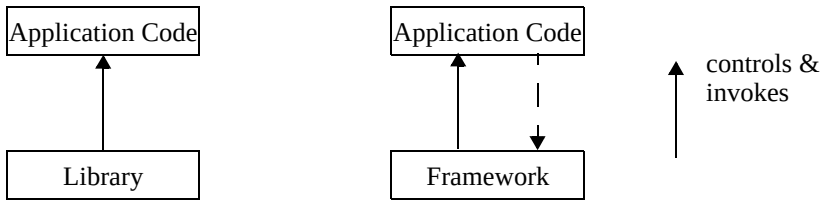

Figure 3. *The inversion of control.*

In [16], a distinction is made between *calling* and *called* frameworks. Calling frameworks are the active entities in an application, controlling and invoking the other parts, whereas called frameworks are passive entities that can be called by other parts of the application-in other words, they are more like class libraries. One of the problems when composing two calling frameworks is that both frameworks expect to be the controlling entity in the application and in control of the main event loop.

An example is the composition of the measurement system framework and the GUI framework, as shown in Figure 4. The measurement system framework has, from the moment a trigger enters the system, a well-defined control loop that has to be performed in real time and that creates a measurement item, reads sensors, computes, activates actuators, and stores the necessary historical data. The GUI framework has a similar thread of control, though not in real time, that updates the screen whenever a value in the system changes-for example, that of a sensor-or performs some action when the user invokes a command. These two control loops can easily collide with each other, potentially causing the measurement part to miss its real-time deadlines and causing the GUI to present incorrect data due to race conditions between the activities.

Solving this problem is considerably easier if the frameworks are supplied with source code that makes it possible to adapt the framework to handle this problem. However, the control loop in the framework may not be localized in a single entity. Often it is distributed

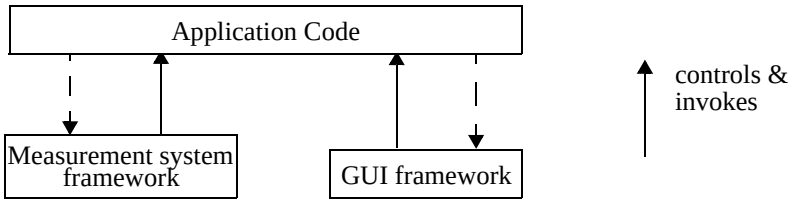

Figure 4. *The problem of framework control*

over the framework code, which causes changes to the control to affect considerable parts of the framework.

3.2 Composition with legacy components

A framework presents a design for an application in a particular domain. Based on this design, the software engineer may construct a concrete application through extension of the framework, for example, subclassing the framework classes. When the framework class only contains behavior for internal framework functionality but not the domain-specific behavior required for the application, the software engineer may want to include existing, legacy, classes in the application that need to be integrated with the framework. It is, however, far from trivial to integrate the legacy class in the application, since frameworks often rely heavily on the subclassing mechanism. Since the legacy component will not be a subclass of the framework class, developers may run into typing conflicts. We refer to [9] for a more extensive discussion.

The measurement system framework is used as an example to illustrate the problem. The framework handles the measurement cycle and provides for attaching a sensor, a measurement algorithm, and an actuator by specifying the interface classes Sensor, Calcula-

tionStrategy, and Actuator. The main classes in the framework are shown in Figure 5

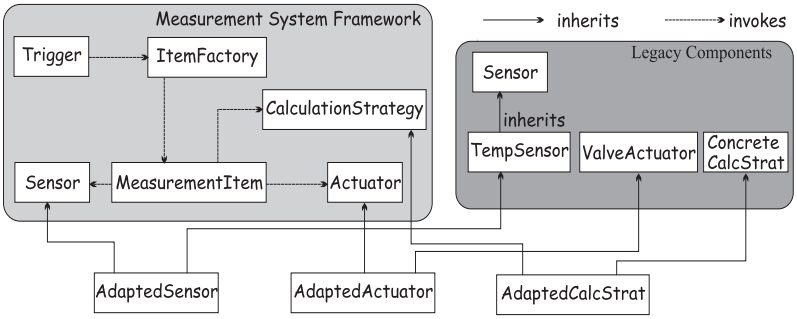

Figure 5. *The measurement system framework and associated legacy classes.*

Assume that we have a library of legacy components with different types of sensors, actuators, and strategies for measurement and calculation, and that we have found suitable classes that we want to use in the three places in the framework's interface. Our library sensor class is called TempSensor, and it happens to be a subclass of the class Sensor, which is an abstract class, defined in our library. However, since the framework also defines a class Sensor, the usage of the library class will lead to conflicts. Even if the class name for the library superclass matches with that of the framework interface class, the class TempSensor cannot be used, because these names do not designate the same class. An alternative approach, which is used in Figure 5, is to make use of the Adapter design pattern [5] for class adaptation. This approach normally solves the problem, and identifies name clashes between class names in the framework and the legacy components. However, as we identified in [2], the Adapter design pattern has some disadvantages associated with it. One disadvantage is that for every element of the interface that needs to be adapted, the software engineer has to define a method that forwards the call to the corresponding method in the legacy class. Moreover, in the case of object adaptation, those requests that otherwise would not have required adaptation have to be forwarded as well, due to the intermediate adapter object. This leads to considerable implementation overhead for the software engineer. In addition, the pattern suffers from the self problem and lacks expressiveness. Finally, since

behavior of the Adapter pattern is mixed with the domain-related behavior of the class, traceability is reduced.

The problems related to integration of legacy components in framework-based applications are studied in more detail in [9].

3.3 Framework gap

Often, when thinking about composition problems of components, the first thing that comes to mind is different kinds of overlap between the components, but there may also exist problems due to nonoverlap between the components. This occurs when two frameworks are composed and the resulting structure still does not cover the application's requirements. This problem is generally referred to as *framework gap* (see, for example, [16]).

If the framework is a called framework, the framework gap problem may be solved with an additional framework interface, including both the existing and the additionally required functionality. Figure 6 illustrates such a *wrapping* approach.

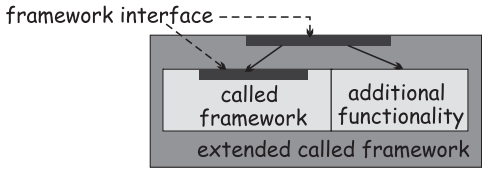

Figure 6. *Called framework extended to fill the framework gap.*

In the case where a calling framework lacks functionality, mediating software is needed to alleviate the problem. Consider two calling frameworks, A and B, and that we have a framework gap. The mediating software must provide functionality for informing framework A about the actions that had happened in framework B. This information must be presented for framework A in terms understandable to A. The mediating software may also need to cut out parts of the functionality offered by the frameworks. The functionality has to be replaced by the mediating software together with application-specific code that composes the functionality from frameworks A and B and the functionality required for the framework gap. The situation will now be that the two frameworks have to call the mediating software, which then calls the application-specific functionality. An additional

problem is that the mediating software becomes dependent on the current framework versions, which may lead to rather complex maintenance problems for the application when new versions of the framework must be used.

3.4 Composition overlap of framework entities

When developing an application based on reusable components, it may occur that two (or more) frameworks both contain a representation-that is, a class-of the same real-world entity, but modeled from their respective perspectives. When composing the two frameworks, the two different representations should be integrated or composed since they both represent the same real-world entity. If the represented properties are mutually exclusive and do not influence each other, then the integration can be solved by multiple inheritance. However, very often these preconditions do not hold and there exist shared or depending properties between the representations, causing the composition of the representations to be more complex. We can identify at least three cases where alternative composition techniques are necessary.

First, consider the situation where both framework classes represent a state property of the real-world entity, but the entity is represented in different ways. For example, a sensor in the fire alarm framework has a simple presentation which a Boolean state for its value while the same sensor in the measurement framework has a more complex representations such as temperature and pressure. When these two representations of a sensor are composed into an integrated sensor, this requires every state update in one framework sensor class to be extended with the conversion and update code for the state in the other framework sensor class.

In the second case, assume that both framework classes represent a property p of the real-world entity, but one framework class represents it as a state and the other framework class as a method. The value of the particular property p is indirectly computed by the method. For instance, an actuator is available both in the fire alarm and the measurement system frameworks. In an application, the software engineer may want to compose both actuator representations into a single entity. One actuator class may store as a state whether it is currently active or not, whereas the other class may indirectly

deduce this from its other state variables. In the application representation, the property representation has to be solved in such a way that reused behavior from both classes can deal with it.

In the third case, the execution of an operation in one framework class requires state changes in the other framework class. Using the example of the actuator mentioned earlier, an activate message to one actuator implementation may require that the active state of the other actuator class needs to be updated accordingly. When the software engineer combines the actuator classes from the two frameworks, this aspect of the composition has to be explicitly implemented in the glue code.

3.5 Composition of entity functionality

Sometimes a real-world entity's functionality has to be modeled through composition with parts of functionality from different frameworks. Consider the case of a typical software structure with three layers, each represented by a framework: at the top is a user interface layer, in the middle an application domain-specific layer, and at the bottom a persistence layer. Our real-world entity is now represented in the application domain-specific framework, but some aspects of the entity have to be presented in the user interface layer, and the entity also has to be made persistent for some kinds of transactions, and so on. Just composing the respective classes from the three frameworks or using multiple inheritance will not result in the desired behavior. For example, changes of the state caused by messages to the application domain-specific part of the resulting object will not automatically affect the user interface and persistence functionality of the objects.

Thus, the software engineer is required to extend the application domain class with behavior for notifying the user interface and database classes, for example, using the Observer design pattern [5]. One could argue that the application domain class should have been extended with such behavior during design, but, as mentioned earlier, most frameworks are not designed to be composed with other frameworks but to be extended with application-specific code written specifically for the application at hand.

This problem occurred in the fire alarm framework, where several entities had to be persistent and were stored in nonvolatile memory,

that is, an EEPROM. To deal with this, two objects-one application object and one persistence object-implemented each entity. These two objects were obviously tightly coupled and had frequent interactions, due to the fact that they both represented parts of one real-world entity.

4. Underlying Causes

The framework composition problems identified in the previous section can be related to a number of causes that underlie these problems. In this section, we discuss what we believe to be the four primary causes of these problems: framework cohesion, domain coverage of the framework, the design intention of the framework designer, and lack of access to the source code. In the following, each cause will be described.

4.1 Framework cohesion

The main issue in the framework usage phase is to develop an application through reusing an existing framework. The existing framework has to be extended or composed with some other software artifact to realize the requirements of the application under development. Three different software artifacts that can be used for the adaptation and composition are new application-specific source code, class libraries, and other frameworks.

Virtually every application that is developed based on a framework will require application-specific code to be developed and, since this code is designed to be composed with the framework, composition problems rarely occur. However, as described earlier, if a class library or a framework has to be composed with our framework, several composition problems may occur. But why is it so hard to compose these software artifacts with our framework? One of the answers to this question is *framework cohesion*. The functionality of a class in the framework can be divided into various types. The two types relevant for the discussion here are the domain-specific behavior corresponding to the real-world entity that the class represents and the interaction behavior to communicate to other framework entities for mutual updating. The latter type of functionality we refer

to as *cohesive behavior*: It establishes the cohesion between the entities in the framework. So, for a class from a class library or another framework to replace a class in the framework, this class not only has to represent the appropriate domain behavior but also correct cohesive behavior. Since the cohesive behavior is very specific for the framework, it is rather unlikely that a separately developed class will fulfill these requirements.

4.2 Domain coverage

An object-oriented framework provides an abstract design for an application in some problem domain. However, the framework need not cover the complete domain but may contain classes for only a subset of the relevant entities in the domain. To determine whether a framework covers the complete domain is a rather subjective activity since application domains are not defined very precisely. For instance, we considered the measurement system framework that we developed to cover a complete application domain. During discussions with other software engineers, we found that they considered the framework to provide partial domain coverage since it provided only discrete measurement of items and did not incorporate continuous measurement required for, among other things, production of fluid products.

When composing a framework with another framework, a developer may experience no domain overlap, little overlap, or considerable overlap. No overlap between the frameworks will obviously not cause problems with respect to composition of overlapping entities, but it may be the source of a framework gap problem. Little domain overlap is often relatively easy to deal with since it only requires the adaptation of a few classes in both frameworks. Considerable overlap, however, may cause the reuse of one of the frameworks to be more effort consuming than writing the code from scratch. This problem is especially problematic for long-lived applications, since the frameworks on which an application is based often evolve over time, just like the application itself. Considerable redesign of a framework is then required to be repeated for every consecutive version of the application, if the application is to benefit from the improvements of later framework versions.

The solution for this problem is primarily a matter of standards. If an accepted definition of application domains and the boundaries between the domains would exist, framework designers could take these definitions into account, and composition of frameworks would probably require less effort since most gaps and overlaps could be avoided. However, defining a generally accepted classification of domains is far from trivial. For instance, the statistical analysis framework described in Section 2. provides user interface functionality specifically for statistical representations, such as histograms and normal distributions. A general-purpose GUI framework will most likely not have support for these representations, but the statistical graphical representations may be very hard to integrate with the GUI framework. When defining standard application boundaries, the developer has to decide where such boundary functionality should be assigned, which is a nontrivial task.

4.3 Design intention

Several authors have identified that a reusable software artifact has to be designed for reuse by adaptation and composition. Since class libraries generally have relatively low cohesion between the classes, it is often feasible to compose class library elements with existing software. Object-oriented frameworks, on the other hand, are generally designed for reuse through extension of the framework or through composition of framework components. Due to the high cohesion inside the framework, the composition with other frameworks or other existing software is generally hard.

We can identify two variants of framework composition: horizontal and vertical. Horizontal integration is the situation where the two frameworks that are to be composed can be found on the same layer in a software system. Vertical integration indicates the situation where one framework depends on the services of another framework, that is, a so-called layered system. For example, the measurement system framework and the statistical analysis framework are composed horizontally, whereas these two frameworks are composed vertically with the user interface framework. Independent of whether vertical and horizontal composition is used, an additional issue is whether there is one-way or two-way communication between the

composed frameworks. Two-way communication often complicates the composition of the frameworks considerably.

In conclusion, the design intentions for the framework should be defined explicitly to make it easier for the software engineer to decide on the composability of a framework for the application at hand. It should be made clear whether the framework can be reused by extension only or that provisions for composition are available. In addition, it should be made clear whether the framework is intended for two-way communication or one-way communication. Since the design intention for most existing frameworks, generally implicitly, is for reuse by extension and one-way communication from the framework to the newly written application-specific code, this is a cause for the framework composition problems that we have identified.

4.4 Access to source code

Another, seemingly trivial, cause for framework composition problems is the lack of access to the source code of the framework. This lack of access can be a practical limitation where the framework has been delivered to the reusing software engineer in the form of a library of compiled code in combination with a set of header files. It may also be a conceptual lack of source code access due to the complexity of the code of the framework. The internal workings of the framework often are so complex that it will require considerable effort from the software engineer before he or she is able to make the changes necessary for composition purposes. Since research in software reuse has identified that the perceived effort required for reusing a component should be less than the perceived effort of developing the component from scratch, the conceptual lack of access to the source code may also constitute an important obstacle for framework reuse.

Access to the source code is important since framework composition, as identified earlier in this chapter, may require editing of the framework code to add behavior necessary for other frameworks. If no access to the source code is available, the only way to achieve the additional behavior required from the framework is through wrappers encapsulating the framework. However, as identified by [7], wrapper solutions may cause problems such as significant amounts of addi-

tional code and serious performance problems. In addition, wrappers are unable to extend behavior in response to intraframework communication. For example, one framework object invoking another object in the framework may require an object in another part of the application to be notified. However, since the wrapper is unable to intercept this communication, it is not possible to achieve this.

5. From Problems to Causes to Solutions

In the previous sections, several framework composition problems and their primary causes have been identified. In this section, we start by relating the causes to the identified problems. Subsequently, the existing solution approaches for each problem are discussed and their limitations are identified.

It is important to note that some of the solutions will cause changes to the framework and thus result in a new version of the framework. This is not a problem for the actual application under development but may cause problems for previously developed applications using the framework. This problem of maintaining previously developed applications is often referred to as *backward compatibility.* The issue of backward compatibility will arise in other situations as well, for example, fault recognition in a previously developed application or incomplete domain understanding. A more extensive discussion of backward compatibility for frameworks and the selection of an appropriate maintenance strategy can be found in [3].

Table 1 relates the framework composition problems to the identified causes. Each cell describes whether the cause is the primary source of the problem or complicates the solution of the problem, that is, whether it is a secondary cause. In the section following, these

relations are explained in more detail and the available solution approaches are described.

Table 1. Framework composition problems and their causes

	Composition of Framework Control	Composition with Legacy Component	Framework Gap	Overlap of Entities	Composition of Entity Functionality
Cohesive Behavior	Complicating factor	Primary cause	---	Complicating factor	Complicating factor
Domain Coverage	---	---	Primary cause	Primary cause	---
Design Intention	Primary cause	Complicating factor	---	Complicating factor	Primary cause
No Source Code Access	Complicating factor	Complicating factor	Complicating factor	Complicating factor	Complicating factor

5.1 Composition of framework control

The context for this problem is a situation where two frameworks are to be composed such that both assume ownership of the main event loop of the application. When composing two such frameworks, the threads of control of both frameworks have to be composed in some way. The main cause for this problem is that both frameworks are intentionally designed for adaptation and not for composition, which makes neither of the frameworks able to give up the control. Since the control loop often is deeply embedded in the source code of the framework, lack of access to the source code of the frameworks considerably complicates resolving the problem. Some changes to the control loop may even be impossible to achieve without changing the actual source code. For instance, events internal to the framework that are required externally in the application at hand, such as the notification of another framework, cannot be intercepted by framework wrappers or adapters but require actually changing the framework code. An additional factor complicating the composition of calling frameworks is the *cohesive behavior* inside the framework. The cohesive behavior, since it indicates the behavior of framework

classes for updating the other framework classes, mixes the event behavior with the domain behavior of the classes. Since the event loop becomes implicit through the cohesive behavior, making changes to the event loop is more difficult due to the-potentially many-locations where code changes are required.

Integrating the framework control from the composed frameworks requires changes to the event loops in the involved frameworks. This can be achieved by three solutions, as discussed in the following points.

Concurrency. One possible solution is to give each framework its own thread of control rather than merging the control loops. This solution approach can be used only when there is no need for the frameworks or the application code to be notified about events from the other frameworks. A disadvantage of this approach is that the application-specific objects that potentially are accessed by more than one framework need to be extended with synchronization code. An advantage is that no access to the source code is required.

Wrapping. A second solution is to encapsulate each framework by a wrapper intercepting all messages sent to and by the framework. The wrapper, upon receipt of relevant messages, can notify interested entities in the framework. This allows some level of integration of event handling in the framework-based application, but only based on events external to the framework. Internal framework events are not intercepted and can consequently not be dealt with.

Remove and rewrite. If external notification of internal framework events is necessary in the application, a solution is to remove the relevant parts of the control loops from the frameworks and to write an application-specific control loop that satisfies the needs from the involved frameworks and the application. This requires access to the source code. This may be difficult since the control loop often is not centralized to one framework entity but spread out in the framework; in other words, the cohesive behavior of the framework complicates the solution. This solution generally is a more effort-consuming approach than the previously indicated approaches.

5.2 Composition with legacy components

The software engineer may want to use legacy components in combination with the framework when a framework class does not contain

the correct application-specific domain behavior for the actual application. In addition, a legacy library of classes with the required application-specific behavior is available that fulfills the application's requirements. The integration between a framework and legacy components is not as easy as one might expect due to the cohesive behavior of the framework, which makes it difficult to replace a framework class with a legacy class, and due to potential typing conflicts in the programming language. A complicating factor in solving the composition problem is that the framework is designed for adaptation and extension and not for composition. One approach is to design the framework for composition by means of the role concept. Obviously, access to the source code of the framework simplifies the composition, since one can change the framework to refer to the legacy class rather than a framework class. Some solutions to the problem of composing legacy components with a framework are discussed in the following sections.

Adapter design pattern. Composing a legacy component with a framework is almost a schoolbook example of a situation for which the Adapter design pattern has been designed. Provided that no name clashes occur, this solution is very suitable, but it results in some implementation overhead, as for each operation in the class interface that has to be adapted there must be methods that forward the request to the equivalent method in the legacy class. Additional drawbacks are that the Adapter pattern suffers from the self problem and traceability is reduced since pattern-specific behavior is mixed with domain-specific behavior for the class [2].

Change framework. Provided the software engineer has access to the framework's source code, one possible solution is to change those parts of the framework that refer to the framework class that should be replaced by the legacy class.

Roles. As described in [9], a solution dealing with the problem once and for all is to represent the framework's reuse interface as a set of roles. A framework role is a description of the required behavior of the object that is taking the place in the reuse interface. By specifying the reuse interface as roles instead of as superclasses, we focus on the required services instead of the entire object. Using this solution, it is possible to fill the roles either with classes related to the framework or existing legacy classes, provided that the class fulfills

the requirements. This, however, requires the framework to be designed based on this approach.

5.3 Framework gap

The framework gap problem occurs when two (or more) frameworks have to be composed to satisfy the application's requirements, but the composition does not cover the requirements completely. Thus, we have a gap of missing functionality. The main cause for this problem is that the frameworks do not cover their respective domains sufficiently. As described earlier in the chapter, the definition of an application domain is still rather ad hoc, and this may lead to domain overlap or domain gaps when composing frameworks. Also for this problem, the lack of access to source code may complicate the solution to a framework considerably. In either case, one can identify a number of solution approaches for closing the framework gap.

Wrapping. As described earlier in this chapter (see, for example, Figure 6), one can extend the framework interface of a called framework with the missing functionality through an additional Application Program Interface (API). The wrapping entity, in this case, aggregates the original framework and the extension for closing the gap, but provides a uniform interface so that clients are unaware of the internal structure.

Mediating software. For a calling framework, the software engineer has to develop some kind of mediating software that manages the interactions between the two frameworks. The mediating software not only manages the interaction, but also contains the functionality required for closing the framework gap. Potential disadvantages are that it has proven to be rather difficult to develop mediating components and that the software becomes vulnerable for future changes of framework versions and thus is expensive to maintain.

Redesign and extend. If a developer has access to the source code of the framework and also intends to reuse the framework in future applications, the most fruitful approach may be to redesign the framework and extend the source code to achieve better domain coverage, thus bridging the gap.

5.4 Composition overlap of framework entities

When two frameworks represent the same real-world entity, from their respective perspectives and with their own representations, and these frameworks need to be composed, the two representations of the real-world entity have to be composed as well. This constitutes the problem of overlap of framework entities. The main cause for this problem is that both frameworks cover part of the application domain, since the real-world entity has been modeled from different perspectives. The composition of the two framework classes is often complicated due to the cohesive behavior of the classes since, after the composition, actions in one class may need to notify the other framework class. We present four solution approaches to the problem:

Multiple inheritance. An obvious solution to this problem is to use multiple inheritance, but this requires that the properties of the classes not affect each other and are mutually exclusive, which is seldom the case. The main issue is to provide a solution that takes care of necessary updates and conversions between different representations provided by the two frameworks.

Aggregation. An alternative solution is to use aggregation instead of inheritance, which will result in an aggregate class that has each framework representation as parts. The aggregate class is the application's representation of the real-world entity. However, this approach requires that the source code be available for the frameworks, since the involved frameworks have to be changed so that every location in the framework that references the framework-specific representation of the real-world entity is replaced by a reference to the aggregate class. In addition, it is necessary that the interface of the aggregate class contain the framework class interface for both frameworks and the necessary conversion and update operations. The implementation of the aggregate class must, in appropriate methods and places, include the necessary conversion and update behavior. The major drawback of this solution is that both frameworks have to be changed and that a considerable amount of implementation overhead is required.

Subclassing and aggregation. Another solution that does not require changes in the frameworks is to subclass each framework class that represents the real-world entity. Both subclasses are

involved in a bidirectional association relationship to enable update and conversion behavior. In addition, each subclass has to override the operations in the superclass. In the subclass implementation of relevant operations, there are calls to the corresponding subclass for conversion and updates and, in addition, the operations call the overridden operations in the superclass. A limitation of this solution is that the application's representation of the real-world entity is partitioned over two classes. To overcome this limitation, we may introduce an additional aggregate class that has the two subclasses as parts. The aggregate class solution also implies that the application-specific part of the code has to work with the aggregate class (to reach the part objects) and that there will be communication between the two subclass objects that will not pass the aggregate object.

Subject orientation. A fourth solution is provided by the subject-oriented programming approach [14]. Subject orientation supports composition of subjects-that is, an object-oriented program or program fragment such as a framework that models a domain in its own subjective way. Subjects can be composed, either as source code or as binary code. Since frameworks can be subjects, the subject-oriented approach may alleviate the problem of overlapping framework entities. However, although useful in theory, the lack of widespread tool support may make subject composition less appropriate in practice.

5.5 Composition of entity functionality

This problem occurs when a real-world entity has to be modeled through composition of parts of functionality from different frameworks, for example, composing application domain-specific functionality with persistence functionality. This means that the real-world entity is represented in an application domain-specific framework and has to be composed with functionality from a persistence framework. The cause for this composition problem of functionality is that the frameworks are designed for extension and thus it is not easy to add functionality. The cohesive behavior of a framework complicates the composition of functionality since it makes it difficult to break up the existing collaborations inside the framework and add the necessary functionality. Aggregation and multiple inheritance and the use of the Observer design pattern are discussed as possible

solutions in the following text. A third solution may be the use of the subject-oriented programming approach [14], as described in the preceding section.

Aggregation or multiple inheritance. The simplest solution to this problem is to use multiple inheritance or aggregation to compose the framework classes with the required functionality into a single class. The problem with these solutions is that state changes caused by messages to the application domain-specific part of the composed object will not automatically influence the persistence functionality part of the object. One way to solve this is to add update behavior in appropriate places in the frameworks and the composed class, but this is a rather complex action to perform.

Observer design pattern. To overcome the problem with the previous solution, the application domain-specific class can be extended with notification behavior, for example, through applying the Observer design pattern [5]. This may seem an obvious solution; thus, it should have been included in the application domain-specific framework from the beginning. But, as often is the case, the framework is originally designed for extension and not for composition.

6. Summary

Object-oriented frameworks provide an important step forward in software reuse, since frameworks allow reuse of large components instead of the traditional small, building block-like components. Initially, frameworks were primarily used as the only reused entity in application development, and the application was constructed by extending the framework with application-specific behavior. During recent years-among other reasons, due to the increased availability of frameworks for various domains-there is a movement toward the use of multiple frameworks in application development. These frameworks have to be composed to fulfill the application's requirements, which may lead to composition problems.

In this chapter, we have studied the framework composition problems that may occur during application development, their main causes, the solutions available to the software engineer, and the limitations of these solutions. The composition problems that we identified are related to the composition of framework control, the

composition of the framework with legacy components, the framework gap, the composition overlap of framework entities, and the composition of entity functionality. These problems are primarily caused by the cohesion of involved frameworks, the accuracy of the domain coverage of frameworks, the design intentions of the framework developer, and the lack of access to the source code of the framework. The identified problems and their causes are, to some extent, addressed by existing solutions, such as the Adapter design pattern, wrapping, or mediating software, but these solutions generally have limitations and either do not solve the problem completely or require considerable implementation effort.

To the best of our knowledge, no work exists that covers framework composition problems as a primary topic of study, but several authors have identified some composition problems. For instance, [15] suggests not to assume ownership of the event loop during framework design and to stress flexibility of the design. [17] proposes the use of software adapters as a means to overcome the problem of type-incompatible interfaces between components. This problem is similar to that of the composition of framework control. A software adapter uses protocols, which defines the allowed sequences of message exchanges between the components, and it compensates for differences between the component interfaces. The software adapter also allows message exchanges to be initiated from either of components. [16] identified the framework gap problem and suggests ways of dealing with it. Also, [6] discusses the notion of architectural mismatch, which, although not specific to object-oriented frameworks, may also affect framework composition. Finally, [7] discusses the problems of using wrappers to adapt components for use in applications.

The contribution of this chapter, we believe, is twofold. It provides software engineers using object-oriented frameworks for application development with an understanding of the problems they may experience during framework-based application development, as well as an understanding of the primary causes and solution approaches. Second, it provides researchers in object-oriented software engineering with topics that need to be addressed by future research.

References

[1] Betlem, B.H.L., R.M. van Aggele, J. Bosch, and J.E. Rijnsdorp. An Object-Oriented Framework for Process Operation. Technical report, Department of Chemical Technology, University of Twente, 1995.

[2] Bosch, J. Design patterns as language constructs. Journal of Object-Oriented Programming 11(2), May 1988:18-32.

[3] Bosch, J., P. Molin, M. Mattsson, P.O. Bengtsson, and M. Fayad. Object-oriented frameworks-Problems and experiences. In Object-Oriented Application Frameworks, M. Fayad, D. Schmidt, and R. E. Johnson, editors. New York: John Wiley & Sons, 1998.

[4] Bosch, J. Design of an object-oriented framework for measurement systems. Object-Oriented Application Frameworks, M. Fayad, D. Schmidt, and R.E. Johnson, editors. New York: John Wiley & Sons, 1998.

[5] Gamma, E., R. Helm, R. Johnson, and J.O. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.

[6] Garlan, D., R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. Proceedings of the Seventeenth International Conference on Software Engineering, pp.179-185, Seattle, WA, April 1995.

[7] Hölzle, U. Integrating independently-developed components in object-oriented languages. Proceedings of ECOOP 1993, Kaiserslautern, Germany, July 1993, pp. 36-56, O. Nierstrasz, editor, LNCS 707. New York: Springer-Verlag, 1993.

[8] Johnson, R.E., and B. Foote. Designing reusable classes. Journal of Object-Oriented Programming 1(2), June 1988:22-35.

[9] Lundberg, C., and M. Mattsson. On using legacy software components with object-oriented frameworks. Proceedings of Systemarkitekturer 1996, Borås, Sweden, 1996.

[10] Lundberg, L. Multiprocessor performance evaluation of billing gateway systems for telecommunication applications. Proceedings of the ICSA Conference on Parallel and Distributed Computing Systems, pp. 225-237, September 1996.

[11] Mattsson, M. Object-oriented frameworks-A survey of methodological issues. Licentiate thesis, Lund University, Department of Computer Science, 1996.

[12] Molin, P., and L. Ohlsson. Points & Deviations-A pattern language for fire alarm systems. Proceedings of the 3rd International Conference on Pattern Languages for Programming, Paper 6.6, Monticello IL, September 1996.

[13] Moser, S., and O. Nierstrasz. The effect of object-oriented frameworks on developer productivity. IEEE Computer, Theme Issue on Managing Object-Oriented Software Development, Mohamed E. Fayad and Marshall Cline, editors, 29(9)September 1996:45-51.

[14] Ossher, H., M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. Theory and Practice of Objects Systems 2(3):179-202.

[15] Pyarali, P., T.H. Harrison, and D. Schmidt. Design and performance of an object-oriented framework for high-speed electronic medical imaging. Computing Systems Journal, USENIX, 9(4), November/December 1996:. 331-375.

[16] Sparks, S., K. Benner, and C. Faris. Managing object-oriented framework reuse. IEEE Computer, Theme Issue on Managing Object-Oriented Software Development, Mohamed E. Fayad and Marshall Cline, editors, 29(9), September 1996:53-61.

[17] Yellin, D.M., and R.E. Strom. Interfaces, protocols, and the semi-automatic construction of software adapters. Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1994), pp. 176-190, Portland, OR,, October 23-27, 1994.

PAPER III

Observations on the Evolution of an Industrial OO Framework

Michael Mattsson and Jan Bosch

Published in the proceedings of ICSM'99, International Conference on Software Maintenance, Oxford, UK, 1999

III

Abstract

Recently an approach for identifying potential modules for restructuring in large software systems using product release history was presented [4]. In this study we have adapted the original approach to better suit object-oriented frameworks and applied it to an industrial black-box framework product in the telecommunication domain. Our study shows that using historical information as a way of identifying structural shortcomings in an object-oriented system is viable and useful. The study thereby strengthens the suggested approach in [4] and demonstrates that the approach is adaptable and useful for object-oriented systems. The usefulness of the original approach has been validated through this study too.

Keywords:
Object-oriented frameworks, Software evolution, Object-oriented framework evolution, Experience report

1. Introduction

In recent years, object-oriented framework technology has become common technology in object-oriented software development [1, 2,

5, 10, 11]. Frameworks allow for large-scale reuse and studies show that the use of framework technology reduces development effort, e.g. [9].

It is argued that designing a good framework requires several iterations [8, 6, 12]. Of special importance in the development of a framework is the identification of the stable and variable parts of the design. The stable part of the design represents a reusable core of application domain functionality, which is reused for all applications. The stable part provides hooks that are intentionally designed to provide for customising or tailoring of the framework (i.e. the variable parts).

As with all software, frameworks evolve to meet new demands from customers/framework users. To alleviate or remedy the problem of complex software and the difficulties of evolving it, it is important to identify modules that are likely candidates for restructuring. This is of importance in framework evolution since there are large amounts of effort invested in the design of the framework and information about where likely changes occur may reduce the cost of redesign the framework to meet the new demands. Studies show that software engineers can only identify a subset (<50 percent) of future changes and they can not provide the complete picture of change [7]. Thus, a more objective method for identifying structural shortcomings will provide more reliable input for maintenance effort estimations and thereby more accurate estimates.

Recently an approach for identifying potential modules for restructuring in large software systems (about 10 millions lines of code) has been proposed by Gall et al. using product release history [4]. We have adapted the approach to better suit object-oriented frameworks. Instead of programs we are investigating classes as the observed entities.

In this paper, we describe the evolution of an industrial black-box framework product, the Mediation framework, in the telecommunication domain. The investigation covers four consecutive versions of the framework that were developed over a period of six years. Over 30 instantiations of the framework have been delivered to customers.

The new versions of the framework were forced by product improvements, e.g. better performance, and new customer demands. Our analysis is based on information about modules and classes (the class declarations only) of the framework. The constant development

of the framework has increased both the size and the complexity. The size of the Mediation framework has increased from 322 classes to 598 classes.

The objective of our study is to evaluate the approach to identifying potential structural shortcomings of the framework using the aforementioned framework as a case. The approach uses historical data, as in the case by Gall et al. [4]. Once the eventual shortcomings have been identified, the relevant subsystems or modules can be subject to restructuring. As an effect of our adapted approach we are also validating the usefulness of the original approach by Gall et al.

The paper is organised as follows: In section 2, the Mediation framework is described in sufficient detail to get an understanding of what kind of software system is the subject of the case study. The subsystems and modules have been renamed to make the discussion clearer and avoid bothering the reader with unnecessary details. Section 3 describes the evolution observations on the complete Mediation framework. In section 4, the evolution observations of the subsystems and their modules is described. Section 5 discusses related work and the paper is concluded in section 6.

2. The Case Study

2.1 The method

The method used in our study is an adaptation of the approach proposed by Gall et al. [4]. The focus for the study by Gall et al. was on macro-level software evolution by tracking the release history of a system. Their system comprises about 10 million LOC and was structurally divided into a number of subsystems where each subsystem where further divided into modules and each module consisted of a number of programs. Based on historical information about the programs, subsystems etc. they investigated the change rate, growth rate and size of the software (and its parts) using the *version numbering* of the programs as the units for 20 releases of the software systems. The method steps in the approach are

1. calculate, for all releases, the change and growth rate for the whole system,
2. calculate, for all releases, the change and growth rate for each of the subsystems,
3. for those subsystems that exhibit high growth and change rates calculate, for all releases, the change and growth rates for the modules,
4. those modules that exhibit high change and growth rates are identified as likely candidates for restructuring.

In our study we have adapted the approach to better suit for smaller systems (about 600 classes). We use the same decomposition of the system into subsystems and modules. Regarding the modules these are in our study composed of classes (not programs). So our calculations of change and growth rates are made in terms of classes as the units. Changed classes are identified by a *change in the number of public operations* in the class interface. This is because a change in the public interface represents an addition or better understanding of the functionality that the class has to offer the rest of the system. Changes to the private part of the class interface are mostly related to perfective changes such as performance improvements and as such less interesting in this study. Regarding the method steps we use the same steps as the approach by Gall et al.

Essentially, the adaptation is the change of programs to classes as the observed unit to make the approach useful for object-oriented software systems.

2.2 The software system

The subject for the study consists of four successive versions of a proprietary black-box application framework. This framework encompasses the Billing Gateway product developed by Ericsson Software Technology. The framework provides functionality for mediation between network elements, i.e., telecommunication switches, and billing systems (or other administrative post-processing systems) for mobile telecommunication. Essentially, the Mediation framework collects call information from mobile switches

(NEs), processes the call information and distributes it to among others, billing processing systems. Figure 1 presents the context of the Mediation framework graphically. The driving quality requirements in this domain are reliability, availability, portability, efficiency (in the call information processing flow) and maintainability. The framework is black-box, i.e., the framework provides a number of predefined objects for application configuration. The four versions of the Mediation framework have been developed during the last six years.

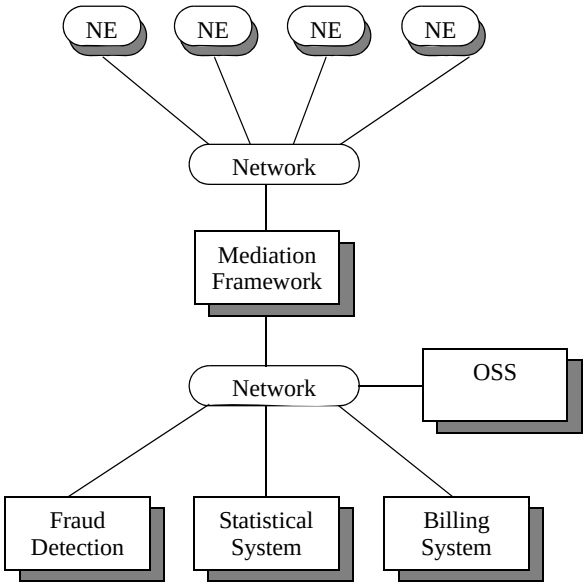

Figure 1. *The Mediation framework*

The structure of the Mediation framework system is a traditional tree structure with four levels; the system, subsystem, module and class level. Each level consists of one or more elements and each element on a specific level is connected to one element on a higher level. The logical structure of the Mediation framework has been defined during the evolution of the four versions of the framework. The time spans between the versions are approximately 18 months each.

3. Framework Evolution Observations

Based on the structural information derived from the four framework versions we are able to make statements about the Mediation framework's evolution. For example, the size of the framework and its subsystems and the number of classes changed in a particular module from one version to the next.

Using the extracted information we try to identify potential shortcomings in the current structure of the Mediation framework.

We have extracted information, about the following subset of system properties, which is an adaptation of the descriptions found in [4]:

- The *size* of each system, subsystem or module is defined as the number of classes it contains. We use the notion of classes as the size counting entity instead of measuring the size of the source code. This is because it is easier to collect necessary information and measure on the detailed design level compared to the source code level.
- The *change rate* is the percentage of classes in a particular system, subsystem or module that changed from one version to the next. To compute the change rate, two versions are required for comparison. The relative number of the changed classes represents the change rate.
- The *growth rate* is defined as the percentage of classes in a particular system, subsystem or module, which have been added (or deleted) from one version to the next. To compute the growth rate, two versions are compared and the numbers of the added and deleted classes are computed. The relative number of the new classes (i.e. the difference between added and removed classes) represent the growth rate.

Using the information gathered from the Mediation framework we can make statements about framework evolution observations related to: 1) the whole Mediation framework system, 2) the subsystems of the Mediation framework; and 3) the modules of particular subsystems that we chose for further study.

3.1 System observations

Diagram 1 shows the historical development of the size of the Mediation framework. The framework has a high growth rate; it consisted of 322 classes initially and 598 classes at version 4.0. Thus, the total size of the Mediation framework increased by over 85 percent over the four versions. From the diagram we also see that the size of the Mediation framework is growing in a way similar to linear.

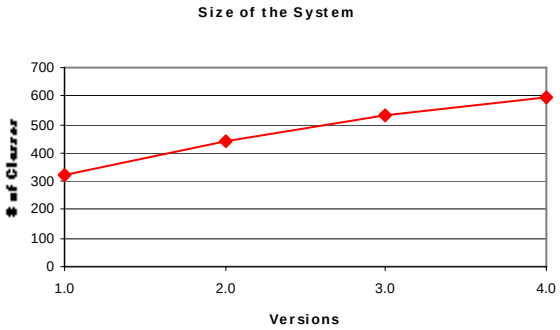

Diagram 1. *Evolution of the size of the Mediation*

Diagram 2 shows the number of classes that were added in each version (i.e. added minus removed classes). For example, the diagram shows that at version 2.0 more than 120 classes were added to the system. A general observation is that the *number of added classes is decreasing* with newer versions of the framework. This can be taken as an indication that the framework is becoming more mature, i.e. achieving a structural stability.

Diagram 3 compares the change and growth rates for each version of the Mediation framework. In general, the change rate is higher than the growth rate. The growth rate is decreasing with successive versions of the framework, from about 40 percent at version 2.0 down to about 15 percent at version 4.0. This *decrease in growth rate* is an another indication of the framework's maturity.

The change rate is increasing at version 4.0 and is not following the trend for version 2.0 and 3.0. One reason for this may be an *architectural change*, i.e. a change affecting entities throughout the whole system. In the Mediation framework the architectural change was the introduction of hot billing.

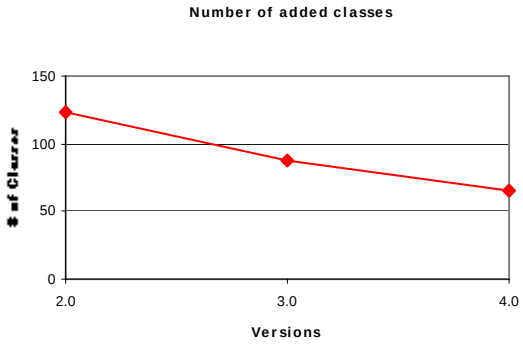

Diagram 2. *Number of added classes in each version*

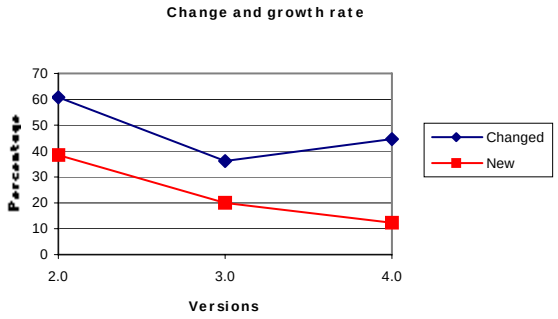

Diagram 3. *Comparison of the change and growth rates*

The change rates are around 40-50 percent in between all versions. In version 2.0 and 3.0 this may be a sign of better domain understanding but in version 4.0 the major reason is the architectural change introduced by the hot billing requirement.

A summary of the findings at the system level is:

- The size of the Mediation framework is increasing almost linearly.
- The development of the whole Mediation framework becomes stable with respect to the growth rate that is decreasing over the versions.

- The change rate is about 40-50 percent for all versions. The change rate shows a decrease to version 3.0 but increases again in version 4.0. The increase of the change rate in version 4.0 is an indication that some unexpected evolution has taken place, i.e. evolution with architectural impact.

The latter two observations are signs of structural stability of the Mediation framework eventually with a small exception for the increase of the change rate in version 4.0.

The observations on the system levels indicate that the framework evolves in a satisfactory fashion. This is because the growth rate is decreasing and the change rate is at an acceptable level (<50 percent). However, the observations are only valid on system level and it may be the case that the subsystems do not behave in the same way. In the next section we will characterise the evolution of the four subsystems for the Mediation framework.

4. The Evolution of Subsystems

Up to now we have achieved an impression of how the overall system has evolved over time. In this section, we will examine the subsystems further to see if the subsystems evolve the same way as the whole system.

Table 1. Change and growth rates of subsystems

Subsystem	Change rate	Growth rate	Relative size
A	51%	191%	34%
B	64%	153%	24%
C	92%	191%	37%
D	75%	700%	5%

In table 1, we see the change and growth rates for the four subsystems in the Mediation framework. Subsystem D exhibits the highest growth rate, 700 percent, but since subsystem D only comprises 5 percent of the total system size in version 4.0 the subsystem is of minor interest. Subsystems A and C have a growth rate on 191 per-

cent each. Their relative sizes of the whole system are 34 percent for subsystem A and 37 percent for subsystem C, respectively. Thus, both are candidates for further study but since Subsystem A represent the GUI part of the system and comprises, to a very high degree, automatic generated UI classes it is not a representative subsystem.

If we consider the change rate it is obvious that subsystem C has undergone major changes between version 1.0 to version 4.0 since it has a change rate of 92 percent. It may also be of interest to investigate subsystem B since its change rate, 64 percent, is higher than the change rates for the whole system (diagram 3). Subsystem C represents a major part of the system, 37 percent of the classes, and we will investigate it further on the module level in the following subsection.

Concluding, in the following section we will further investigate the evolution characteristics for Subsystem C due to its high growth and change rates and Subsystem B due to its relatively high change rate.

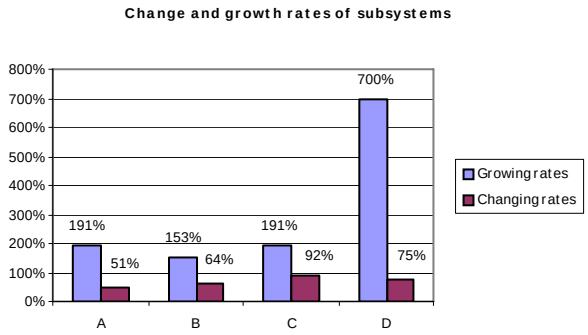

Diagram 4. *Change and growth rates of subsystems*

4.1 The evolution of subsystem C

Subsystem C exhibits the highest growth and change rates among the subsystems. These facts make the subsystem a likely candidate for restructuring and/or reengineering activities. We will investigate how the modules in subsystem C evolves and whether they possess the same evolution characteristics as the whole subsystem.

Subsystem C consists of 5 modules named module C1 to C5. Their relative size in Subsystem C is presented in diagram 5. We see

that all five modules comprise between 10-35 percent of the subsystem. Modules C3 and C4 have a relative size larger than 25 percent whereas as modules C1, C2 and C5 all have a relative size below 15 percent. In fact, module C3 and C4 are two of the three largest modules of the Mediation framework.

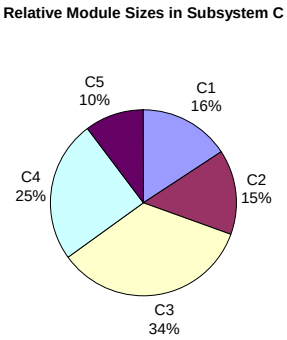

Diagram 5. *The relative size of modules in subsystem C*

Diagram 6, which shows the number of classes for each module in Subsystem C, confirms that module C3 and C4 are large modules and that they both have had a large increase of the number of added classes. In addition modules C1, C2 and C5 has been relatively stable in size but a minor increase can be observed for module C1.

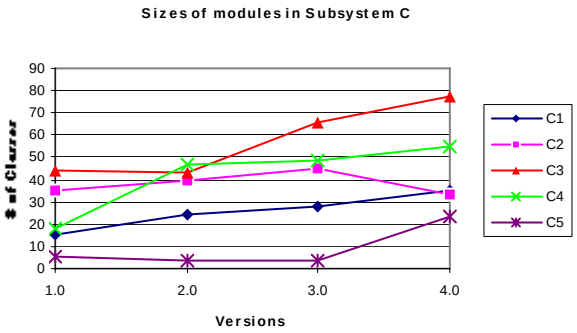

Diagram 6. *Sizes of modules in subsystem C*

Diagram 7 shows the growth and change rates for the modules in subsystem C from version 1.0 to version 4.0. Module C5 has a growth rate of 460 percent that is extremely high. Also modules C1 and C4 have high growth rates, 233 and 306 percent respectively.

Module C5 has the highest growth rate but is the smallest module in the subsystem. Also Module C4 has a high growth rate and it represents 25 percent of the subsystem. Through discussions with the software engineers it was clarified that the reason for the high growth rate is that the module is responsible for coding and decoding the Call Data Records which exist in a number of formats. Thus, the high growth rate is the result of an increased number of formats supported by the Mediation framework, e.g. module C4 handles a number of different Call Data Record formats which results in a set of classes whose objects are used as parameters in the framework. I.e. a typical black-box framework evolution observation.

All modules except Module C5 exhibit high change rates. A reason for the relative low change rate for Module C5 in the subsystem is that it captures the encapsulation of some other software such as database systems.

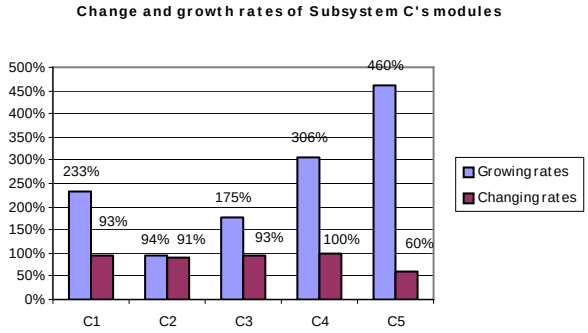

Diagram 7. *Change and growth rates of subsystem C's*

Regarding the change rates for the modules in Subsystem C we see in diagram 8 that they behave as the system in whole, i.e. the change rate is increasing from version 3.0 to version 4.0. Modules C3 and C4 exhibit the highest change rates throughout all the versions.

Concerning the high change rate, all modules in subsystem C have high change rates and the main reason for this is two-fold;

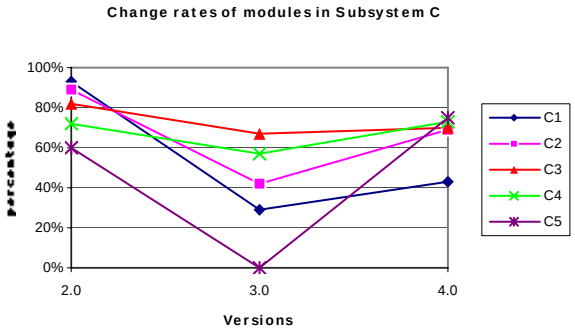

Diagram 8. *Change rates of modules in subsystem C*

1. a design concept called Node has been divided into two concepts, Childnode and ParentNode, to achieve higher maintainability and lower complexity,
2. another design concept called Point has changed from being file-oriented to be transaction-oriented due to the need of provide decreased latency for the Call Data Record (the billing information input) processing. This to fulfil the requirement of hot billing.

To summarise the findings of Subsystem C we can foresee changes in all modules due to the changes of the two design concepts. These changes have presumably not achieved full effect on the system design. Regarding Module C4 and its growth rate; this is viewed as an anticipated kind of evolution due to the nature of black-box frameworks.

4.2 The evolution of subsystem B

We present a few observations of the Subsystem B evolution characteristics since the subsystem showed a relatively high change rate.

In diagram 9, we see the change and growth rates for the modules in Subsystem B. We see that module B4 has the highest growth rate, 242 percent, and that module B2 has the highest change rate, 73 percent. Module B2 is in fact the largest module in Subsystem B representing 68 percent of the classes in the subsystem.

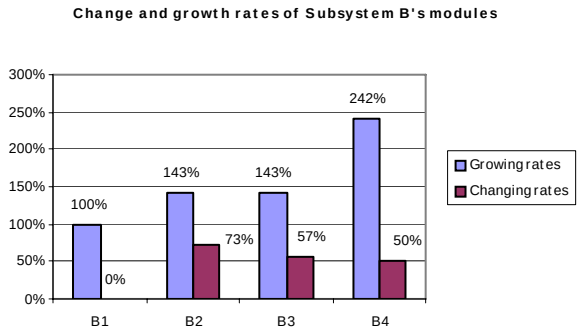

Diagram 9. *Change and growth rates of subsystem B's modules*

The high growth and change rates of Module B2 are explained by the development from a simple arithmetic language to a mini-language for handling data units (a design concept based on ASN.1 descriptions).

Module B4 has the highest growth rate but it is relatively small, 30 classes, and the increase can be explained by an increased number of supported operating system thread representations (i.e. a typical black-box framework evolution observation similar to the one described earlier).

To summarise subsystem B, we have again an anticipated kind of evolution for Module B4. We may foresee changes in Module B2 or modules that are using Module B2 depending on the future evolution of the mini-language.

Through the examination of the four subsystems, a different picture evolves since the subsystems do not exhibit the same behaviour as the framework as a whole. Especially subsystem C and B are likely candidates for forthcoming restructuring and/or re-engineering activities.

5. Related Work

Related work includes, of course, the work by Gall et al. [4] where historical information about modules and programs is used as a means to identify structural shortcomings of a large telecommunication switching system. Our work uses a similar approach but on a smaller system, 300-600 classes, and we use historical information

about modules and classes, which are entities of smaller granularity than in the work Gall et al. Recent work by Lindvall and Sandahl [7] shows that software developers are not so good at predicting from a requirements specification how many and which classes that will be changed. Their empirical study shows that only between 30-40 percent of the actual changed classes were predicted to be changed by the developers. This can be seen as an argument for a more objective approach for identifying change-prone and evolution-prone parts of a software system.

Other research related to framework evolution is the work by Roberts and Johnson [12] and by Erni and Lewerentz [3]. Roberts and Johnson present a pattern language that describes typical steps for the evolution of an object-oriented framework. Common steps in the evolution are development of a white-box framework (extensive use of the inheritance mechanism), the transition to a black-box framework (extensive use of composition), development of a library of pluggable objects etc. These steps give a coarse-grained description of a framework's evolution. However, they do not explicitly describe where and how the framework evolves when it has reached a certain state.

Erni and Lewerentz describe a metrics-based approach that supports incremental development of a framework. By measuring a set of metrics (which requires full source code access) an assessment of the design quality of the framework can be performed. If the metric data is within acceptable limits, the framework is considered to be good enough from a metrics perspective otherwise another design iteration has to done. The approach is intended to be used during the design iterations, before a framework is released and does not consider long term evolution of a framework.

6. Conclusion

We have studied the structure of four consecutive versions of an object-oriented black-box framework in the telecommunication domain. Based on historical information about the classes and their public interfaces we have applied an adapted version of the approach in [4] in order to identify structural shortcomings of the framework’s structure by measuring the growth and change rates of the system.

Our study exhibits results similar to the study by Gall et al., i.e. there is a difference in the behaviour of the whole system versus its subsystems. A relatively stable evolution over time was observed at the system level, but detailed studies of the subsystems showed that subsystem C and B exhibit different characteristics regarding the change and growth rates when compared to the whole system. These major differences would not been identified only by an analysis of the whole system. Two modules were identified as potential subjects for redesign, i.e. module C4 and B2. In addition, to a lesser extent, module C3 is a likely candidate for redesign.

Both our study and the one by Gall et al. show that using historical information as a way of identifying structural shortcomings in a software system is viable and useful. In addition, the approach suggested in [4] is possible to use for object-oriented systems on design level using class interfaces and changes to these interfaces as the observed units. In other words, the approach by Gall et al. is possible to scale down to object-oriented systems.

The validity of the approach of using historical is strengthened by the following. First, our results are similar to the ones by Gall et al., i.e. there exists a difference in the behaviour of the whole system versus its subsystems. Second, we presented the results of this study to the software development organisation owning the studied framework. They agreed to our observations, found the results interesting and consider using the approach in the future.

Acknowledgments

We would like to thank Ericsson Software Technology for providing us access to the Mediation framework, in particular Mikael Roos and Drazen Milicic.

References

[1] G. Andert, 'Object Frameworks in the Taligent OS,' Proceedings of Compcon 94, IEEE CS Press, Los Alamitos, California, 1994.

[2] J. Bosch, P. Molin, M. Mattsson, PO Bengtsson, M. Fayad,‘Object-Oriented Frameworks - Problems & Experiences’, In *Object-Oriented Foundations of*

Framework Design, M. E. Fayad, D. C. Schmidt, R. E. Johnson (eds.), Wiley & Sons, 1999, forthcoming.

[3] K. Erni, C. Lewerentz, ‘Applying design metrics to object-oriented frameworks,’ *Proceedings of the Metrics Symposium*, 1996.

[4] H. Gall, M. Jazayeri, R. G. Klösch, G. Trausmuth, ‘Software Evolution Observations Based on Product Release History’, Proceedings of the Conference on Software Maintenace - 1997, 1997, pp. 160-166.

[5] H. Huni, R. Johnson, R. Engel, 'A Framework for Network Protocol Software,' *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages and Applications,* Austin, USA, 1995.

[6] R.E. Johnson, V.F. Russo, 'Reusing Object-Oriented Design,' *Technical Report UIUCDCS 91-1696*, University of Illinois, 1991.

[7] Lindvall, M. and Sandahl, K. (1998). How Well do Experienced Software Developers Predict Software Change? Journal of Systems and Software, 43(1):19 – 27.

[8] M. Mattsson, 'Object-Oriented Frameworks - A survey of methodological issues,' *Licentiate Thesis*, LU-CS-TR: 96-167, Department of Computer Science, Lund University, 1996.

[9] M. Mattsson,’Effort Distribution in a Six Year Industrial Application Framework Project’,*Proceedings of the International Conference on Software Maintenance - 1999*, Oxford, England, 1999.

[10] Peter Molin and Lennart Ohlsson, 'Points & Deviations -A pattern language for fire alarm systems,' *Proceedings of the 3rd International Conference on Pattern Languages for Programming*, Monticello IL, USA, September 1996.

[11] S. Moser, O. Nierstrasz, 'The Effect of Object-Oriented Frameworks on Developer Productivity,' *IEEE Computer*, pp. 45-51, September 1996.

[12] D. Roberts, R. Johnson, 'Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks,' *Proceedings of the Third Conference on Pattern Languages and Programming*, Montecillio, Illinois, 1996.

PAPER IV

Stability Assessment of Evolving Industrial Object-Oriented Frameworks

Michael Mattsson and Jan Bosch

Accepted with minor revisions to the Journal of Software Maintenance, 1999

IV

Abstract

Object-oriented framework technology has become a common reuse technology in software development. As with all software, frameworks evolve over time. Once the framework has been deployed, new versions of a framework potentially cause high maintenance cost for the products built with the framework. This fact in combination with the high costs of developing and evolving a framework make it important for organisations to achieve controlled and predictable evolution of the framework's functionality and costs. We present a metrics-based framework stability assessment method, which have been applied on two industrial frameworks, from the telecommunication and graphical user interface domains. First, we discuss the framework concept and the frameworks studied. Then, the stability assessment method is presented including the metrics used. The results from applying the method as well as an analysis of each of the frameworks are described. We continue with a set of observations regarding the method, including framework differences that seem to be invariant to the method. A set of framework stability indicators based on the results is then presented. Finally, we assess the method against issues related to management and evolution of frameworks, framework deployment, change impact analysis, and benchmarking.

1. Introduction

During recent years, object-oriented framework technology has become common technology in object-oriented software development [3][11][15][18] since it bears the promise of reduced development effort, e.g. [16], through large-scale reuse and increased quality. An object-oriented application framework is a reusable asset that constitutes the basis for a number of applications in the domain captured by the framework. Examples of application frameworks are Java AWT [8], Microsoft Foundation Classes [19] and ET++ [21] in the domain of graphical user interfaces. Examples of application frameworks in other domains include fire alarm systems [15] and measurement systems [4].

As with all software, frameworks normally evolve through a number of iterations, leading to new versions, in response to the incorporation of new or changed requirements, better domain understanding, experiences and fault corrections. Once the framework has been deployed, new versions of a framework cause potentially high maintenance cost for the products built based on the framework. This fact in combination with the high costs of developing and evolving an application framework indicates the importance of understanding and characterising framework stability and evolution.

One of the main objectives for management of software development organisations is to achieve controlled and predictable evolution of the framework's functionality and its development, maintenance and framework instantiation costs. This requires that there must exist methods assisting and providing management with information about the framework and its evolution. Areas of importance in the management and evolution of an object-oriented framework are:

- Framework deployment: Assessment of the framework to decide when it can be released for regular product development in the organisation or placed on the commercial market. The assessment is necessary to avoid the shipping of frameworks that do not fulfil their functional and quality requirements and, consequently, will cause unnecessary maintenance and release costs. The assessment data can in some cases, if publicly available, be used by potential customers for evaluating and select-

ing a framework from several alternatives. For instance, the assessment may cover aspects such as reliability and robustness for its users.

- Change impact analysis: Another type of framework analysis is concerned with assessing the impact of changes on the next framework version and assessing the impact on the applications built which have to replace the old framework version with a new one. The assessment results can be used by management to predict required maintenance effort for both the framework and the applications incorporating instantiations of the frameworks.

- Benchmarking: Empirical information collected from the assessment of successful frameworks can be valuable in guiding the development and evolution of new frameworks and can be used to develop criteria for framework classification, e.g. benchmarking, in terms of structural stability or other framework maturity aspects [1]. This is desirable since it is hard and expensive to develop and maintain object-oriented frameworks and empirical data is valuable input to the cost and effort estimation activities. Currently, little information exists about the cost and benefit of the development, maintenance and use of framework technology is exists. Empirical information about the distribution of effort in framework development, evolution and instantiation will help in the process of allocating specialists and other personnel to the developments as well as in effort estimations. For example, in [12] we present the effort distribution for the initial development, maintenance and instantiation of a six-year industrial object-oriented framework is presented.

In this paper we present a method, stability assessment, which is used for characterising framework evolution, and, consequently, addresses the aforementioned issues described above.

The stability assessment method is a metrics-based assessment method for assessing structural and behavioural stability of an evolving object-oriented framework. The notions of structural and behavioural stability are viewed as the dual property of evolvability. Through collection of a set of architectural and class level metrics for two or more framework versions together with the calculation of some aggregated metrics, the obtained metric values are used for the

assessment of the framework. Thus, the purpose of the method is to decide how structural and behavioural stable a framework under evolution is in terms of the metrics defined. Based on the experiences of applying the method on two frameworks we present a set of framework stability indicators.

The stability assessment method has been applied to two object-oriented frameworks, one proprietary application framework in the telecommunication domain, the Billing GateWay (BGW) framework and one commercial application framework in the graphical user interface domain, the Microsoft's Foundations Classes (MFC) framework [19]. Four consecutive versions of the BGW framework, that were developed over a period of six years, and five versions of the MFC framework have been analysed. In [14] we have presented some of the results of applying the method. Through the application of the method on these two frameworks it is possible to better evaluate them with respect to the aforementioned issues concerning controlled and predictable framework evolution i.e. framework deployment, change impact analysis and benchmarking. In addition, the following aspects will be discussed: difficulty to apply the method, the need of tool support and experienced advantages and disadvantages of the method.

It is important to notice that the aforementioned topics and the stability assessment method are not directly intended to be used for the project manager developing the first version of an object-oriented framework. Neither it is fully useable for teams reusing the framework. The usefulness is for the teams maintaining and evolving the framework. In particular, it is worth noticing that a developed framework that is intended to be reused several times has to be handled by the organisation as a product. Thus, long term commitment for the maintenance and evolution of the framework is required. It is in this organisational and long-term context the stability assessment method and the data collection are justified. Through routinely and systematically applying the method and collecting the data the organisation will achieve knowledge of the evolution of object-oriented frameworks. Currently, little knowledge about the stability and evolution of object-oriented frameworks is available.

The contribution of this paper is two-fold:

- Extension and application of a framework stability assessment method on two industrial object-oriented frameworks.

- Presentation of framework stability characterising data together with five framework stability indicators.

The organisation of the paper is as follows. Section 2 presents the notion of, and terminology related to object-oriented frameworks. The two studied frameworks are described in section 3. The stability assessment method is described in section 4. Section 5 contains the results from applying the method to the frameworks together with an analysis of the results. Section 6 discusses a set of observations related to the results, the method and the studied frameworks. In section 7 the method is assessed with respect to the aforementioned management issues and related work is presented in section 8. Section 9 concludes the paper.

2. Object-Oriented Frameworks

In this section we present the notion of object-oriented framework, an overview of major framework development activities and some different maturity stages a framework can have.

Object-oriented frameworks are reusable designs of all or part of a software system. A framework is described by a set of abstract classes and the way instances (objects) of those classes collaborate [18]. The development cost of frameworks is high and it must be easy to use. They must provide enough features to be used and hooks supporting features that are likely to change.

The process of developing and using an object-oriented framework comprises three major activities; problem domain analysis, framework design and instantiation of the framework (see Figure 1).

Problem domain analysis. Domain analysis can be viewed as a broader and more extensive analysis that tries to capture the requirements of the complete problem domain including future requirements. Depending on the problem domain different kind of information sources are used to obtain knowledge about the domain. Examples of sources are previous experiences of the domain, domain experts and existing standards. If the problem domain is large or covers a complete application domain, specialised domain analysis methods [17][18] may be used that provide systematic method steps for performing the analysis.

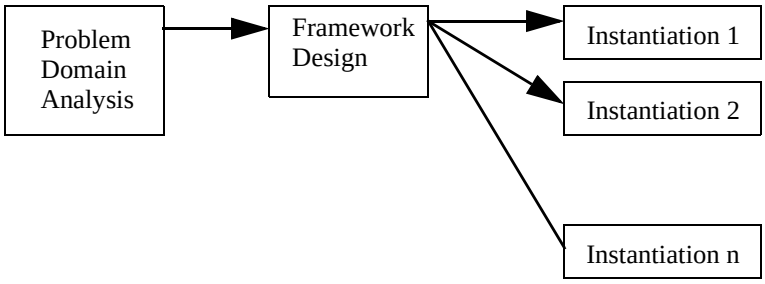

Figure 1. *Development and usage of*

Framework design. The objective with this activity is to end up with a flexible framework. The activity is effort consuming since it difficult to find the right abstractions and identify the stable and variable parts of the framework, which often results in a number of design iterations. To improve the extensibility and flexibility of the framework to meet the future instantiation needs, design patterns [7] may be used. The resulting framework can be a white-box, black-box or visual builder framework (see below) depending on the domain, effort available and intended users of the framework.

Instantiation of the framework. The instantiation of a framework differs depending on the kind of framework, white-box, black-box or visual builder (see below). A framework can, if small and specialised, be instantiated one or many times within the same application, instantiated in different applications and in the cases of one large monolithic framework the instantiation activity results in an application.

Roberts and Johnson [18] describe the typical maturation path an object-oriented framework takes. We describe the four major different kinds of frameworks that may be developed.

White-box framework. A white-box framework is relying on the inheritance mechanism as the primary instantiation technique. The instantiations of a concept are easily made through creating a subclass of an appropriate abstract class. A disadvantage of white-box frameworks is that one has to know about internal class details and that the creation of new subclasses requires programming.

Black-box framework. A black-box framework differs from a white-box framework in that the primary instantiation technique is composition. Thus, a black-box framework allows one to combine

the objects into applications and one does not need to know internal details of the objects. The use of composition to create applications implies that programming is avoided and it is possible to allow the compositions to vary at run time. Often the initial design of a framework is a white-box framework, whereas subsequent versions evolve into a black-box framework.

Visual builder framework. To evolve to the visual builder stage the framework must be a black-box framework. A black-box framework makes it possible to develop an application by connecting objects of existing classes. An application consists of different parts. First, a script is used that connects the objects of the framework and then activates them [18]. The other part is the behaviour of the individual objects, which is provided by the black-box framework. Thus, framework instantiation in the black-box case is mainly a script that is similar for all applications based on the framework. One now develops a separate graphical program that encompasses the black-box framework that makes it possible to specify the objects to be included in the application and the interconnection of them graphically. Thus, we have a visual builder framework. The visual builder framework generates the code for the application from the specification. With addition of the graphical features, the visual builder framework provides a user-friendly graphical interface, which makes it possible for domain experts to develop the applications by manipulating images on the screen.

Language Tools. The visual builder framework creates complex composite objects. These compositions have to be easily inspected and debugged. The visual builder framework actually is a graphical domain-specific programming language. Thus, one can create specific language tools for the framework that support inspection and debugging of the relevant complex composite objects.

3. The Two Framework Cases

In this section we present the two frameworks studied. The context of the BGW framework together with a structural view of its decomposition is presented. In addition, we present how to instantiate the framework through a graphical configuration window together with the major requirements causing the evolution for each framework

version. The MFC framework we describe rather briefly since it is relatively well known. For a more detailed presentation of the framework we recommend, e.g. [19].

3.1 The Billing GateWay framework

The Billing GateWay (BGW) framework is a major part of the Billing GateWay product developed by Ericsson Software Technology AB, and provides functionality for billing data mediation between network elements, i.e., switches, billing systems, or other post-processing systems for mobile telecommunication. The framework collects call information from mobile switches (NEs, Network Elements), processes the call information and mediates it to billing processing systems. Figure 2 presents the context of the BGW framework graphically. The driving quality requirements in this domain are reliability, availability, portability, call records throughput and maintainability.

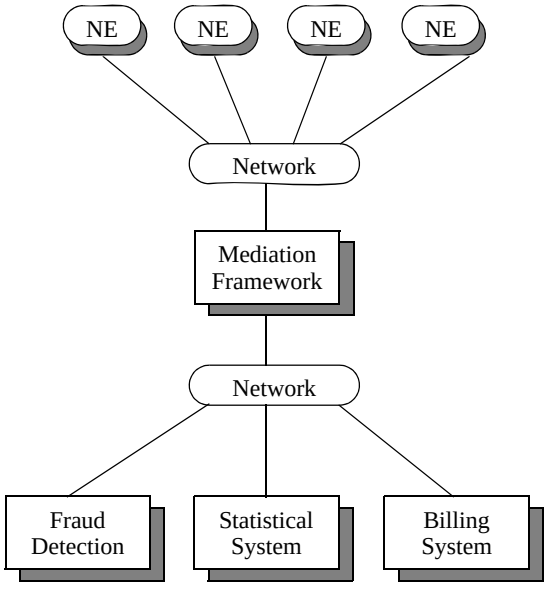

Figure 2. *The Billing GateWay framework*

The BGW framework is a large monolithic framework, which is intended to be instantiated into applications. The BGW framework

has from the first version been a visual builder framework. That is, a black-box framework with an associated graphical instantiation interface and application generation functionality. The BGW framework provides a number of pre-defined objects for graphical configuration of the actual application. Successively, from the second version of the BGW framework, a few specific language tools have been developed. Four versions of the framework have been developed, all in C++.

In Figure 3 a more detailed structural view of the BGW framework can be found. The BGW framework is divided into four major blocks and a set of system wide software units. The GUI block is responsible for the user interface, the Kernel block has responsibility for the main control and steers the flow of data from the Communication block to the Processing block and back to the Communication block. The Processing block processes the Call Data Records through different kinds of operations such as decoding, encoding, filtering, formatting, splitting and routing. The Communication block provides the interfaces to the communication protocols (protocols, e.g. RPC, and formats, e.g. FTAM, for distribution and collection). The set of System-wide software units are accessible from the other blocks and handles issues such as exception handling, threads, databases and some other system-wide issues.

In Figure 4 a snapshot of the graphical configuration window that is used for instantiating the BGW framework is shown. At the left in the window a palette exist where it is possible to select the different type of entities and associated operations to be performed which should constitute the instantiated application. For example, it possible to select NEs (e.g. a mobile switch) and PPSs (post processing systems) and the different kinds of processing such as filtering and formatting. At the right in the window, we see from the left side, two NEs, then two different filters, two formatters and at the right, two post-processing systems. Each of these entities has during the configuration/instantiation to be specified in more detail, through entity-specific menus.

When all entities have been specified, we have specified the application to be developed based on the framework, and it is possible to generate the application. All the specification data is divided to appropriate classes and objects and the application is generated.

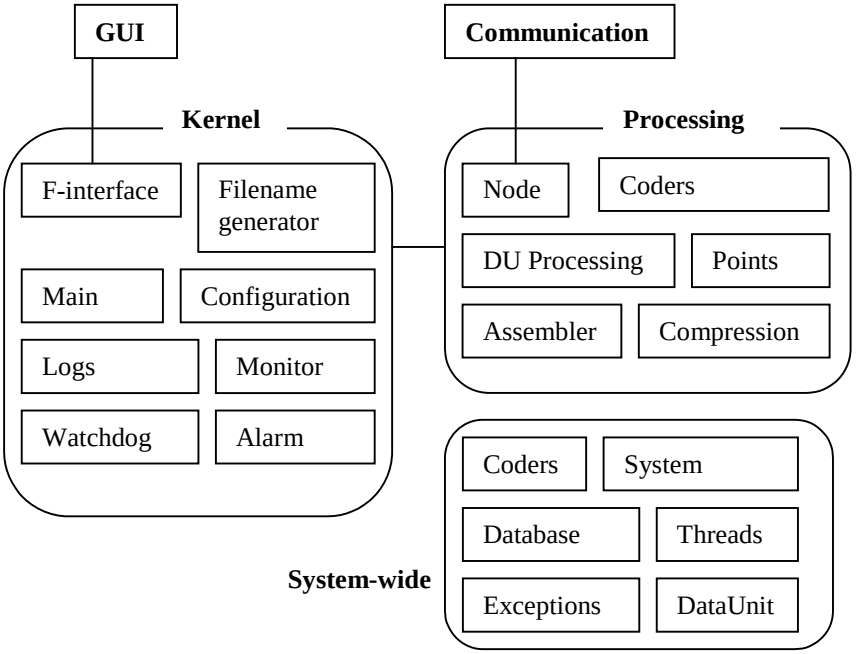

Figure 3. *Structural view of the BGW framework.*

To give an understanding of the evolution of BGW framework we describe the major requirements for the four framework versions since they have impact on the structure and functionality of the framework.

Version 1. The major requirement for the development of version 1 of the framework was to develop an architecture that supported the identified variable and stable parts of the domain. The organisation could here rely on the experience from a previous developed application in the same domain.

Version 2. The major requirements for version 2 of the framework were increased flexibility and robustness. This was achieved through refinement of some of the design concepts, improve encapsulation, identification of minor domain specific libraries which provide generalised and abstract routines for data handling and formatting.

Version 3. The major requirement for the development of version 3 was the introduction *of hot billing*. Hot billing makes it possible to collect call data records from network elements and distributes them to post processing systems within seconds of call completion. Thus, hot billing opens up for invoicing services tailored to specific cus-

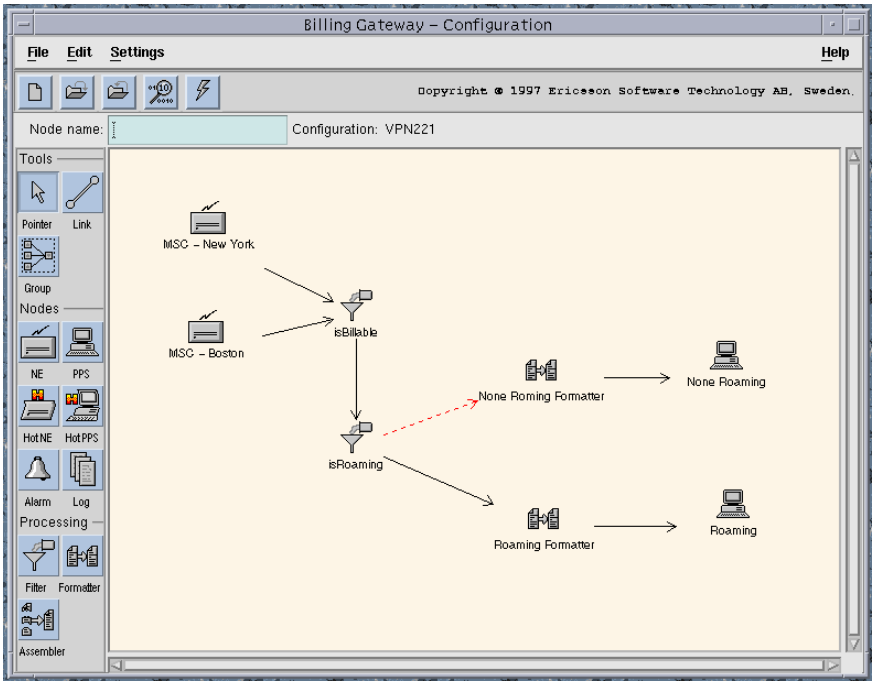

Figure 4. *The graphical configuration window for the BGW*

tomer needs, such as real-time billing and phone rental. This requirement caused major changes of the data processing in the system.

Version 4. For version 4 of the BGW framework major requirements was portability to new operating platforms and advanced data processing such as call data record matching and rating. Matching makes it simple to collect data from different nodes and times and thereby simplifying the charging of services with more than one call data record such as distance related charging.

In addition we provide normalised effort data for the four BGW versions to indicate the magnitude of maintenance effort needed, table 1. The effort for the development of the BGW framework versions are in the range 10 000 to 20 000 man-hours (the company requested too not publish the exact figures). The normalization is made with respect to the number of hours in version 1. That means, in a hypothetical example, if the development of version 1 took 10 000 hours the development of version 2 took 12 660 hours. The dura-

tion of the development and evolution of the BGW framework was approximatly six years.

Table 1. Normalized framework maintenance effort data

	V1	V2	V3	V4
Normalized effort	1,000	1,266	1,761	1,620

3.2 The Microsoft Foundation Class framework

The MFC framework addresses the domain of graphical user interfaces for Windows applications. The MFC framework has evolved through a number of versions and we have assessed the first five versions of the framework. Version 1.0 of the framework was available on the market 1992 and version 5 was released early 1997. Thus, the duration of the development and maintenance of the MFC framework is similar to the BGW case, around six years.

Through using the MFC framework the developer seldom has to access the Windows API (Application Program Interface) directly. Instead, one can use the MFC classes which provides services that wraps the functionality of the API. The framework comprises support for [19]:

- Window handling
- Interfaces for graphical devices
- File handling
- Exception handling
- Database handling

among other issues. For a more detailed description of the MFC framework we refer to, e.g. [19].

4. The Method

The stability assessment method is a metric-based approach for assessing the stability of an evolving framework. Recently, a method for quantitatively assessing stability of an object-oriented framework

was proposed by Bansiya [1]. In this study we have extended the method with an additional aggregated metric, the relative-extent-of-change metric. We have applied our method on a number of versions of two industrial object-oriented frameworks, four consecutive versions of the BGW framework and five consecutive versions of the MFC framework. The framework stability assessment method consists of the following four steps:

1. Identification of evolution characteristics
2. Selecting metrics to assess each evolution category
3. Data collection
4. Analysing the changed characteristics and computing the aggregated metrics

Based on the results of applying our method we propose a set of (preliminary) framework stability indicators, section 6.

4.1 Identification of evolution characteristics

When assessing framework stability, one has to select suitable indicators. The stability assessment method focus on measuring changes on object-oriented design components, i.e. the classes and in [1] two categories of evolution characteristics, *architectural* and *individual class* characteristics, are identified. First, the changes of architectural characteristics (that) are related to the *interaction* (collaborations) among classes and the *structuring* of classes in inheritance hierarchies.

Second, the characteristics of individual classes (that) are related to the assessment and change of the *structure*, *functionality* and *collaboration* (relationships) of individual classes between versions. The structure of objects is described and detailed by the data declarations in the class declarations. The functional characteristics are related to the object's methods. The parameter and data declarations that use user-defined objects define the collaboration characteristics of a class.

4.2 Selecting metrics to assess each evolution category

Several authors have proposed and studied metrics for object-oriented programming [2] [5][10] and, to some extent, for object-oriented frameworks [6]. In table 2, the architectural framework

Table 2. Architectural metrics

Architectural Metrics (Structure)	Architectural Metrics (Interaction)
DSC, design size in classes	ACIS, average number of public methods in all classes
NOH, number of hierarchies	ANA, average number of distinct (parent) classes from which a class inherits information
NSI, number of single inheritance instances	ADCC, average number of distinct classes that a class may collaborate with
NMI, number of multiple inheritance instances	
ADI, average depth of the class inheritance structure	
AWI, average width of the class inheritance structure	

structure and interaction metrics used in the method are described. The individual class metrics comprise in total of 11 metrics divided into three separate suits, functional, structural and relational, and are described in table 3. The metrics used are a subset of object-oriented metrics presented in [2][5].

Table 3. Class metrics

Structural metrics	Functional metrics	Relational metrics
NOD, the number of attributes.	NOM, count of all the methods defined.	NOA, the number of distinct classes (parents) which a class inherits.
NAD, the number of user defined objects used as attributes in a class.	NOP, the number of polymorphic methods in a class.	NOC, the number of immediate children (sub classes) of a class.
NRA, the number of pointers and references used as attributes.	NPT, the number of parameter types used in the methods of a class.	DCC, count of the number of classes that a class is directly related to.

Table 3. Class metrics

Structural metrics	Functional metrics	Relational metrics
CSB, the size of the objects in bytes from the class declaration.	NPM, average of the number of parameters per method in a class.	

4.3 Data collection

The metrics data was collected using the QMOOD++ tool [2]. The data has been collected for each version of the two frameworks with the same tool, thereby reducing the probability for different interpretations of the metric definitions. The data for the MFC framework is obtained from [1].

4.4 Analysing the changed characteristics and computing the aggregated metrics

For each of the frameworks and their versions, the collected architectural data are analysed and the two aggregated, normalised- and relative-extent-of-change, metrics are calculated. These two metrics are described in detail in section 5 since they are better explained in the context of the other metrics used.

5. Results and Analysis

We present the metric values from the assessment of the two cases, table 4, together with analysis of the obtained results. First, the architectural, normalised architectural and the associated aggregated metric values are presented and discussed for each of the two frameworks. Second, the class metric values with their associated aggregated metric values are presented and discussed for the frameworks.

5.1 The architectural metrics

The BGW Case. The architectural metric data collected for the four versions of the BGW framework shows that the number of classes

nearly doubles from 322 in version 1 to 598 in version 4. The values for number of hierarchies (NOH) and the number of single inheritance (NSI) metrics agree with the increasing value of number of classes. In addition, the values of average depth of inheritance (ADI) and average number of ancestors (ANA) are nearly the same due to the low number of multiple inheritance instances (NMI).

Table 4. Architectural metric values

	BGW				MFC					OWL			
Metric	1	2	3	4	1	2	3	4	5	4	4.5	5	5.2
DSC	322	445	535	598	72	92	132	206	233	82	142	357	356
NOH	30	35	40	44	1	1	1	5	6	12	16	29	27
NSI	265	371	434	480	66	84	117	174	194	34	56	202	198
NMI	5	1	1	1	0	0	0	0	0	9	13	30	30
ADI	1.97	2.24	2.09	2.14	1,68	2,11	2,45	2,33	2,30	0,78	0,81	1,42	1,41
AWI	0.73	0.75	0.74	0.73	0,92	0,91	0,89	0,85	0,83	0,52	0,49	0,65	0,64
ANA	2.00	2.25	2.09	2.14	1,68	2,11	2,45	2,33	2,30	1,00	0,97	1,56	1,55
ACIS	15,84	17,58	18,90	18,68	44,6	75,6	95	114	109	27,7	24	37,2	36,8
ADCC	3,61	4,07	3,99	3,98	6,03	7,59	9,02	9,33	8,94	1,87	2,26	4,73	4,71

The average depth of inheritance (ADI) reflects the level of specialisation in the framework and is expected to increase in newer versions and the average width of inheritance (AWI) is expected to decrease in new versions since new classes are generally added at deeper levels in the inheritance hierarchy. In the BGW study, we find a minor increase in the average depth of inheritance (ADI) values and a minor decrease of the average width of inheritance (AWI) values. Although this is as expected, the differences between the versions could have been larger. One possible explanation may be that this is a property of black-box and visual frameworks compared to white-box frameworks.

The average number of public methods (ACIS), 15-19 methods, and the average of directly coupled classes (ADCC), 3,6- 4,0 classes, are relatively constant throughout the versions.

The MFC Case. We see that MFC has had an increase of the number of classes more than three times, from 72 classes in version 1 to 233 classes in version 5. Regarding inheritance relationships MFC

does not make use of multiple inheritance and in version 4 the number of inheritance hierarchies was increased to five from only one in the previous versions. The values of average depth of inheritance (ADI) and average number of ancestors (ANA) is the same due to the lack of multiple inheritance in MFC. The average number of public methods (ACIS) has increased from 75,6 public methods up to 110-115 in the last two versions. The average number of directly coupled (ADCC) classes is relatively constant around 9,0 with an exception for the early version where the value was lower.

5.2 The normalised architectural metrics

In table 5 the normalised architectural metrics and the normalised-extent-of-change metric are presented. In the table the architectural metric values are normalised with respect to the metric's values in the first version of the framework. The metric values for the first version of each framework is used as a base for normalisation. The normalised metric values are computed by dividing the actual metric values of a version with the metric's value in the previous version for the actual framework. The normalised-extent-of-change metric is computed by summarising the normalised metrics for a framework version. The reason for the normalisation is that that we work with metrics whose values are of different ranges in the aggregated metrics. The normalisation is performed separately for each framework since they are developed by two independent organisations.

The aggregated metric has for the first version of the framework the same value as the number of architectural metrics (nine). Second, the normalised-extent-of-change metric is then computed by taking the difference of the aggregated metrics for the framework version, V_i, and the first framework version V_1. For the first version of a framework the normalised-extent-of-change metric is set to 0. For example, the normalised-extent-of-change metric for version 3 of the BGW framework is computed as 9,41 (aggregate metric values for version 3) minus 9 (aggregate metric value for version 1) which is 0,41. Thus, the normalised-extent-of-change metric has the value 0,41 for version 3 of the BGW framework.

The normalised-extent-of-change metric is a *relative indicator* of the framework's architectural stability. A high value of the metric indicates relative instability of the framework structure and a low

value of the normalised-extent-of-change metric indicates greater stability [1]. One way of viewing the normalised-extent-of-change metric is that it captures the enhancements of the framework, e.g. a high value indicates that the framework's architectural structure has changed significantly.

Table 5. Normalized architectural metrics values

	BGW				MFC					OWL	
Metric	1	2	3	4	1	2	3	4	5	4	5
DSC	1	1,38	1,20	1,12	1	1,28	1,43	1,56	1,13	1	4,35
NOH	1	1,17	1,14	1,10	1	1	1	5	1,2	1	2,42
NSI	1	1,40	1,17	1,11	1	1,27	1,39	1,49	1,11	1	5,94
NMI	1	0,2	1	1	1	1	1	1	1	1	3,33
ADI	1	1,14	0,93	1,02	1	1,26	1,16	0,95	0,99	1	1,82
AWI	1	1,03	0,97	0,99	1	0,99	0,98	0,96	0,98	1	1,25
ANA	1	1,12	0,93	1,02	1	1,26	1,16	0,95	0,99	1	1,56
ACIS	1	1,11	1,08	0,99	1	1,70	1,26	1,20	0,96	1	1,34
ADCC	1	1,13	0,98	1	1	1,26	1,19	1,03	0,96	1	2,53
Aggregate change (V_i)	**9**	**9,68**	**9,41**	**9,35**	**9**	**11,01**	**10,57**	**14,14**	**9,31**	**9**	**24,55**
Normalized-extent-of-change	**0**	**0,68**	**0,41**	**0,35**	**0**	**2,01**	**1,57**	**5,14**	**0,31**	**0**	**15,55**

The BGW Case. For the BGW framework the normalised values for number of hierarchies (NOH) is decreasing from 1,17 to 1,10. This indicates that the addition of new inheritance hierarchies is slowly decreasing with newer versions of the framework. Another observation is that the average number of directly coupled classes (ADCC) is fluctuating just over and below 1.0. This indicates that the "relationship complexity" is not increasing with newer versions, i.e. the number of relationships for a class is not increasing significantly in a new version of the framework. Regarding the normalised-extent-of-change metric; it is decreasing, starting from a low value, 0,68, which we interpret as the BGW framework is enhanced in relatively modest pace.

The MFC Case. The normalised value for the number of hierarchies (NOH) metric makes a jump up to 5 in version 4 due to the introduction of five separate hierarchies in the framework. The normalised values for the number of multiple inheritance hierarchies (NMI) is constantly 1 since there are no occurrences of multiple inheritance in the framework. In the MFC framework case one can make the same observation as in the BGW framework regarding the average number of directly coupled classes (ADCC) metric. It is close to 1, indicating that the "relationship complexity" is not increasing with newer versions of the framework. Regarding the normalised-extent-of-change metric; it is fluctuating up and down indicating that major enhancements and restructuring of the framework's architectural structure have occurred. One example is the introduction of four new inheritance hierarchies in version 4.

For both frameworks we see that for the normalised average number of public methods (ACIS) as well as the normalised average number of directly coupled classes (ADCC) metrics the values are steadily decreasing. A possible explanation is that the framework is becoming more stable and modifications can be realised more localised in the framework.

5.3 The class metrics

The individual class metrics defined in section 4 can be computed in two ways, i.e., including or excluding inherited properties. The first approach requires analysing a class and all its ancestors for the computation of the class metrics. Typically, changes made to the internal parts of parent classes in a framework ripple the effect of changes to all descendent of the parent class. Thus, in this study the class metrics values are counted including the inherited properties.

The metric for each of the 11 class characteristics can be changed between two successive versions. If the value of a particular metric (e.g. number of methods, NOM) changes from one version to the next, this is defined as *one Unit of Change*. Since there are 11 characteristics that can be changed for a class, a class can contribute with between 0 to 11 Units of Change. The 11 metric values of each class are compared with the metric values of the class in its predecessor version to compute the total number of units-changed for the classes. The *total-extent-of-change* for a framework version is the sum of

units-changed in all classes between a version and its predecessor. In table 6, the 11 class metrics obtained for the framework versions are presented.

Table 6. Changes in class characteristics

	BGW					MFC					
Metrics	V1/2	V2/3	V3/4	Total Units Changed	% Change	V1/2	V2/3	V3/4	V4/5	Total Units Changed	% Change
NOC	19	31	47	97	2.4	7	10	12	12	41	1.8
NOA	47	13	10	70	1.7	18	11	5	0	34	1.5
DCC	89	80	141	310	7.5	29	46	77	23	175	7.7
NOM	148	142	198	488	11.9	68	84	126	120	398	17.5
NOP	105	93	117	315	7.7	64	81	119	12	276	12.1
NPT	77	125	187	389	9.5	68	87	124	93	372	16.3
NPM	147	156	190	493	12.0	68	84	124	117	393	17.2
NOD	78	126	144	348	8.5	49	47	83	25	204	8.9
NAD	69	51	82	202	4.9	6	10	21	5	42	1.8
NRA	76	88	58	222	5.4	14	38	66	15	133	5.8
CSB	77	125	187	389	9.5	48	48	90	26	212	9.3
Totals	932	1 030	1 361	4 114	100.0	439	546	847	448	2280	100.0

To assess the significance of the total-extent-of-change measure we compare it with the maximum possible change. The maximum possible change represents the case where all the metrics would have changed values for all classes in a framework version. In table 7, the values for the *relative-extent-of-change metric* are presented. Relative-extent-of-change is computed as the *total-extent-of-change* divided with the maximum possible change.

The reason for introducing the relative-extent-of-change metric is two-fold. Firstly, the relative-extent-of-change metric is calculated based on class metrics, rather than system-wide metrics, which is the case for the normalised-extent-of-change metric. Thus, the relative-extent-of-change metric gives us a possibility to *validate* the original proposed normalised-extent-of-change metric. Secondly, the normalised-extent-of-change metric addresses the framework's architec-

tural structure, which implies that the metric is more, focused on capturing the enhancements, e.g. addition of features, of the framework. This in contrast to the relative-extent-of-change metric which analyses the *core* of the framework, i.e. it only measure changes in the set of classes that exist in two consecutive versions. That means that the metric captures how well the domain is captured and understood by the framework developers. The core of a framework version n consists of the classes that exist in two consecutive versions of the framework. A more formal definition is that the core is the set of classes that exist in both version n-1 and version n of the framework. That means that the core do not comprise the classes which existed in version n-1 and now not exist in version n of the framework. Neither are the classes added to framework version n part of the core. A consequence of this definition is that the core can grow with newer framework versions since it is defined in terms of two consecutive versions and not the first version of the framework. The set of added classes can be seen as representing a wider and better domain understanding captured by framework version n. This means that the core for framework version n+1 is expanded with the set of added classes in version n except for those classes that has been removed in version n+1 of the framework and is reduced with those classes in the core for version n which not exist in framework version n+1. Thus, a low relative-extent-of-change value indicates good domain coverage and understanding where a high relative-extent-of-change value indicates that the domain was not well understood and important abstractions have not been found.

The BGW Case. For the BGW framework the majority of the changes are related to methods and method parameters i.e. the functionality suit of metrics. For example, the number-of-methods (NOM) 11,9%, the average number-of-parameter types (NPT) 9,5% and average number-of-parameters per method (NPM), 12,0%. An exception is the number-of-polymorphic-methods (NOP) which only represent 7,5% of the total changes. A possible explanation is that the BGW framework is intentionally designed as a visual builder framework, thus favouring parameterisation before inheritance. Regarding the relative-extent-of-change metric, the values for the BGW framework are realatively low, between 21-26%, indicating good domain coverage and that there is a relatively stable core in the framework that remain rather unchanged.

The MFC Case. For the MFC framework, the same pattern occurs, the majority of changes belong to the functional suit of metrics. For example, the number-of-methods (NOM) is 17,5%, number-of-polymorphic-methods (NOP) 12,1% and the average number-of-parameters per method (NPM) metric 17,2%. Regarding the relative-extent-of-change metric values for the MFC framework it is high, over 50%, for the first four versions, indicating problems with understanding the domain and capturing useful abstractions.

In table 6, we see that the NOA (number-of-ancestor) metric changed values for 18 classes between version 1 and 2 of the MFC framework. The reason for this large change in the metric values was the restructuring of the MFC class hierarchy, which introduced several new classes in the middle of the hierarchy [1][19].

Table 7. Relative-extent-of-change metric values

	BGW				MFC				
	V1/2	V2/3	V3/4	Total	V1/2	V2/3	V3/4	V4/5	Total
Actual Units Changed	932	1030	1361	3323	439	546	847	448	2280
Max. units of Change	3542	4895	5885	14322	792	1012	1452	2266	5522
Relative-extent-of-change	*26,3%*	*21,0%*	*23,1%*	*23,2%*	*55,4%*	*54,0%*	*58,3%*	*19,8%*	*41,3%*

6. Observations

In this section we discuss a number of observations related to the results of applying the method, the method itself and the differences between the two frameworks which the method seems to be invariant to.

6.1 Observations on the results

By studying the results of applying the method on the two frameworks we have made a few general observations of the stability, or evolvability, of object-oriented frameworks which we present as a set of framework stability indicators.

Considering the values for the MFC and BGW frameworks, table 4, we see that the initial ADI values are increasing from values below 2.0 and then are successively increasing to values above 2.1. During the evolution of the frameworks, some versions have peak values for the ADI metric. For later versions, the ADI metric decreases to a level between 2.1 and 2.3. The AWI metric for the BGW framework is around 0,75 and decreases to 0.83 for the MFC framework. Thus, we formulate our framework stability indicator 1 that is fulfilled for the BGW and MFC frameworks as:

Framework Stability Indicator 1: Stable frameworks tend to have narrow and deeply inherited class hierarchy structures, characterised by high values for the average depth of inheritance (above 2.1) of classes and low values for the average width of inheritance hierarchies (below 0,85).

Considering the ratio between the number of subclasses and the total number of classes for a framework version we see that the ratio is going towards a limit value of 0.8 for the BGW and MFC frameworks, table 8. Thus, we have framework stability indicator 2:

Framework Stability Indicator 2: A stable framework has an NSI/DSC (number of single inheritance/design size in classes) ratio just above 0.8 if multiple inheritance is seldom used in the framework. That is, the number of subclasses in a stable framework is just above 80 percent.

Table 8. The NSI/DSC (Number of single inheritance/design size in classes) ratio

	BGW				MFC					OWL			
Metric	1	2	3	4	1	2	3	4	5	4	4.2	5	5.2
NSI	265	371	434	480	66	84	117	174	194	34	56	202	198
DSC	322	445	535	598	72	92	132	206	233	82	142	357	356
NSI/DSC	*0,82*	*0,83*	*0,81*	*0,80*	*0,92*	*0,91*	*0,89*	*0,84*	*0,83*	*0,41*	*0,39*	*0,57*	*0,56*

An analysis of the ADCC metric values, of the BGW and MFC frameworks, identifies a peak in version 2 and then it is decreasing slowly down to 1 or below in later versions, table 5. The fact that the ADCC values is close to 1 means that the relationship complexity is not increasing with newer versions nor is it decreasing, i.e. the number of relationships for a class is relatively stable and constant

throughout all versions of the framework. We formulate framework stability indicator 3 as:

Framework Stability Indicator 3: The normalised ADCC (averaged directly coupled class) metric is going towards 1.0 or just below for stable frameworks.

The MFC framework shows a normalised-extent-of-change value beginning at 2,01 and then successively decreasing down to 0,31 for version 5, table 5. For version 4 the normalised-extent-of-change value is 5,14 which is an atypical value for the MFC framework. The reason for this is the introduction of four new hierarchies in version 4 of MFC; otherwise the values would probably be around 1,5 or lower. In the case of the BGW framework, the normalised-extent-of-change metric decreases from 0,68 and ending with 0,35 for version 4. All versions having values less than 0,7. This results in framework stability indicator 4:

Framework Stability Indicator 4: The normalised-extent-of-change metric is below 0.4 for a stable framework.

The relative-extent-of-change metric is in one way similar to the normalised-extent-of-change metrics since it intends to capture the stability of the framework. On the other hand, the relative-extent-of-change metric is on another abstraction level since it is composed of a set of finer grained class metrics and is not directly measured on the architectural level of the framework. In table 7, the relative-extent-of-change metric values are presented. For example, version 1 of the BGW framework has 322 classes which gives a theoretical maximum of 322 * 11 (number of metrics that can change for a class) = 3 542 possible changes for the (previous) version. The actual number of changes between version 1 and version 2 of the BGW framework is 932 which represents a (932 / 3 542) * 100 = 26,3% change between version 1 and 2. Thus, the relative-extent-of-change metric is 26,3% for version 2 of the BGW framework.

The large BGW framework exhibits a relative-extent-of-change value just above 25% for the first transition, version 1 to version 2. For the successive transitions the value is below 25%. In the case of the MFC framework, it begins at a level above 55% for the first transition and continues to be on a level above 50% until version 5 when a large drop occurs down to 19,8%. This results in the formulation of framework stability indicator 5 as:

Framework Stability Indicator 5: A stable framework exhibits a relative-extent-of-change value of less than 25%.

In table 9, we see for the two frameworks for which versions the framework stability indicators are fulfilled. The five framework stability indicators are labelled, SI1-SI5, in the table. An ‘Y‘ indicate that the stability indicator is fulfilled and a ‘N’ that it is not fulfilled.

Table 9. Fulfilment of framework stability indicators

BGW	SI1	SI2	SI3	SI4	SI5	MFC	SI1	SI2	SI3	SI4	SI5
V1	N	Y	N	N	N	V1	N	N	N	N	N
V2	Y	Y	N	N	N	V2	N	N	N	N	N
V3	N	Y	Y	Y[a]	Y	V3	N	N	N	N	N
V4	Y	Y	Y	Y	Y	V4	Y	Y	Y	N	N
						V5	Y	Y	Y	Y	Y

a.The normalized-extent-of-change values was 0,41 for the version.

The BGW framework fulfils stability indicators 1 and 2 for version 2 and all but indicator 1 for version 3, see table 10. Version 4 of the BGW framework fulfils all the stability indicators. For the MFC framework the situation is that it fulfils indicators 1,2 and 3 for version 4. For version 5 of the MFC framework the stability indicators 4 and 5 are fulfilled too.

The reason why the BGW framework achieves stability in earlier versions than the MFC framework is probably that in the BGW case the supplier and the customers of the framework has a closer relationships, the customers are relatively few and sophisticated users/ domain experts. In the MFC case the customer is a mass-market with a large number of anonymous users of the framework.

The values used in the stability indicators may be discussed since they may be affected by organisational standards but our experiences from the application of the method on two frameworks from different organisations support that there is a degree of confidence in the values used.

6.2 Observations on the method

The differences with our assessment method compared to the original approach [1] are:

- The framework stability assessment method have been *applied* on two different frameworks, the BGW and MFC frameworks, where especially the BGW framework represents a more typical study subject since it represents a more common situation in software industry.
- An aggregated metric, the relative-extent-of-change metric based on individual class level metrics, which is on another abstraction level than the original normalised-extent-of-change metric has been added. The introduction of the relative-extent-of-change metric gives a possibility to *validate* the original proposed normalised-extent-of-change metric. In table 10, we see that the stability indicator based on the relative-extent-of-change metric, stability indicator 4, indicates framework stability for version 3 and 4 for the BGW framework and for version 5 for the MFC framework. The architectural metric, normalised-extent-of-change, used in stability indicator 5 indicates framework stability for version 3 and 4 for the BGW framework and for version 5 for the MFC framework. Thus, we see that the normalised-extent-of-change metric indicates stability in the same version as the relative-extent-of-change metric indicates it. One conclusion is that the normalised-extent-of-change metric is a good indicator of framework stability. The other aspect of relative-extent-of-change metric is that it gives us an indication of the stability of the core of the framework is, i.e. how well the domain is captured in the framework.
- As a result of applying the method a set of *experience-based framework stability indicators* has been added which make it possible to get a more objective way of deciding if a framework is stable or not.

The original approach has been validated through application of the assessment method twice and the method does not show any particular weakness. The same set of metric used in the original approach has been used. There also exist a possibility to change the

metric set in the assessment procedure, making the method more flexible, but in our study we have chosen to use the original set.

Both the BGW and MFC frameworks have reached stability according to the stability indicators. This shows one strength of the assessment method since it seems to cope with a number of framework dissimilarities.

6.3 Observations on the studied cases

We describe four major differences between the BGW and MFC framework that are invariant to the stability assessment method and thereby do not seem to affect the method.

The MFC framework is a commercial available white-box object-oriented framework with an anonymous market whereas the BGW framework is a proprietary framework. A proprietary owned framework seems to be a more typical study subject since it represents the more common situation in software industry and because the framework developing organisation has to deal with explicit customers and customer requirements rather than distributing the framework to a mass market.

Second, the BGW framework is a visual builder (black-box) framework, whereas the MFC framework is white-box. Our experience is that black-box frameworks generally contain a larger number of classes and inheritance hierarchies than white-box frameworks. The rationale for this is that the normal extension mechanism in white-box frameworks is subclassing through inheritance, whereas black-box (visual) frameworks use parameterisation. To support parameterisation, black-box frameworks generally include multiple “options”-hierarchies containing concrete subclasses that can be used in framework instantiations.

A third difference is the size, measured in number of classes, of the two frameworks. The original BGW framework (322 classes) was originally 4.5 times larger than first MFC framework (72 classes). Version 4 of the BGW framework (598 classes) is still 2.9 times larger than the fifth version of the MFC framework (233).

Finally, the domains covered by the frameworks are quite different. The MFC framework implements graphical user interface (GUI) functionality. Compared to most domains, the GUI domain is quite stable in the behaviour it is supposed to provide. The BGW domain is

considerable less stable and the developers of the framework have had to incorporate impressive amounts of requirement changes and additions during its lifetime.

7. Assessment

The method has been applied to two different frameworks, which makes it possible to better assess them with respect to the three management issues i.e. framework deployment, change impact analysis and benchmarking as well as a discussion about how difficult it is to apply the method, the need of tool support and experienced advantages and disadvantages of the methods. In the following we will discuss how the method perform with respect to the issues mentioned above.

The outcome of the stability assessment method gives an indication about how stable a framework version is. Especially the relative- and normalised-extent-of-change metrics provide information of how large the impact the requirements had on the framework structure. If historical trends for the current and other frameworks exist for the metrics, the metric values give a “rough” indication of maintenance effort needed. This with reservation for requirements that have architectural impact and thus affect the whole framework. Deviating metrics values from the values in the stability indicators are indications of evolution, which is not desirable and eventually has to be handled explicitly, if time schedule and budgets allow. Thus, the stability assessment method is useful for change impact analysis.

The framework deployment aspect is not directly addressed by the method. But if the method is applied more frequently during the iterative development of the framework version, the extent-of-change metrics provide information with respect to whether the framework is becoming more stable or not during the iterations. The other metrics in the stability assessment method also indicate stability or not depending if the values are deviating from the ones used in the stability indicators. The observed trends in the metric values give useful information for deciding whether to release the framework version or to iterate the development once again.

Finally, the issue of benchmarking is supported by the method since it collects a number of metrics and thereby provides empirical

information about the evolution of a framework. A historical set of data from previous assessments together with other information will make the prediction of the current framework's evolution more accurate.

The method as such is fairly simple to apply but it requires that the set of metrics used is clearly defined and consistently interpreted. Both between framework versions and different frameworks in order to provide comparable data. Tool support is highly recommended since a huge amount of data has to be dealt with. The tool used [2] in our study had problems with the large BGW framework. The problem was related to the number of file directories used in the framework and not the number of classes.

An advantage of the method is the possibility to change the underlying set of metrics. This is useful if the frameworks developed are adhering to a specific coding standard, a specific technical domain, e.g. distributed systems, or any other organisational issue. A minor disadvantage of the method is the calculation of the aggregated metrics; especially the class level metrics changes, if no tool support is provided.

The stability assessment method has a modest cost with it to apply but offers a number of benefits. The metric set used in the method provides a comprehensive view of the framework since it addresses five different framework stability aspects, the indicators. Another benefit is the possibility to change the metric set used in the method. It is also possible, as in our case, to select a metric set which provides information about inheritance structure changes.

8. Related Work

In this section we present related work to ours which addresses framework evolution and stability.

In [13] we present a method for identifying evolution-prone modules in a framework, which has been applied on the BGW framework. The method uses historical data related to the classes in the framework. Thus, the purpose of method is to identify those parts of the framework that changes most and do not adress the aspect of how the whole framework evolve.

The work by Roberts and Johnson [18] presents a pattern language that describes typical steps for the evolution of an object-oriented framework. Common steps in the evolution are development of a white-box framework (extensive use of the inheritance mechanism), the transition to a black-box framework (extensive use of composition), development of a library of pluggable objects etc. These steps give a coarse-grained description of a framework's evolution with respect to its technical maturity. However, they do not explicitly address and describe where and how the framework evolves when it has reached a certain state.

Assessments of frameworks have been addressed by Bansiya [1] and Erni and Lewerentz [6]. Bansiya's assessment approach is similar to ours since we have used it as a basis. One difference is that we have added an additional metric, which has been used for validating one of the original metrics. In addition, we have developed a set of framework stability indicators, which can be used for indicating framework stability. We have also applied the method on two industrial frameworks.

Erni and Lewerentz [6] describe a metrics-based approach that supports incremental development of a framework. By measuring a set of metrics (which requires full source code access) an assessment of the design quality of the framework can be performed. If the metric data is within acceptable limits, the framework is considered to be good enough from a metrics perspective otherwise another design iteration has to done. The approach is intended to be used during the design iterations, before a framework is released and does not consider long term evolution of a framework, as is the case with the Stability Assessment method.

9. Conclusion

Object-oriented framework technology has become common technology in object-oriented software development. An object-oriented framework is a reusable asset that constitutes the basis for a number of applications in the same domain. As with all software, frameworks tend to evolve. Once the framework has been deployed, new versions of a framework cause high maintenance cost for the products built with the framework. This fact in combination with the high costs of

developing and evolving an application framework make it important to have knowledge of the nature of framework evolution.

Reusable frameworks has to be considered as in-house products, thus requiring long term commitment for the maintenance and evolution of the framework. We have presented one method, stability assessment that assesses the stability of a framework version, which are useful in this organisational and long-term context. The method is metric-based and the set of metrics used is replaceable with other metric sets, which may be required due to specific domains and/or development processes.

Through applying the stability method on two different frameworks, both in the proprietary and commercial domain we have formulated five experience-based framework stability indicators which can be used for indicating framework stability:

- Stable frameworks tend to have narrow and deeply inherited class hierarchy structures, characterised by high values for the average depth of inheritance (above 2.1) of classes and low values for the average width of inheritance hierarchies (below 0,85).
- A stable framework has an NSI/DSC (Number of single inheritance classes/design size in classes) ratio just above 0.8 if multiple inheritance is seldom used in the framework. That is, the number of subclasses in a stable framework is just above 80 percent
- The normalised ADCC (averaged directly coupled class) metric is going towards 1.0 or just below for stable frameworks.
- The normalised-extent-of-change metric is below 0.4 for a stable framework.
- A stable framework exhibits a relative-extent-of-change value of less than 25%.

Through the extension and application of the method on two industrial frameworks we have also validated the method's applicability for assessing framework stability in an industrial context.

Currently, little knowledge about the stability and evolution of object-oriented frameworks is available. Through using the stability assessment method the obtained results will provide management

with information which will make it possible to make better and well-informed decisions about an object-oriented framework's evolution, especially with respect to change impact analysis, benchmarking and, to some extent, framework deployment.

Nevertheless, we believe that the framework stability assessment method and the framework stability hypotheses discussed in this paper provide a useful and industrial applicable approach to assessing the stability of object-oriented frameworks. However, we believe that some qualitative analysis is necessary to complement the metrics-based stability assessment. As part of future work, we intend to collect data from more object-oriented frameworks in order to validate our framework stability indicators.

References

[1] Bansiya, J. (1999) 'Evaluating structural and functional stability', in Fayad, M. E.; Schmidt, D. C.; and Johnson, R. E., (Eds.) *Object-Oriented Application Frameworks: Problems & Perspectives* , John Wiley & Sons Inc., New York NY, pp. 599-616.

[2] Bansiya, J. (1997) *A Hierarchical Model for Quality Assessment of Object-Oriented Design*, Doctoral Dissertation, Computer Science Department, University of Alabama in Huntsville, Huntsville AL, 214 pp.

[3] Bosch, J.; Molin, P.; Mattsson, M.; Bengtsson, P.; and Fayad, M. E. (1999) 'Framework problems and experiences', in Fayad, M. E.; Schmidt, D. C.; and Johnson, R. E., (Eds.) *Object-Oriented Application Frameworks: Problems & Perspectives* , John Wiley & Sons Inc., New York NY, pp. 55-82.

[4] Bosch, J. (1999) 'Measurement systems framework', in Fayad, M. E.; Schmidt, D. C.; and Johnson, R. E., (Eds.) *Domain-Specific Application Frameworks: Frameworks Experience by Industry*, John Wiley & Sons Inc., New York NY, pp. 117-206.

[5] Chidamber, S. R.; and Kemerer, C. F. (1994) 'A metrics suite for object-oriented design', *IEEE Transactions on Software Engineering*, 20(6), 476-493.

[6] Erni, K.; and Lewerentz, C. (1996) 'Applying design metrics to object-oriented frameworks', in *Proceedings of the 1996 3rd International Metrics Symposium*, IEEE Computer Society Press, Los Alamitos CA, pp. 64-74.

[7] Gamma, E; Helm, R.; Johnson, R.; and Vlissides, J. (1995) *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Reading MA, 395 pp.

[8] Geary, D. (1997) *Graphic Java 1.1: Mastering the AWT*, Sun Microsystems Press, Mountain View CA, 877 pp.

[9] Johnson, R.; and Russo, V. (1991) *Reusing Object-Oriented Design*, Technical Report UIUCDCS 91-1696, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign IL, 40 pp.

[10] Li, W.; and Henry, S. (1993) 'Object-oriented metrics that predict maintainability', *Journal of Systems and Software*, 23(2), 111-122.

[11] Mattsson, M. (1996) *'Object-Oriented Frameworks - A survey of methodological issues'*, Licentiate Thesis, LU-CS-TR: 96-167, Department of Computer Science, Lund University, Lund Sweden, 130 pp.

[12] Mattsson, M. (1999) 'Effort distribution in a six year industrial application framework project', in *Proceedings International Conference on Software Maintenance - 1999*, IEEE Computer Society Press, Los Alamitos CA, pp. 326-333.

[13] Mattsson, M.; and Bosch, J. (1999) 'Observations on the evolution of an industrial OO framework', in *Proceedings International Conference on Software Maintenance - 1999*, IEEE Computer Society Press, Los Alamitos CA, pp. 139-145.

[14] Mattsson, M.; and Bosch, J. (1999) 'Characterizing stability in evolving frameworks', in *Proceedings of the 29th International Conference on Technology of Object-Oriented Languages and Systems*, IEEE Computer Society Press, Los Alamitos CA, pp. 118-130.

[15] Molin, P.; and Ohlsson, L. (1996) 'The points and deviations pattern language of fire alarm systems', in *Proceedings International Conference on Pattern Languages for Program Design 2*, Addison Wesley Longman Inc., Reading MA, pp. 431-445.

[16] Moser, S.; and Nierstrasz, O. (1996) 'The effect of object-oriented frameworks on developer productivity', *IEEE Computer*, 29(9) 45-51.

[17] Prieto-Diaz, R.; Arango, G. (1991) *Domain analysis and software systems modeling*, IEEE Computer Press, Los Alamitos, pp. 9-88.

[18] Roberts, D.; and Johnson, R. (1996) 'Patterns for evolving frameworks', in *Proceedings International Conference on Pattern Languages of Program Design 3*, Addison Wesley Longman Inc., Reading MA, pp. 471-486.

[19] Schäfer, W.; Prieto-Diaz, R.; and Matsumoto, M. (1994) *Software Reusability*, Ellis-Horwood Ltd., New York NY, pp. 17-49.

[20] Shepherd, G.; and Wingo, S. (1994) *MFC Internals: Inside the Microsoft Foundation class architecture*, Addison-Wesley Publishing Co., Reading MA , 709 pp.

[21] Weinand, A.; Gamma, E.; and Marty, R. (1989) Design and implementation of ET++, a seamless object-oriented application framework, *Structured Programming*, 10(2), 63-87.

PAPER V

Effort Distribution in a Six Year Industrial Application Framework Project

Michael Mattsson

Published in the proceedings of ICSM'99, International Conference on Software Maintenance, Oxford, UK, 1999

Abstract

It has for a long time been claimed that object-oriented framework technology delivers reduced application development efforts but no quantitative evidence has been shown. In this study we present quantitative data from a six year object-oriented application framework project in the telecommunication domain. During six years four major versions of the application framework have been developed and over 30 installations at customer sites has been done. We present effort data both for the framework development and the installations as well as relative effort per phase for the framework versions. The effort data presented gives quantitative support for the claim that framework technology delivers reduced application development efforts. In fact, the effort data shows that the average application development effort is less than 2% of the framework development effort.

Keywords:
Object-oriented frameworks, framework evolution, framework development costs, framework customization costs

V

1. Introduction

Object-oriented framework technology has become common technology in object-oriented software development [1, 2, 3, 8]. Frameworks allow for large-scale reuse but it is both hard and expensive to develop object-oriented frameworks [4, 5, 11, 12].

Currently, little information of how costly and/or beneficial the development and use of framework technology is exists. Despite that a large number of organizations are developing and using object-oriented frameworks.

In this paper, we describe the effort distribution of an *evolving* object-oriented framework, the Mediation framework, which is a proprietary framework in the telecommunication domain. The effort data covers four consecutive versions of the framework that were developed over a period of six years. The effort needed for the development of a framework version is in the range 10 000 to 20 000 manhours. The size of the framework has increased from 322 classes to 598 classes during the same time.

Over 30 instantiations of the reusable framework has been delivered to customers and we present the effort needed, for each instantiation, in percentage of the framework development effort.

The contribution of this paper is quantitative support for the claim that object-oriented framework technology delivers reduced application development efforts as well as a description of the relative distribution of effort per phase in an evolving industrial framework project.

The paper is organized as follows: In section 2 the development process used, the notion of object-oriented frameworks as well as the Mediation framework are described briefly. Section 3 presents some metric values for the framework to give an understanding of the structural complexity of the software. Framework development effort data is presented in section 4 and in section 5 the framework customization effort data is described. Related work is found in section 6 and the paper is concluded in section 7.

2. Case Study Description

In this section we describe some background information for the study such as the development process used by the organization, the notion of object-oriented framework and its evolution stages, a description of the actual framework and what the customization of the studied framework comprises.

2.1 Development process

The organization's development process used for the framework development comprises the following activities:

Pre-study Phase. The purpose of the pre-study phase is to assess feasibility from technical and commercial viewpoints, based on the expressed and unexpressed requirements and needs. Activities comprise planning, securing competence, analysis of possible technologies available and an analysis of required effort.

Feasibility Phase. This phase comprises activities such as more detailed planning, definition of the project organization as well as communication/information paths. A risk analysis is performed and quality assurance techniques are defined for the project and implementation proposals are developed.

Design phase. Architectural design and detailed design is performed in this phase.

Implementation phase. The design is implemented and revised if necessary. Basic testing on the implemented classes is made. In the Mediation framework case the implementation was done in C++.

Test phase. Integration test and system test is performed here.

Industrialization. In this phase the developed software product is installed at a selected customer site. When the product is in operation and had been functioning correctly the system development is considered finished.

Documentation. This activity comprises the development and writing of user manuals system reference manuals and development of course material for the software product.

Administration. The administration activities comprise project management, quality management, configuration management and project meetings.

2.2 Object-oriented frameworks and their stages

Object-oriented frameworks are reusable designs of all or part of a software system. A framework is described by a set of abstract classes and the way instances (objects) of those classes collaborate [10]. The development cost of frameworks is high and it must be easy to learn. They must provide enough features to be used and hooks supporting features that are likely to change. Roberts and Johnson [10] describe the typical maturation path an object-oriented framework takes. We describe in the following, the four major stages.

White-box framework. A white-box framework is relying on the inheritance mechanism as the primary instantiation technique. The instantiations of a concept are easily made through creating a subclass of an appropriate abstract class. A disadvantage of white-box frameworks is that one has to know about internal class details and that the creation of new subclasses requires programming.

Black-box framework. A black-box framework differs from a white-box framework in that the primary instantiation technique is composition. Thus, a black-box framework allows one to combine the objects into applications and one does not need to know internal details of the objects. The use of composition to create applications implies that programming is avoided and it is possible to allow the compositions to vary at run time. Often the initial design of a framework is a white-box framework, whereas subsequent versions evolve into a black-box framework.

Visual Builder. To evolve to the Visual Builder stage the framework must be a black-box framework. A black-box framework makes it possible to develop an application by connecting objects of existing classes. An application consists of different parts. First, a script is used that connects the objects of the framework and then makes them active [10]. The other part is the behavior of the individual objects, which is provided by the black-box framework. Thus, framework instantiation in the black-box case is mainly a script that is similar for all applications based on the framework. One can now develop a separate graphical program, a Visual Builder, that makes it possible to specify the objects to be included in the application and the interconnection of them. The Visual Builder generates the code for the application from the specification. The addition of a Visual Builder to the framework provides a user-friendly graphical interface

which make it possible for domain experts to develop the applications by manipulating images on the screen.

Language Tools. The Visual Builder framework creates complex composite objects. These compositions have to be easily inspected and debugged. What one has in the Visual Builder framework is a graphical domain-specific programming language. Thus, one can create specific Language Tools for the framework that supports inspection and debugging of the relevant complex composite objects.

2.3 The Mediation framework

The Mediation framework provides functionality for mediation between network elements, i.e., telecommunication switches, and billing systems (or other administrative post-processing systems) for mobile telecommunication. Essentially, the Mediation framework collects call information from mobile switches (NEs), processes the call information and distributes it to a billing processing systems. Figure 1 presents the context of the mediation framework graphically. The driving quality requirements in this domain are reliability, availability, portability, efficiency (in the call information processing flow) and maintainability.

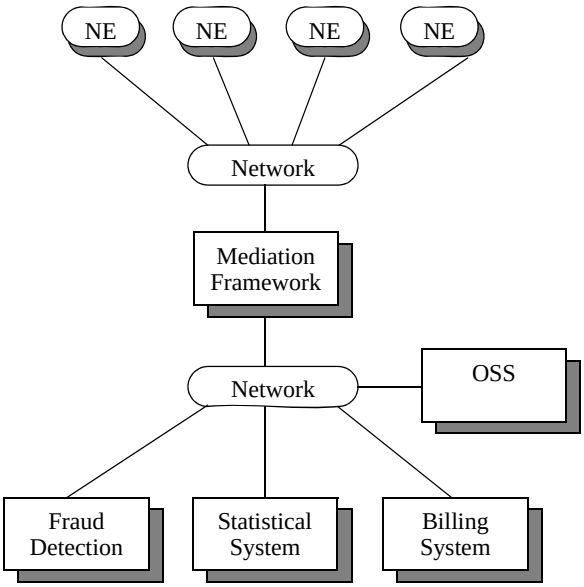

Figure 1. *The Mediation framework context.*

The Mediation framework has from version 1 been a Visual Builder framework, e.g. a black-box framework with an associated graphical customization interface and code generation functionality. From version 2 a few Mediation framework specific language tools have been developed.

2.4 Customization of the Mediation framework

The customization process of the Mediation framework is in this study an installation and customization process. This is due to the fact that the organization makes no distinction with respect to time reporting of the customization of the framework and the installation of the customized framework. Thus, the effort reported for customization of the Mediation framework comprises the following activities:

- Installation of necessary hardware at the customer site (is done in most of the cases).
- Installation of the Mediation software.

- Configuration of the Mediation framework for the actual customer. The customer can do configuration and re-configuration but most customers prefer the supplier to do the configuration.
- Acceptance Test.

The effort data reported in this study does not include time for travelling.

3. Product Metric Values

In this section we describe some product data (metrics) for the Mediation framework. These give an understanding of the structural complexity of the software. In table 1 we present data for all four versions of the following product metrics:

- DSC, Design size in classes
- NOH, Number of hierarchies
- NSI, Number of single inheritance classes
- NMI, Number of multiple inheritance classes
- ADI, Average depth of inheritance. The ADI metric is computed by dividing the sum of maximum path lengths to all classes by the number of classes. Path length to a class is the number of edges from the root to the class in an inheritance hierarchy.
- AWI, Average width of inheritance. The AWI metric is computed by dividing the sum of the number of children over all classes by the number of classes in the system..

Table 1. Product metrics values

Metric\Version	V1	V2	V3	V4
DSC	322	445	535	598
NOH	30	35	40	44
NSI	265	371	434	480
NMI	5	1	1	1

Table 1. Product metrics values

Metric\Version	V1	V2	V3	V4
ADI	1,97	2,24	2,09	2,14
AWI	0,73	0,75	0,74	0,73

We see that the size of the Mediation framework has increased from 322 to 598 classes, an increase of 86 percent. The increase of subclasses in the framework is 81 percent which is significantly more than the increase for the number of inheritance hierarchies which is 47 percent. The number of occurrences of multiple inheritance has dropped from five in version 1 down to only one. The characteristics of the inheritance hierarchies is described by the average depth of inheritance (ADI) metric and is around 2,1 for the later versions and the average width of inheritance (AWI) metrics is below 0,75 for all versions. For more information about the structural stability of the Mediation framework we refer to [6] and [7].

In table 2, we present the number of framework instantiations made from each version of the Mediation framework.

Table 2. Number of instantiations per version

Framework Version	V1	V2	V3	V4
Number of Instantiations	4	7	20	n/a

We see that for version 1 of the framework four instantiations were done, for version 2 seven instantiations and for version 3, 20 instantiations of the framework have been made at customer sites. For version 4 of the Mediation framework no instantiations have been reported yet since the development of version 4 was finished recently.

4. Framework Development Effort Data

In this section we describe the effort data for the four developed versions of the Mediation framework.

The effort data is collected from the company's internal time reporting system. Thus, the reliability of the obtained figures is dependent on the accuracy from the software engineers.

The cost measured in man-hours for the development of the Mediation framework versions are in the range 10 000 to 20 000 hours[1]. In table 3, the normalized effort data for the four versions are presented. The normalization is made with respect to the number of hours in version 1. That means, in a hypothetical example, if the development of version 1 took 10 000 hours the development of version 2 took 12 660 hours.

Table 3. Normalized framework development effort data

	V1	V2	V3	V4
Normalized effort	1,000	1,266	1,761	1,620

We see that the development cost for version 2 was approximately 25 percent higher than in version 1. For version 3 a major revision of the framework was made at a cost 76 percent higher than version 1. The development of version 4 is a little bit less expensive than version 3, 162 percent of the cost for version 1.

4.1 Relative effort distribution per phase

In table 4, the relative distribution of the effort between the different activities is presented. The distribution of effort is per developed version. The analysis and conclusions of the figures are based on interviews with the software engineers from the framework developments.

Table 4. Relative distribution of effort

Activity	V1	V2	V3	V4
Pre Study	7,6%	1,4%	5,4%	12,8%
Feasibility	0,0%	7,7%	6,3%	9,2%
Total	100,0	100,0	100,0	100,0

1.The company requested not to publish the exact figures

Table 4. Relative distribution of effort

Activity	V1	V2	V3	V4
Design	12,0%	7,9%	10,6%	12,7%
Implementation	29,3%	16,3%	23,8%	18,8%
Test	13,8%	16,9%	19,2%	21,6%
Administration	23,0%	38,9%	22,0%	17,6%
Industrialization	9,6%	3,9%	3,7%	2,5%
Documentation	4,7%	6,8%	8,9%	4,6%
Total	100,0	100,0	100,0	100,0

The figures in the table show that the relative effort for the pre-study and feasibility activities is increasing for later versions of the framework. The reason for this is that new requirements and proposed changes have to be analyzed more carefully. The reason for the figure 0,0 percent for feasibility version 1 is explained by the fact that the software engineers all had experiences from the application domain and the technologies to be used. The dip in design and implementation effort in version 2 exists because no major new functionality was introduced and a large amount of the effort was spent on restructuring the system. The increase in test effort is caused by the increased size of the framework from 233 to 598 classes. The reason for the peak in administration effort for version 2 is explained by a higher focus on quality processes. The organization realized during version 1 problems to manage both the development and customizations of the frameworks. Rösel reports the same kind of problems in [11]. To handle the change from project based development to more product based development new processes were introduced and existing ones improved. The high percentage of industrialization effort in version 1 compared to the other three versions were explained by the software engineers that the organization was for the first time fully responsible for the ownership of a software product.

If we do not include the administrative activities such as project and quality management, industrialization and documentation and combine the pre study and feasibility activities into an analysis activity and thereby focus on the more technical activities we achieve the

following distribution of effort, table 5 and figure 2. By only considering the more technical phases the figures may be more useful and thereby comparable for other organizations.

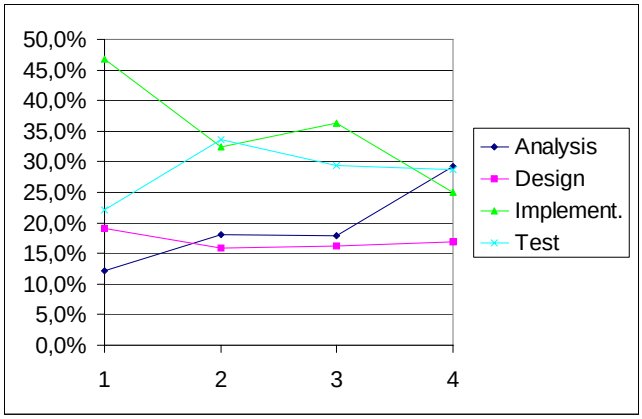

Figure 2. *Graph of relative effort distribution per version and technical phase.*

Table 5. Relative distribution of effort per technical phase and version

	V1	V2	V3	V4
Analysis	12,1%	18,1%	18,0%	29,3%
Design	19,1%	15,8%	16,2%	16,9%
Implementation	46,7%	32,5%	36,4%	25,1%
Test	22,0%	33,6%	29,4%	28,7%
Total	100,0	100,0	100,0	100,0

Figure 2 shows that during the evolution of the Mediation framework the emphasis has shifted from implementation to analysis. The implementation effort is in version 4 only 25 percent of the total effort for the technical activities compared to 50 percent for version 1 of the framework. The efforts spent on analysis increased, from 12 percent of the total effort, nearly three times, up to 30 percent of the total effort. The relative effort spent on design is just below 20 percent for all four versions. The test effort is around 30 percent with an exception for version 1 where it is lower. A possible explanation for

this could be that part of the testing has been performed during the implementation phase.

The shift of emphasis from implementation and test towards analysis is probably caused by the growing size of the software and that new requirements and proposed changes have to be analyzed carefully before being implemented.

5. Framework Customization Effort Data

In this section we present for each version the effort data for the instantiations of the framework in chronological order.

The figures in this section present the total effort spent on each customization in percentage of the total effort spent for the development of the framework version. For example if the development of a, hypothetical, version took 10 000 hours and the customization took 420 hours it will in this kind of figure have the value 4,2 percent. In addition we will for each version present the following basic statistics for the customizations: average, minimum, lower fourth, median, upper fourth, and maximum values.

5.1 The Mediation framework version 1

The major requirement for the development of version 1 of the framework was to come up with an architecture that supported the identified variable and stable parts of the domain. The organization could here rely on the experience from a previous developed application in the same domain.

For the first version of the Mediation framework four customizations have been made. In figure 3, we present the total effort spent on each customization in percentage of the total effort spent for the development of the framework version.

We see in figure 3 that the customization effort varies between 0,8 to 1,55 percent of the total framework development effort.

The basic statistics for the customization of version 1 is found in table 6. One observation we can make is that the total development effort for the four applications is the sum of the framework development effort, 1,0 and the total effort for the four customizations, 4*0,0106, in total 1,0424. This means that if we have the assumption

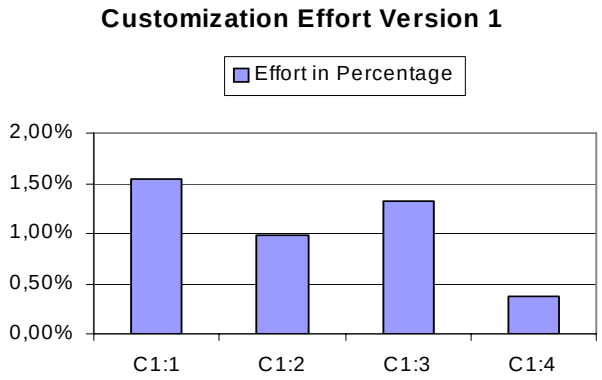

Figure 3. *Customization effort version 1.*

that we know that four applications should be developed the development effort of each application, in average, has to be less than 1,0424/4 = 0,2606. This if we had decided to not use the framework approach.

The most interesting observation is that all customization is below 1,6 percent of the framework development effort, which gives us a framework customization/development effort ratio at 1:62.

Table 6. Basic statistics version 1 customizations

Average	Min	Lower Fourth	Median	Upper Fourth	Max
1,06%	0,38 %	1,38%	1,16%	1,55%	1,55%

5.2 The Mediation framework version 2

For version 2 of the Mediation framework the development effort is 1,266, 27 percent higher than the development effort for version 1.

The major requirements for version 2 of the framework were increased flexibility and robustness. This was achieved through refinement of some of the design concepts, improved encapsulation, identification of minor domain specific libraries which provide generalized and abstract routines for data handling and formatting.

In figure 4 the effort needed for the seven customizations of version 2 of the framework is presented. We see that customization C2:7 is deviating from the other customizations with an effort figure at 6,89 percent of the framework development effort of version 2.

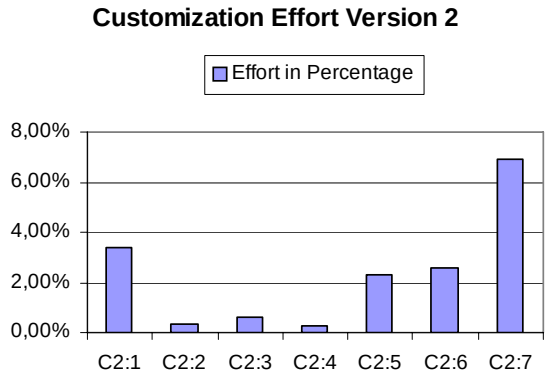

Figure 4. *Customization effort version 2.*

The basic statistics for the seven customizations are presented in table 7. The average effort needed for a customization is 2,34 percent of the effort needed for developing version 2 of the framework. We also see that the upper fourth for a customization is 3,00 percent of the framework development. This gives us a framework customization/development effort ratio at 1:33 for 75 percent of the customizations. The extreme customization, C2:7, results in a worst case framework customization/development effort ratio at 1:15.

Table 7. Basic statistics version 2 customizations

Average	Min	Lower Fourth	Median	Upper Fourth	Max
2,38%	0,30%	0,47%	2,28%	3,00%	6,89%

Using the same assumption as in the case for version 1 that we know that we will develop seven application in the framework domain we obtain the following results. Total effort for the development of the framework version and the seven applications is 1 + 7*0,0234 = 1,1638. Note that we should not use the value 1,266 for framework development effort since we are expressing the customi-

zation effort in percentage of the framework development effort. We should neither use the value 2,266, which is the sum of effort needed for the development of version 1 and version 2 since we are in a new decision situation when we should decide on the development of version 2.

Given that we know that we will develop seven applications the development effort in each application, in average, has to be less than 1,1638/7 = 0,166 if we decide to not use the framework approach.

5.3 The Mediation framework version 3

The development effort for version 3 of the Mediation framework is 1,620, that means 62 percent higher than the effort needed for developing version 1.

The major requirement for the development of version 3 was the introduction of hot billing. Hot billing makes it possible to collect call data records from network elements and distribute them to post processing systems within seconds of call completion. Thus, hot billing opens up for invoicing services tailored to specific customer needs, such as real-time billing and phone rental. This requirement caused major changes of the data processing in the system.

The effort distribution for the 20 customizations of version 3 of the Mediation framework is presented in figure 5 and the basic statistics is found in table 8.

Table 8. Basic statistics version 3 customizations

Average	Min	Lower Fourth	Median	Upper Fourth	Max
1,62%	0,12%	0,60%	1,24%	2,01%	6,28%

We see that the majority of the customizations are in the interval 0,5-2,5 percent of the effort needed for the development of the framework version. The average customization is 1,62 percent and the upper fourth 2,01 percent of the framework development effort. For the upper fourth this gives us a framework customization/development effort ratio at 1:49. The worst case gives a framework customization/development effort ratio 1:15 since the maximum effort

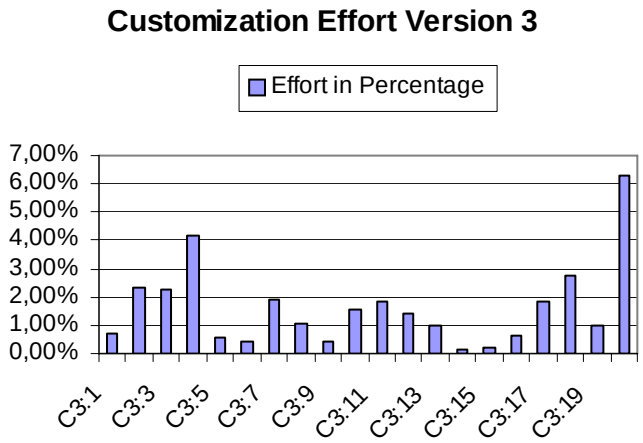

Figure 5. *Customization effort version 3.*

required for a customization is 6,28 percent of the development effort.

5.4 The Mediation framework version 4

The development effort for version 4 of the framework is 62 percent higher that for version 1. For version 4 of the framework major requirements was portability to new operating platforms and advanced data processing such as call data record matching and rating. Matching makes is simple to collect data from different nodes and times and thereby simplifying the charging of services with more than one call data record such as distance related charging.

Currently, no effort data is available for customizations of version 4 since the development was finished recently.

5.5 Summary

If we summarize the framework customization/development effort ratios for the three versions that has been customized, table 9, we see

Table 9. Basic statistics framework customization/development effort ratios

Version	Average	Lower Fourth	Median	Upper Fourth
1	1:94	1:72	1:86	1:64
2	1:42	1:214	1:44	1:33
3	1:61	1:167	1:81	1:49
Total	1:58	1:168	1:75	1:44

that the median effort ratio varies between 1:44 to 1:86 for the different versions. The upper fourth ratio varies between 1:33 to 1:64. The last row in table 9 presents the framework customization/development ratios for all 31 customizations. The average framework customization/development ratio is 1:58 which meant that half of the customizations made required an effort less than 1,8 percent of the framework development effort. The upper fourth customization/ development ratio is 1:44, which means that 75 percent of the customization made required an effort less than 2,3 percent of the total development effort.

6. Related Work

To the best of our knowledge we are only aware of two references that discuss economic aspects of object-oriented framework development and customization.

Moser and Nierstrasz [9] discuss the productivity of the framework user. Their study compares the function point approach with their own System Meter approach for estimating framework customization productivity. The idea behind the System Meter approach is to distinguish between the internal and external sizes of objects, and thus to measure only complexity that has a real effect on the size of the system to be implemented, i.e. incremental functionality. Their

study reports that the effect of framework reuse does not increase the software engineer's productivity using the System Meter approach. Compared to our study they do not discuss the costs and distribution of costs for framework development and customization but only the productivity aspect.

Rösel [11] is the only reference that presents some quantitative statements about framework development and reuse. Rösel's experience is that the cost of a framework may be 50 percent of the total cost for the first three applications. Once these three or more applications are implemented and being used the framework will most likely have paid for itself. He also remarks that to reach this pay-off level significant commitment and investment are required. Compared to our study Rösel gives no explicit quantitative support for his claim and the topic is just briefly discussed.

7. Conclusion

We have described the effort distribution in a six year industrial application framework project from the telecommunication domain (the Mediation framework). Effort data from four versions of the framework has been collected and the relative distribution of effort per version has been presented.

The effort needed for the development of a framework version is in the interval 10 000 to 20 000 man hours. The effort distribution per phase and version shows a clear shift of emphasis from implementation in the early versions to analysis in the latter versions of the framework.

We have also presented effort data for over 30 customizations (application developments) of the Mediation framework. The effort data shows that object-oriented framework technology delivers reduced application development effort.

The effort customization data collected shows that the effort of developing an application based on the Mediation framework is, in average, less than 2 percent of the effort needed for developing the framework version. In fact, 75 percent of the framework customizations requires an effort less than 2,5 percent of the framework development effort.

The main contribution of this paper is that it gives quantitative support for the well-known claim that object-oriented framework technology delivers reduced application development effort.

References

[1] G. Andert, 'Object Frameworks in the Taligent OS,' *Proceedings of Compcon 94*, IEEE CS Press, Los Alamitos, California, 1994.

[2] J. Bosch, P. Molin, M. Mattsson, PO Bengtsson, M. Fayad,'Object-Oriented Frameworks - Problems & Experiences', In *Object Oriented Foundations of Framework Design*, M. E. Fayad, D. C. Schmidt, R. E. Johnson (eds.), Wiley & Sons, 1999, forthcoming.

[3] H. Huni, R. Johnson, R. Engel, 'A Framework for Network Protocol Software,' *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages and Applications*, Austin, USA, 1995.

[4] R.E. Johnson, V.F. Russo, 'Reusing Object-Oriented Design,' *Technical Report UIUCDCS 91-1696*, University of Illinois, 1991.

[5] M. Mattsson, 'Object-Oriented Frameworks - A survey of methodological issues,' *Licentiate Thesis*, LU-CS-TR: 96-167, Department of Computer Science, Lund University, 1996.

[6] M. Mattsson, J. Bosch, ' Characterizing Stability in Evolving Frameworks', *Proceedings of the 30th International Conference in Technology of Object-Oriented Languages and Systems*, TOOLS '99 Europe, Nancy, France, 1999

[7] M. Mattsson, J. Bosch, 'Observations on the Evolution of an Industrial OO Framework', *Proceedings of the International Conference on Software Maintenance - 1999*, Oxford, England, 1999.

[8] Peter Molin and Lennart Ohlsson, 'Points & Deviations -A pattern language for fire alarm systems,' *Proceedings of the 3rd International Conference on Pattern Languages for Programming*, Paper 6.6, Monticello IL, USA, September 1996.

[9] S. Moser, O. Nierstrasz, 'The Effect of Object-Oriented Frameworks on Developer Productivity,' *IEEE Computer*, pp. 45-51, September 1996.

[10] D. Roberts, R. Johnson, 'Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks,' *Proceedings of the Third Conference on Pattern Languages and Programming*, Montecillio, Illinois, 1996.

[11] A. Rösel, 'Experiences with the Evolution of an Application Family Architecture', *Proceedings of the Second International ESPRIT ARES Workshop, Las Palmas, Spain, pp. 39-48*

[12] S. Sparks, K. Benner, C. Faris, 'Managing Object-Oriented Framework Reuse,' *IEEE Computer,* pp. 53-61, September 1996.

PAPER VI

Assessment of Three Evaluation Methods for Object-Oriented Framework Evolution

Michael Mattsson and Jan Bosch

Research report 1999:20, Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby, Sweden, also submitted to the Journal Information and Software Technology, 1999

Abstract

Object-oriented framework technology has become a common reuse technology in object-oriented software development. As with all software, frameworks tend to evolve. Once the framework has been deployed, new versions of a framework cause high maintenance cost for the products built with the framework. This fact in combination with the high costs of developing and evolving an object-oriented framework make it important to have controlled and predictable evolution of the framework's functionality and costs. We present three methods 1) Evolution Identification Using Historical Information, 2) Stability Assessment and 3) Distribution of Development Effort which have been applied to between one to three different frameworks, both in the proprietary and commercial domain. The methods provide management with information which will make it possible to make well-informed decisions about the framework's evolution, especially with respect to the following issues; identification of evolution-prone modules, framework deployment, change impact analysis, benchmarking and requirements management. Finally, the methods are compared to each other with respect to costs and benefits.

VI

Keywords:
Software reuse, Object-oriented framework, Framework evolution, Framework assessments

1. Introduction

Object-oriented framework technology has developed into more common technology in object-oriented software development [5, 9, 22, 25], because it bears the promise of large-scale reuse and thereby reduced development effort, e.g. [23]. An object-oriented application framework is a reusable asset that constitutes the basis for a number of applications in the domain captured by the framework. Common framework examples from the domain of graphical user interfaces are Java AWT [12], Microsoft Foundation Classes [27] and ET++ [11]. Application frameworks from other domains such as fire alarm systems and measurement systems are found in [22] and [6], respectively.

As with all software, frameworks tend to evolve, leading to new versions, due to the incorporation of new or changed requirements, better domain understanding, experiences and fault corrections. Once a framework has been deployed, new versions of the framework will cause maintenance costs for the products built with the framework. This, in combination with the high costs of developing and evolving an application framework, indicate the importance of understanding and characterizing framework evolution.

One of the main objectives for management of software development organisations is to have controlled and predictable evolution of the framework's functionality and costs (both the development and framework instantiation costs). This requires that there must exist methods assisting and providing management with information about the framework and its evolution. Management issues of interest are:

- Identification of evolution-prone modules: Management may decide to proactively maintain the framework to simplify the incorporation of future requirements. In that case, it is important to know which modules exhibit a high degree of change between framework versions and which modules do not? The evolution-prone modules are likely candidates for restructuring. The information about where likely changes occur may

reduce the cost of redesign the framework. Studies show that software engineers without automated support can only identify a subset (<50 percent) of future changes and they can not provide the complete view of change [16]. Thus, a more objective method for identifying structural shortcomings in an object-oriented framework will provide more reliable input for maintenance effort estimations and thereby increase the accuracy.

- Framework deployment: Assessment of the framework to decide when it can be released for regular product development in the organisation or placed on the commercial market. The assessment is necessary to avoid the shipping of frameworks that do not fulfil their functional and quality requirements and, consequently, will cause unnecessary maintenance and release costs. The assessment data can in some cases, if publicly available, be used by potential customers for evaluating and selecting a framework from several alternatives. For instance, if the assessment covers aspects such as reliability and robustness for its users.
- Change impact analysis: A third type of framework analysis is concerned with assessing the impact of changes both on the next framework version and the applications built using the current version. The assessment results can be used by management to predict required maintenance effort for both the framework and the applications incorporating instantiations of the frameworks.
- Benchmarking: Empirical information collected from the assessment of successful frameworks can be valuable in guiding the development and evolution of new frameworks and can be used to develop criteria for framework classification, e.g. benchmarking, in terms of structural stability or other framework maturity aspects [2]. This is desirable since it is hard and expensive to develop and maintain object-oriented frameworks and empirical data is valuable input to the cost and effort estimation activities. Currently, little information is available concerning the cost and/or benefits associated with the development and use of framework technology. Empirical information about the distribution of effort in framework devel-

opment, evolution and instantiation will help in the process of allocating specialists and other personnel to the developments as well as in effort estimations.

- Requirements management: Framework assessment can be used to select the criteria for deciding when to incorporate a new requirement during framework instantiation or not in the next version of the framework. In addition, it supports decisions on the incorporation of new features in the framework, since the impact is made explicit. A primary apparent strategy for an organisation is to incorporate the requirements for the different instantiations into the next version of the framework. However, some of the instantiations could have been difficult and expensive to perform and are stretching the limits of the framework and should not have impact on the next framework version.

In this paper we present three methods which identify and characterise framework evolution as well as an assessment of the methods against the aforementioned issues. In addition, the following aspects will be discussed about the methods: difficulty to apply, the need of tool support and experienced advantages and disadvantages of the methods. The three methods are:

- Evolution Identification Using Historical Information. This method is used for identifying and characterizing the evolution-proness of the application framework, its subsystem and their modules. Based on historical information obtained from the framework versions, it is possible to make statements about the framework's evolution in terms of size, growth rate and change rate. The statements about the evolution of the framework, its subsystems or subsystem modules can be made from one version to the next or by considering a set of successive versions. Thus, the purpose the method is to identify the evolution-prone parts of the application framework.
- Stability Assessment. The stability assessment method is a metrics-based assessment method for assessing structural and behavioural stability of a framework. The notion of structural and behavioural stability is viewed as the dual property of evolution. Through collection of a set of architectural and class

level metrics together with the calculation of some aggregated metrics, the obtained metric values are used for the assessment of the framework together with a set of framework stability indicators. Each framework stability indicator is used for deciding if the current framework version is stable or not with respect to the metrics dealt with by the indicator. Thus, the purpose of the method is to decide how structural and behavioural stable a framework is in terms of framework stability indicator fulfilment.

- Distribution of Development Effort. The method is based on an analysis of the effort consumed during the development of the framework. Effort data is collected per developed framework version and development phase. Based on this data, a calculation of effort per phase and version in relative terms of the total effort consumed per framework version is done. The result is presented and a qualitative analysis of what caused the changes in relative effort consumption for the framework versions is done. Normally, the application of the method is simple since effort data often is collected and stored in the developing organisation's time reporting system. The main purpose of the method is to characterise and analyse the relative distribution of effort per development phase and version for a framework and the relation between framework development effort and framework instantiation effort.

The three methods have been applied on four consecutive versions of the same framework, the Billing Gateway (BGW) framework (see section 2), a proprietary object-oriented application framework in the telecommunication domain which makes it possible to assess them with respect to the aforementioned issues. The application and detailed results of the evolution identification using historical information method can be found in [19]. The results of applying the stability assessment method can be found in [20] and [21]. In these studies the Microsoft Foundation Classes (MFC) [27] and Borland's Object Windows Library (OWL) [24] frameworks have been assessed too. Both MFC and OWL are addressing the graphical user interface domain. The results from applying the distribution of development effort method on the BGW framework can be found in [18].

In this paper we are focusing on assessing the three methods with respect to their suitability for the requirements that we discussed earlier. Consequently, we provide only some major results from the application of the methods.

The main contribution of this paper is that it presents a qualitative assessment of three methods for evaluating object-oriented framework evolution, which all have been applied on industrial frameworks, with respect to the identified management issues; i.e. identification of evolution-prone modules, framework deployment, change impact analysis, benchmarking, and requirements management.

The remainder of the paper is organised as follows. Framework terminology and the Billing Gateway (BGW) application framework is presented in section 2. The three methods and some results from the application of the methods, as well as a discussion about each method, are described in section 3. In section 4 the assessment of the methods and the comparison is presented. Related work is found in section 5 and the paper is concluded in section 6.

2. Object-Oriented Frameworks

Object-oriented frameworks are reusable designs of all or part of software systems. A framework is defined by a set of abstract classes and the way instances (objects) of those classes collaborate [25]. The development cost of frameworks is high and it must be easy to learn. They must provide enough features to be used and hooks supporting features that are likely to change. Roberts and Johnson [25] describe the typical maturation path an object-oriented framework takes. We describe in the following, the four major stages.

White-box framework. A white-box framework is relying on the inheritance mechanism as the primary instantiation technique. The instantiations of a concept are easily made through creating a subclass of an appropriate abstract class. A disadvantage of white-box frameworks is that one has to know about internal class details and that the creation of new subclasses requires programming.

Black-box framework. A black-box framework differs from a white-box framework in that the primary instantiation technique is composition. Thus, a black-box framework allows one to combine

the objects into applications and one does not need to know internal details of the objects. The use of composition to create applications implies that programming is avoided and it is possible to allow the compositions to vary at run time. Often the initial design of a framework is a white-box framework, whereas subsequent versions evolve into a black-box framework.

Visual builder. To evolve to the visual builder stage the framework must be a black-box framework. A black-box framework makes it possible to develop an application by connecting objects of existing classes. An application consists of different parts. First, a script is used that connects the objects of the framework and then activates them [25]. The other part is the behaviour of the individual objects, which is provided by the black-box framework. Thus, framework instantiation in the black-box case is mainly a script that is similar for all applications based on the framework. Based on this, one can develop a separate graphical program, i.e. a visual builder, that allows the framework user to specify the objects to be included in the application as well as their interconnection. The visual builder generates the code for the application from the specification. The addition of a visual builder to the framework provides a user-friendly graphical interface which makes it possible for domain experts to develop the applications by manipulating images on the screen.

Language tools. A visual builder framework creates complex composite objects. These compositions have to be easily inspected and debugged. A visual builder framework basically defines a graphical domain-specific programming language. Thus, one can create specific language tools for the framework that support inspection and debugging of the relevant complex composite objects.

2.1 The framework cases

In this section we describe the Billing GateWay framework, which is the framework which has been used as a common study subject for the three methods. The MFC and OWL frameworks are not described since they capture a rather well-known domain, graphical user interfaces. For a more detailed description of these two frameworks we refer to [27] and [24], respectively.

The BGW framework is a major part of the Billing Gateway product developed by Ericsson Software Technology, and provides func-

tionality for billing data mediation between network elements, i.e., switches, billing systems, or other post-processing systems for mobile telecommunication. The framework collects call information from mobile switches (NEs), processes the call information and mediates it to billing processing systems. Figure 1 presents the context of the BGW framework graphically. The driving quality requirements in this domain are reliability, availability, portability, call records throughput and maintainability. The framework is a visual builder framework (black-box plus a graphical interface), i.e., the framework provides a number of pre-defined objects for application configuration.

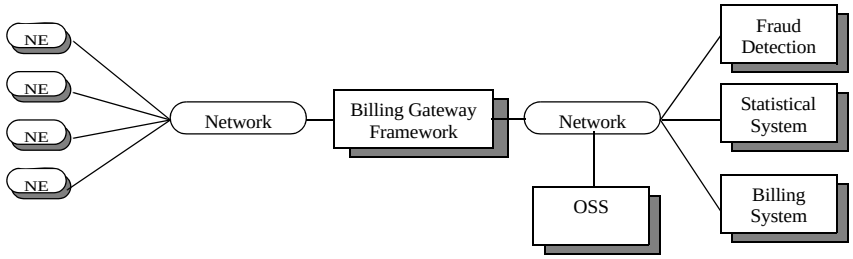

Figure 1. *The Billing Gateway application framework context.*

We have studied four consecutive versions of the BGW framework and the BGW framework has from the first version, been at the visual builder stage, e.g. a black-box framework with an associated graphical instantiation interface and code generation functionality. Successively, from the second version of the BGW framework, a few specific language tools have been developed.

The BGW framework is instantiated through a graphical configuration window since it is a visual builder framework. The application developer (framework user) select different types of entities and associated operations to be performed which should constitute the instantiated application. For example, it is possible to select NEs (e.g. a mobile switch) and PPSs (post processing systems) and different kinds of processing such as filtering and formatting. Each of these entities has during the configuration/instantiation to be specified in more detail, through the use of entity-specific menus. When all entities have been specified, the application to be developed based on the framework is specified, and it is possible to generate the application.

The specification data is divided to appropriate classes and objects and the application is generated

To provide an understanding of the evolution of BGW framework we describe the major requirements for the four framework versions since they have impact on the structure and functionality of the framework.

Version 1. The major requirement for the development of version 1 of the framework was to develop an architecture that supported the identified variable and stable parts of the domain. The organisation could here rely on the experience from a previous developed application in the same domain.

Version 2. The major requirements for version 2 of the framework were increased flexibility and robustness. This was achieved through refinement of some of the design concepts, improve encapsulation and the identification of minor domain specific libraries which provide generalised and abstract routines for data handling and formatting.

Version 3. The major requirement for the development of version 3 was the introduction of hot billing. Hot billing makes it possible to collect call data records from network elements and distributes them to post processing systems within seconds of call completion. Thus, hot billing opens up for invoicing services tailored to specific customer needs, such as real-time billing and phone rental. This requirement caused major changes of the data processing in the system.

Version 4. For version 4 of the BGW framework major requirements was portability to new operating platforms and advanced data processing such as call data record matching and rating. Matching makes it simple to collect data from different nodes and times and thereby simplifying the charging of services with more than one call data record such as distance related charging.

3. The Methods and Major Results

In this section we present the three methods for evaluating object-oriented framework evolution together with the major results achieved when applying the methods on the BGW framework. For a more detailed presentation of the results of applying the method we refer to, [19] for the evolution identification using historical information

method, [20] and [21] for the stability assessment method and [18] for the distribution of development effort method.

3.1 Evolution identification using historical information

Method description

The evolution identification method for identifying and characterizing the evolution-proness of an object-oriented framework, its subsystem and the modules is an adaptation of an approach proposed in [10]. The focus for the study in [10] was on macro-level software evolution by tracking the release history of a system. Their system comprises about 10 million LOC and was structurally divided into a number of subsystems where each subsystem was further divided into modules and each module consisted of a number of programs. Based on historical information about the programs, subsystems etc. the authors investigated the change rate, growth rate and size of the software (and its parts) using the *version numbering* of the programs as the units for 20 releases of the software systems.

We have adapted the approach to better suit for smaller, object-oriented, systems (200-600 classes). We decompose our object-oriented framework into subsystems and modules. Our modules are composed of classes, not programs as in [10]. This means that calculations of change and growth rates are made in terms of classes as the units. Changed classes are identified by a *change in the number of public operations* in the class interface. The rationale for this is that a change in the public interface represents an addition or better understanding of the functionality that the class has to offer the rest of the system. Changes to the private part of the class interface are mostly related to perfective changes such as performance improvements and, are as such less interesting in this study. We have extracted information, about the following subset of framework properties:

- The *size* of each framework, subsystem or module is defined as the number of classes it contains. We use the notion of classes as the size counting entity instead of measuring the size of the source code. This since it is easier to collect necessary information and measure on the detailed design level compared to the source code level.

- The *change rate* is the percentage of classes in a particular framework, subsystem or module that changed from one version to the next. To compute the change rate, two versions are required for comparison. The relative number of the changed classes represents the change rate.
- The *growth rate* is defined as the percentage of classes in a particular framework, subsystem or module, which have been added (or deleted) from one version to the next. To compute the growth rate, two versions are compared and the numbers of the added and deleted classes are computed. The relative number of the new classes (i.e. the difference between added and removed classes) represent the growth rate.

Based on structural information extracted from the different framework versions it is possible to make statements about a framework's evolution related to

1. the complete framework system,
2. the subsystems of the framework; and
3. the modules of particular subsystems that we chose for further study.

For example, the size of the framework and its subsystems and the number of classes changed in a particular module from one version to the next. The method steps are

1. calculate, for all releases, the change and growth rate for the complete framework,
2. calculate, for all releases, the change and growth rate for each of the subsystems,
3. for those subsystems that exhibit high growth and change rates calculate, for all releases, the change and growth rates for the modules,
4. those modules that exhibit high change and growth rates are identified as likely candidates for restructuring.

Major results

In this section we present results from the application of the method on the BGW framework. The results describe observations on the framework, subsystem and module level of the framework.

Framework level observations. Figure 2 shows the number of classes that were added in each version (i.e. added minus removed classes). For example, the figure shows that at version 2.0 more than 120 classes were added to the system. A general observation is that the *number of added classes is decreasing* with newer versions of the framework. This can be taken as an indication that the framework is becoming more mature, i.e. achieving a structural stability.

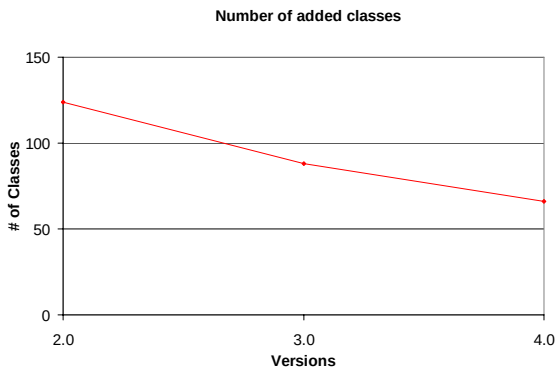

Figure 2. *Number of added classes in each version.*

Figure 3 compares the change and growth rates for each version of the BGW framework. In general, the change rate is higher than the growth rate. The growth rate is decreasing with successive versions of the framework, from about 40 percent at version 2.0 down to about 15 percent at version 4.0. This *decrease in growth rate* is another indication of the framework's structural stability.

The change rate is increasing at version 4.0 and is not following the trend for version 2.0 and 3.0. One reason for this may be a change with *architectural impact*, i.e. a change affecting entities throughout the whole system. In the BGW framework the architectural change was the introduction of hot billing. This requirement caused a conversion from file-based data representation (containing several megabytes) to block-based data representation (containing several tens of

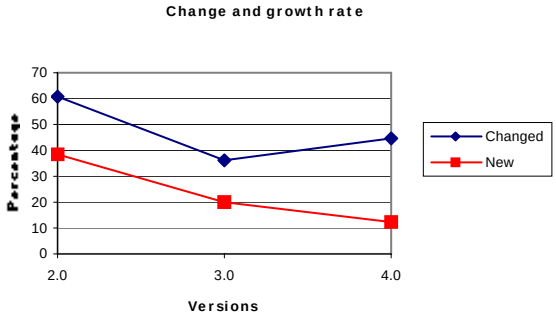

Figure 3. *Comparison of the change and growth rates.*

kilobytes). This change had architectural impact and caused changes in many parts of the system.

The change rates are around 40-50 percent in between all versions. In version 2.0 and 3.0 this may be a sign of better domain understanding but in version 4.0 the major reason is the architectural change introduced by the hot billing requirement.

The observations on the framework levels indicate that the framework evolves in a satisfactory fashion. This since the growth rate is decreasing and the change rate is at an acceptable level, less than 50 percent. However, the observations are only valid on system level and it may be the case that the subsystems do not behave in the same way.

The evolution of subsystems. Up to now we have achieved an impression of how the overall framework has evolved over time but are the subsystems evolving the same way as the whole system?

Table 1. Change and growth rates of subsystems

Subsystem	Change rate	Growth rate	Relative Size
A	51%	191%	34%
B	64%	153%	24%
C	92%	191%	37%
D	75%	700%	5%

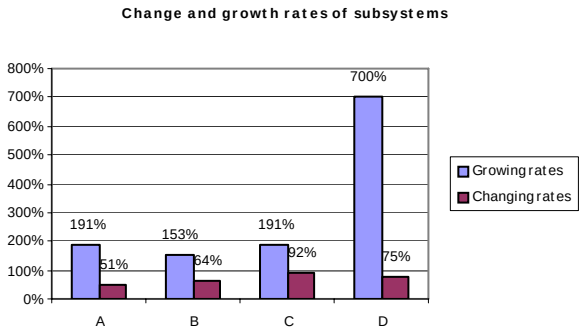

Figure 4. *Change and growth rates of subsystems.*

In table 1, we see the change and growth rates for the four subsystems in the BGW framework. Subsystems A and C have a growth rate on 191 percent each. Their relative sizes of the whole system are 34 percent for subsystem A and 37 percent for subsystem C, respectively. Thus, both are candidates for further study but since Subsystem A represent the GUI part of the system and consists, to a very high degree, of automatic generated UI classes it is not a representative subsystem. Subsystem D exhibits the highest growth rate, 700 percent, but since subsystem D only comprises 5 percent of the total system size in version 4.0 the subsystem is of minor interest.

If we consider the change rate it is obvious that subsystem C has undergone major changes between version 1.0 to version 4.0 since it has a change rate of 92 percent. It may also be of interest to investigate subsystem B since its change rate, 64 percent, is higher than the change rates for the whole system. Subsystem C represents a major part of the system, 37 percent of the classes, and we will investigate it further on the module level.

Based on the observations regarding growth and change rates on the subsystem level, we will further investigate the evolution characteristics for Subsystem C due to its high growth and change rates. Subsystem B is also a candidate due to its relatively high change rate, but we refer to [19] for a discussion of Subsystem B.

The evolution of subsystem C. Subsystem C exhibits the highest growth and change rates among the subsystems. These facts make the subsystem a likely candidate for restructuring and/or re-engineer-

ing activities. We will investigate how the modules in subsystem C evolve and whether they possess the same evolution characteristics as the whole subsystem.

Subsystem C consists of 5 modules, C1-C5. Figure 5, which shows that module C3 and C4 are large modules and that they both have had a large increase of the number of added classes. In addition modules C1, C2 and C5 has been relatively stable in size but a minor increase can be observed for module C1.

Regarding the growth and change rates for the modules in subsystem C; these are presented in Figure 6. Module C4 has a high growth rate, 306 percent, and it represents 25 percent of the subsystem. Through discussions with the software engineers it was clarified that the reason for the high growth rate is that the module is responsible for coding and decoding the Call Data Records which exist in a number of formats. Thus, the high growth rate is the result of an increasing number of formats supported by the BGW framework, e.g. module C4 handles a number of different Call Data Record formats which results in a set of classes whose objects are used as parameters in the framework. I.e. a typical black-box (and visual builder) framework evolution observation.

Module C5 has the highest growth rate, 460 percent, but is the smallest module in the subsystem. All modules except Module C5 exhibit high change rates. A reason for the relative low change rate for Module C5 is that it captures the encapsulation of some software packages

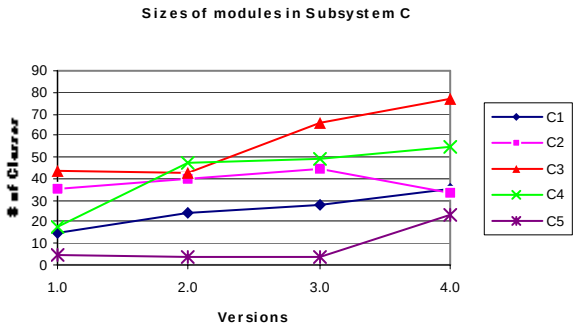

Figure 5. *Sizes of modules in Subsystem C.*

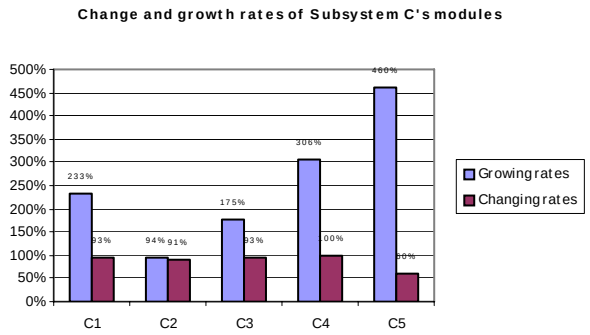

Figure 6. *Change and growth rates of Subsystem C's modules.*

We see that the change rates for the modules in Subsystem C behave as the system in whole, i.e. the change rate is increasing from version 3.0 to version 4.0, figure 7. Modules C3 and C4 exhibit the highest change rates throughout all the versions.

Concerning the high change rate, all modules in subsystem C have high change rates and the main reason for this is two-fold;

1. a design concept called Node has been divided into two concepts, Childnode and ParentNode, to achieve higher maintainability and lower complexity,

2. another design concept called Point has changed from being file-oriented to be transaction-oriented due to the need of provide decreased latency for the Call Data Record (the billing information input) processing. This was required to fulfil the requirement of hot billing.

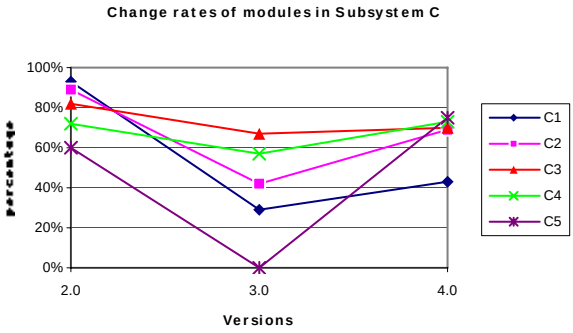

Figure 7. *Change rates of modules in subsystem C.*

To summarize the findings of Subsystem C we can foresee changes in all modules due to the changes of the two design concepts. These changes have presumably not achieved full effect on the system design. Regarding Module C4 and its growth rate; this is viewed as an anticipated kind of evolution due to the nature of black-box frameworks.

Discussion

Our observations from using the method are similar to the study in [10], i.e. there is a difference in the behaviour of the whole system versus its subsystems. A relatively stable evolution over time was observed at the framework system level, but detailed studies of the subsystems showed that subsystem C and B exhibit different characteristics regarding the change and growth rates when compared to the whole system. These major differences would not been identified only by an analysis of the whole system. Two modules were identified as potential subjects for redesign, i.e. module C4 and B2 (not discussed, see [19]). In addition, to a lesser extent, module C3 is a likely candidate for redesign. We have not discussed subsystem B in this paper and refer to [19] for a more detailed discussion.

3.2 Stability assessment

Method description

Recently, a method for quantitatively assessing stability of an object-oriented framework was proposed by Bansiya [2]. In this study we have extended the proposed method by adding an additional metric, the *relative-extent-of-change* metric as well as a set of framework stability indicators. The method consists of the following five steps.

1. Identification of evolution characteristics.
2. Selecting metrics to assess each evolution category.
3. Data collection.
4. Analysing the changed architectural characteristics and computing the extent-of-change metrics in the architecture between framework versions.
5. Comparison against the framework stability indicators.

Identification of evolution characteristics. When assessing framework stability, one has to select suitable indicators. Bansiya identifies two categories of evolution characteristics, *architectural* and *individual class* characteristics. First, the changes of architectural characteristics are related to the *interaction* (collaborations) among classes and the *structuring* of classes in inheritance hierarchies. Second, the characteristics of individual classes are related to the assessment and change of the *structure*, *functionality* and *collaboration* (relationships) of individual classes between versions. The structure of objects is described and detailed by the data declarations in the class declarations. The functional characteristics are related to the object's methods. The parameter and data declarations that use user-defined objects define the collaboration characteristics of a class.

Selecting metrics to assess each evolution category. Bansiya selected a subset of object-oriented metrics from [4, 7] and we propose to use the same set of metrics but it is possible to change the metrics to others that could be more appropriate with respect to the organisation's processes and application domains. In table 2, the architectural framework structure and interaction metrics used are described. The individual class metrics comprise in total of 11 metrics divided into three separate suits and are described in table 3.

Table 2. Architectural metrics

Architectural Metrics (Structure)	Architectural Metrics (Interaction)
DSC, design size in classes	ACIS, average number of public methods in all classes

Table 2. Architectural metrics

Architectural Metrics (Structure)	Architectural Metrics (Interaction)
NOH, number of hierarchies	ANA, average number of distinct (parent) classes from which a class inherits information
NSI, number of single inheritance instances	ADCC, average number of distinct classes that a class may collaborate with
NMI, number of multiple inheritance instances	
ADI, average depth of the class inheritance structure	
AWI, average width of the class inheritance structure	

Data collection. We recommend to collect the data using the same tool for reducing the probability for different interpretations of the metric definitions. In our studies, the data has been collected for each version of the studied frameworks with the same tool, the QMOOD++ tool [4].

Table 3. Class metrics

Structural metrics	Functional metrics	Relational metrics
NOD, the number of attributes.	NOM, count of all the methods defined.	NOA, the number of distinct classes (parents) which a class inherits.
NAD, the number of user defined objects used as attributes in a class.	NOP, the number of polymorphic methods in a class.	NOC, the number of immediate children (sub classes) of a class.
NRA, the number of pointers and references used as attributes.	NPT, the number of parameter types used in the methods of a class.	DCC, count of the number of classes that a class is directly related to.
CSB, the size of the objects in bytes from the class declaration.	NPM, average of the number of parameters per method in a class.	

Analysing the changed architectural characteristics and computing the extent-of-change metrics in the architecture between framework versions. For each of the frameworks and their versions, the collected architectural data are analysed and the normalized- and relative-extent-of-change metrics are calculated.

The *normalized-extent-of-change-metric* is calculated in the following way. The architectural metric values are normalized with respect to the metric's values in the previous version of the framework. The metric values for the first version of each framework is used as a base for normalization. The normalized metric values are computed by dividing the actual metric values of a version with the metric's value in the previous version for the actual framework. A *temporary aggregated* metric is then computed by summarizing the normalized metrics for a framework version.

The normalized-extent-of-change metric has for the first version of the framework the same value as the number of architectural metrics (in our case nine). Second, the normalized-extent-of-change metric is then computed by taking the difference of the aggregate metrics for the framework version, V_i, and the first framework version V_1. For the first version of a framework the normalized-extent-of-change metric is set to 0. For example in the case of the BGW framework, the normalized-extent-of-change metric for version 3 of the BGW framework is computed as 9,41 (aggregate metric values for version

3) minus 9 (aggregate metric value for version 1) which is 0,41. Thus, the normalized-extent-of-change metric has the value 0,41 for version 3 of the BGW framework.

The normalized-extent-of-change metric is a relative indicator of the framework's architectural stability. A high value of the metric indicates relative instability of the framework structure and a low value of the normalized-extent-of-change metric indicates greater stability.

The relative-extent-of-change metric. The individual class metrics defined can be computed in two ways, i.e., including or excluding inherited properties. The first approach requires analysing a class and all its ancestors for the computation of the class metrics. Typically, changes made to the internal parts of parent classes in a framework ripple the effect of changes to all descendent of the parent class. Thus, in this study the class metrics values are counted including the inherited properties. The metric for each of the 11 class characteristics can be changed between two successive versions. If the value of a particular metric (e.g. number of methods, NOM) changes from one version to the next, this is defined as *one Unit of Change*. Since there are 11 characteristics that can be changed for a class, a class can contribute with between 0 to 11 Units of Change. The 11 metric values of each class are compared with the metric values of the class in its predecessor version to compute the total number of units-changed for the classes. The *total-extent-of-change* for a framework version is the sum of units-changed in all classes between a version and its predecessor. To assess the significance of the total-extent-of-change measure we compare it with the maximum possible change. The maximum possible change represents the case where all the metrics would have changed values for all classes in a framework version. The relative-extent-of-change metric is computed as the *total-extent-of-change* divided with the maximum possible change. For example, version 1 of the BGW framework has 322 classes which gives a theoretical maximum of 322 * 11 (number of metrics that can change for a class) = 3 542 possible changes for the (previous) version. The actual number of changes between version 1 and version 2 of the BGW framework is 932 which represents a (932 / 3 542) * 100 = 26,3% change between version 1 and 2. Thus, the relative-extent-of-change metric is 26,3% for version 2 of the BGW framework.

The relative-extent-of-change metric is in one way similar to the normalized-extent-of-change metrics since it intends to capture the stability of the framework. On the other side, the relative-extent-of-change metric is on another abstraction level since it is composed of a set of finer grained class metrics and is not directly measured on the architectural level of the framework. Thus, the relative-extent-of-change metric gives us a possibility to validate the original proposed normalised-extent-of-change metric. Another important difference is that the normalised-extent-of-change metric addresses the framework's architectural structure, i.e. it is more, focused on capturing the enhancements, e.g. addition of features, of the framework. This compared to the relative-extent-of-change metric which analyses the *core* of the framework, i.e. it only measure changes in the set of classes that exist in two consecutive versions. That means that the metric better describes how well the domain is captured and understood by the framework developers. The core of a framework version *n* consists of the classes that exist in two consecutive versions of the framework. A more formal definition is that the core is the set of classes that exist in both version *n-1* and version *n* of the framework. That means that the core do not comprise the classes which existed in version *n-1* and now not exist in version *n* of the framework. Neither are the classes added to framework version *n* part of the core. A consequence of this definition is that the core can grow with newer framework versions since it is defined in terms of two consecutive versions and not the first version of the framework. The set of added classes can be seen as representing a wider and better domain understanding captured by framework version *n*. This means that the core for framework version *n+1* is expanded with the set of added classes in version *n* except for those classes that has been removed in version *n+1* of the framework and is reduced with those classes in the core for version *n* which not exist in framework version *n+1*. Thus, a low relative-extent-of-change value indicates good domain coverage and understanding where a high relative-extent-of-change value indicates that the domain was not well understood and important abstractions have not been found.

Comparison against the framework stability indicators. The framework stability indicators are based on experiences from applying the

original stability assessment method on three object-oriented frameworks [20, 21].

The five stability indicators are:

- *Framework Stability Indicator 1:* Stable frameworks tend to have narrow and deeply inherited class hierarchy structures, characterized by high values for the average depth of inheritance (above 2.1) of classes and low values for the average width of inheritance hierarchies (below 0,85).
- *Framework Stability Indicator 2:* A stable framework has an NSI/DSC ratio just above 0.8 if multiple inheritance is seldom used in the framework. That is, the number of subclasses in a stable framework is just above 80 percent.
- *Framework Stability Indicator 3:* The normalized ADCC (averaged directly coupled class) metrics is going towards 1.0 or just below for stable frameworks. I.e. the number of relationships for a class is relatively stable and constant throughout all versions of the framework.
- *Framework Stability Indicator 4:* The normalized-extent-of-change metric is below 0.4 for a stable framework.
- *Framework Stability Indicator 5:* A stable framework exhibits a relative-extent-of-change value of less than 25%.

A framework is considered to be more stable when more framework stability indicators are fulfilled for a framework version.

Major results

We present framework stability indicator fulfillments for the BGW and MFC frameworks. For a more detailed presentation of the metric results for the BGW, MFC and OWL frameworks we refer to [20, 21].

In table 4, the five framework stability indicators, I1-I5, and the versions for the three frameworks are listed. An ‘Y‘ indicate that the framework stability indicator is fulfilled and a ‘N’ that it is not fulfilled.

The BGW framework fulfils framework stability indicator 1 and 2 for version 2 and all except indicator 1 for version 3, table 4. Version 4 of the BGW framework fulfils all the indicators. For the MFC framework the situation is that it fulfils framework stability indica-

tors 1,2 and 3 for version 4. For version 5 of the MFC framework, the additional indicators 4 and 5 are fulfilled too.

To summarize, the BGW framework fulfils framework stability indicators to a large extent from version 3 and the MFC framework from version 4.

Table 4. Framework Stability Indicators fulfillments

BGW	I1	I2	I3	I4	I5	MFC	I1	I2	I3	I4	I5
V1	N	Y	N	N	N	V1	N	N	N	N	N
V2	Y	Y	N	N	N	V2	N	N	N	N	N
V3	N	Y	Y	Y	Y	V3	N	N	N	N	N
V4	Y	Y	Y	Y	Y	V4	Y	Y	Y	N	N
						V5	Y	Y	Y	Y	Y

Discussion

The differences with our assessment method compared to the original approach [2] are:

- A set of framework stability indicators have been added which make it possible to get a more objective way of deciding if a framework is stable or not.
- An aggregated metric, the relative-extent-of-change metric based on individual class level metrics, which is on another abstraction level than the original normalized-extent-of-change metric has been added. The introduction of the relative-extent-of-change metric gives a possibility to validate the normalized-extent-of-change metric with it. The results when applying the method showed that the framework stability indicator based on the relative-extent-of-change metric, indicator 4, indicates framework stability for version 3 and 4 for the BGW framework and for version 5 for the MFC framework. The architectural metric, normalized-extent-of-change, used in framework stability indicator 5 indicates framework stability for version 3 and 4 for the BGW framework and for version 5 for the MFC framework. Thus, we see that the normalized-extent-of-change metric indicates stability in the same version as the relative-

extent-of-change metric indicates it. The conclusion is that the normalized-extent-of-change metric is a good indicator of framework stability.

- The framework stability assessment method have been applied on three different frameworks, the BGW, MFC and OWL frameworks [20, 21].

The original approach have (indirectly) been validated through the application of the (extended) assessment method on the three frameworks and the method does not show any particular weakness. The same set of metric used in the original approach has been used. There exists a possibility to change the metric set in the assessment procedure but in our studies we have chosen to use the original set.

Both the BGW and MFC frameworks have reached stability according to the framework stability indicators. This shows one strength of the assessment method since it seems to cope with a number of framework dissimilarities. We identify four major differences between the BGW and MFC framework that are invariant to the stability assessment method.

- The MFC framework is a commercial available white-box object-oriented framework with an anonymous market whereas the BGW framework is a proprietary framework. A proprietary owned framework seems to be a more typical study subject since it represents the more common situation in software industry and because the framework developing organization has to deal with explicit customers and customer requirements rather than distributing the framework to a mass market.
- The BGW framework is a visual builder (black-box) framework, whereas the MFC framework is white-box. Our experience is that black-box frameworks generally contain a larger number of classes and inheritance hierarchies than white-box frameworks.
- The size of the framework, measured in number of classes. The original BGW framework (322 classes) was originally 4.5 times larger than first MFC framework (72 classes). Version 4 of the BGW framework (598 classes) is still 2.9 times larger than the fifth version of the MFC framework (233 classes).

- The domains covered by the frameworks are quite different. The MFC framework implements graphical user interface (GUI) functionality. Compared to most domains, the GUI domain is quite stable in the behaviour it is supposed to provide. The BGW domain is considerable less stable and the developers of the framework have had to incorporate impressive amounts of requirement changes and additions during its lifetime.

3.3 Distribution of Development Effort

Method description

The distribution of development effort method is essentially an analysis of the effort data for the various development phases and activities for developing a framework. The distribution of development effort method comprises the following activities

- Assure that the phases and activities for the framework development and instantiation are clearly defined so there is no doubt about what is to be included and not.
- Collect the effort data from the framework development projects.
- Calculate the relative distribution of the framework development efforts and the relative effort for the instantiations.
- Perform a qualitative analysis of the data to identify trends in the development, causes for the trends and similarities/dissimilarities with previous framework projects.

Major results

We describe the development process used by the organization, that developed the BGW framework as well as the activities that are included in the framework instantiation process. The organization's development process used for the framework development comprises the following phases and activities:

- *Pre-Study.* The purpose of the pre-study phase is to assess feasibility from technical and commercial viewpoints, based on the expressed and unexpressed requirements and needs. Activities comprise planning, securing competence, analysis of possible technologies available and an analysis of required effort.
- *Feasibility.* This phase comprises activities such as more detailed planning, the project organization is defined as well as communication/information paths. A risk analysis is performed and quality assurance techniques are defined for the project and implementation proposals are developed.
- *Design*. Architectural design and detailed design is performed in this phase.
- *Implementation*. The design is implemented and revised if necessary. Basic testing on the implemented classes is made. In the BGW framework case, the implementation was done in C++.
- *Test*. Integration test and system test is performed here.
- *Industrialization*. In this phase the developed software product is installed at a selected customer site. When the product is in operation and had been functioning correctly the system development is considered finished.
- *Documentation*. This activity comprises the development and writing of user manuals, system reference manuals and development of course material for the software product.
- *Administration*. The administration activities comprise project management, quality management, configuration management and project meetings.

The *instantiation process* of the BGW framework is, in this study, an installation and instantiation process. This due to the fact that the organization makes no distinction with respect to time reporting between the instantiation of the framework and the installation of the instantiated framework. Thus, the effort reported for instantiation of the BGW framework comprises the following activities:

- Installation of necessary hardware at the customer site (done in most of the cases).

- Installation of the BGW software.
- Configuration of the BGW framework for the actual customer. The customer can do the configuration themselves but most customers prefer the supplier to do the configuration.
- Acceptance Test.

Development effort data. The effort for the development of the BGW framework versions are in the range 10 000 to 20 000 man hours[1]. In table 5, the normalized effort data for the four versions are presented. The normalization is made with respect to the number of hours in version 1. That means, in a hypothetical example, if the development of version 1 took 10 000 hours the development of version 2 took 12 660 hours.

Table 5. Normalized Framework Development Effort Data

	V1	V2	V3	V4
Normalized effort	1,000	1,266	1,761	1,620

We see that the development cost for version 2 was approximately 25 percent higher than in version 1. For version 3 a major revision of the framework was made at a cost 76 percent higher than version 1. The development of version 4 is a little bit less expensive than version 3, i.e. 162 percent of the cost for version 1.

The effort data is collected from the organisation's internal time reporting system. Thus, the reliability of the obtained figures is dependent on the accuracy from the software engineers. The effort data reported does not include time for travelling.

Relative effort distribution per phase and version. In table 6, the relative distribution of the effort between the different activities is presented. The distribution of effort is per developed version. The analysis and conclusions of the figures are based on interviews with the software engineers involved in the development of the framework versions.

1.The company requested not to publish the exact figures

Table 6. Relative distribution of effort per version and phase

Activity	V1	V2	V3	V4
Pre Study	7,6%	1,4%	5,4%	12,8%
Feasibility	0,0%	7,7%	6,3%	9,2%
Design	12,0%	7,9%	10,6%	12,7%
Implementation	29,3%	16,3%	23,8%	18,8%
Test	13,8%	16,9%	19,2%	21,6%
Administration	23,0%	38,9%	22,0%	17,6%
Industrialization	9,6%	3,9%	3,7%	2,5%
Documentation	4,7%	6,8%	8,9%	4,6%
Total	100,0%	100,0%	100,0%	100,0%

The figures in table 6 show that the relative effort for the pre-study and feasibility activities is increasing for later versions of the framework. The major reason for this is that new requirements and proposed changes have to be analysed more carefully. The figure 0,0 percent for feasibility version 1 is explained by the fact that the software engineers all had experiences from the application domain and the technologies to be used. The dip in design and implementation effort in version 2 is because no major new functionality was introduced and a large amount of the effort was spent on restructuring the system. The increase in test effort is caused by the increased size of the framework from 233 to 598 classes. The reason for the peak in administration effort for version 2 is explained by a higher focus on quality processes. During version 1 the organization experienced problems to manage both the development and instantiations of the framework. The same kind of problems is reported in [26]. To handle the change from project based development to more product-based development, new processes were introduced and existing ones improved. The high percentage of industrialization effort in version 1 compared to the other three versions were explained by the fact that the software engineers were fully responsible for the ownership of a software product for the first time.

If we do not include the administrative activities such as project and quality management, industrialization and documentation and

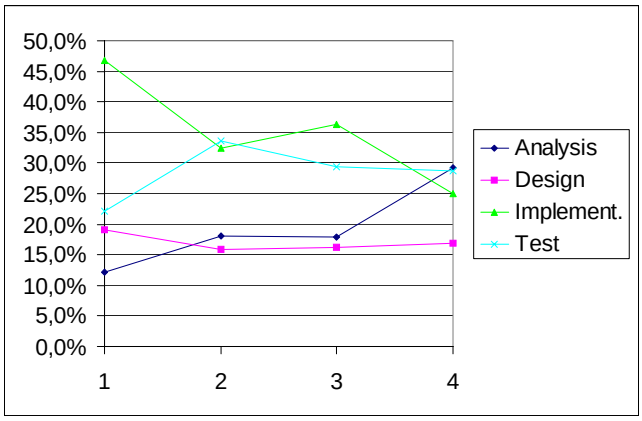

Figure 8. *Graph of relative effort distribution per version and technical*

combine the pre-study and feasibility activities into an analysis activity and thereby focus on the more technical activities we achieve the following distribution of effort, table 7 and figure 8. By only considering the more technical phases the figures may be more useful and thereby comparable for other organizations.

Table 7. Relative distribution of effort per technical phase and version

	V1	V2	V3	V4
Analysis	12,1%	18,1%	18,0%	29,3%
Design	19,1%	15,8%	16,2%	16,9%
Implementation	46,7%	32,5%	36,4%	25,1%
Test	22,0%	33,6%	29,4%	28,7%
Total	100,0%	100,0%	100,0%	100,0%

Figure 8 shows that during the evolution of the BGW framework the emphasis has shifted from implementation to analysis. The implementation effort is in version 4 only 25 percent of the total effort for the technical activities compare to 50 percent for version 1 of the framework. The efforts spend on analysis increased, from 12 percent of the total effort, nearly three times, up to 30 percent of the total effort. The relative effort spent on design is just below 20 percent for all four versions. The test effort is around 30 percent with an exception for version 1 where it is lower. A possible explanation for

this could be that part of the testing has been performed during the implementation phase.

The shift of emphasis from implementation and test towards analysis is probably caused by the growing size of the software and that new requirements and proposed changes has to be analysed carefully before being implemented.

Instantiation effort data

In table 8 we present basic statistics for framework instantiation effort relative framework development effort. For example, version 1 of the BGW framework has been instantiated 4 times and the average instantiation/development effort ratio is 1:94, meaning that the effort needed for instantiating the “average application” is only 1,06% of the effort needed for developing the framework versions. In the table,

Table 8. Basic Statistics framework instantiation/development effort ratios

Version	# Instantiations	Average	Lower Fourth	Median	Upper Fourth
1	4	1:94	1:72	1:86	1:64
2	7	1:42	1:214	1:44	1:33
3	20	1:61	1:167	1:81	1:49
Total	31	1:58	1:168	1:75	1:44

we see that the median effort ratio varies between 1:44 to 1:86 for the different versions. The upper fourth ratio varies between 1:33 to 1:64. The last row in table 8 presents the framework instantiation/development ratios for all 31 instantiations. The average framework instantiation/development ratio is 1:58 which means that half of the instantiations made required an effort less than 1,8 percent of the framework development effort. The upper fourth instantiation/development ratio is 1:44, which means that 75 percent of the instantiation made required an effort less than 2,3 percent of the total development effort.

These figures provide strong evidence supporting the claim that framework technology delivers reduced application development efforts. Especially, if we also consider the fact that we have included

activities not related directly to the pure framework instantiation process in the instantiation effort data.

Discussion

Essentially, the distribution of development effort method is a qualitative analysis of the effort distribution for the development with the aim to identify the reasons and causes why some figures are deviating from the estimations or have the certain values. One may argue that this analysis should be a part of any project conclusion report. The only difference is that it is more important to perform the analysis since the cost of developing and evolving a framework is higher than for a normal software system of comparable size. As mentioned earlier, it is of importance that development phases and activities are well-defined so comparisons with other framework project will be possible. Another issue is to assure that each framework instantiation is viewed as a separate project so that not framework instantiation effort will be charged on the development of the next framework version. Otherwise it will be hard, if not impossible, to obtain the framework instantiation/development ratio for the framework.

4. Assessment and Comparison

The three evaluation methods have been applied to the same framework, i.e. the BGW framework, which makes it possible to asses them and compare them with each other. The assessment will cover the management issues i.e. identification of evolution-prone modules, framework deployment, change impact analysis, benchmarking and requirements management as well as a discussion about the difficulty to apply the methods, their need of tool support and experienced advantages and disadvantages of the methods. In the following section we will assess the three methods and how they perform with respect to the issues above and in section 4.2 the three methods are compared to each other.

4.1 Assessment

Evolution identification using historical information

The evolution identification method performs well with respect to identifying evolution-prone modules. Through systematic application of the method on smaller and smaller parts of the application framework the method ends up with a list of evolution-prone modules (and subsystems). A minor disadvantage with the method is that it also identifies modules which have a natural high growth rate and thus are not a problematic modules that need to be restructured. An example of this are modules that are responsible for specific framework parametrization aspects which grows in a natural way with the number of possible parametrization values. Since the evolution identification method works with historical data it does not contribute to the framework deployment aspects which deal with deciding when to release the current framework version under development. The method assists management with some information when doing a change impact analysis, but since the method uses historical information it will, in most cases, fail in predicting changes for the future version. However, the method identifies modules that are likely candidates for restructuring in the framework. Regarding the benchmarking aspect the method is able to deliver information for classification and/or characterisation of typical evolution-prone module patterns for frameworks. This requires a relatively large number of observed frameworks, their subsystems and modules before typical categories can be established. The requirements management aspect is not at all addressed by the method.

The method is simple to apply since it only makes use of class declarations and the number of classes defined. A potential problem is that one must be aware of how to handle the addition of new modules during the framework evolution. Tool support is recommended, but a specialized tool for calculating the growth and change rates and storing the collected data is relatively simple to develop. However, it requires that the organisation has a coding standard that prescribes that each class must include a description of which subsystem and module it belongs to.

Stability assessment

The outcome of the stability assessment method gives an indication about how stable the framework version is. Especially the relative- and normalised-extent-of-change metrics provide information of how large the impact the requirements had on the framework structure. If historical trends for the current and other frameworks exist for the metrics, the metric values give a “rough” indication of maintenance effort needed. This with reservation for requirements that have architectural impact and thus affect the whole framework. Deviating metrics values from the limit values in the framework stability indicators are indications of evolution which is not desirable and eventually has to be handled explicitly, if time schedule and budgets allow. Thus, the stability assessment method is useful for change impact analysis.

Regarding identification of evolution-prone modules the method gives no support. But a possible approach could be to apply the stability assessment method on subsystems and modules and thereby gain knowledge about evolution-prone modules. Neither the framework deployment aspect is directly addressed by the method. But if the method is applied more frequently during the iterative development of the framework version, the extent-of-change metrics provide information with respect to whether the framework is becoming more stable or not during the iterations. The other metrics in the stability assessment method also indicate stability or not depending if the values are deviating from the ones in the framework stability indicators. The observed trends in the metric values give useful information for deciding whether to release the framework version or to iterate the development once again.

The requirements management aspect is not addressed by the stability assessment method. Eventually, the method will indicate a higher value than expected for any or both of the extent-of-change metrics as a result of incorporating a difficult instantiation requirement. The issue of benchmarking is supported by the method since it collects a number of metrics and thereby providing empirical information about the evolution of a framework. A historical set of data from previous assessments together with other information will make the prediction of the current framework’s evolution more accurate.

The method as such is fairly simple to apply but it requires that the set of metrics used are clearly defined and consistently interpreted. Both between framework versions and different frameworks in order

to provide comparable data. Tool support is highly recommended since a huge amount of data has to be dealt with. The tool used in our study [4] had problems with the large BGW framework. The problem was related to the number of file directories used in the framework and not the number of classes.

An advantage of the method is the possibility to change the underlying set of metrics. This is useful if the frameworks developed are adhering to a specific coding standard, a specific technical domain, e.g. distributed systems, or any other organisational issue. A minor disadvantage of the method is the calculation of the aggregated metrics, especially the class level metrics changes, if no tool support is provided. The limits used in the framework stability indicators may be discussed since they may be affected by organisational standards but our experiences from the application of the method on three different frameworks from three different organisations support that there is a degree of confidence in the limits.

Distribution of development effort

The distribution of development effort method does not contribute directly to the change impact analysis but if stability assessment is performed on the framework the effort data may be useful for future effort estimations on change impact analysis based on a stability assessment.

Obviously, the distribution of development effort method provides benchmarking information which can be useful in the development and evolution of new framework as well as for the instantiation of these. In a longer perspective, and if it is an organisational objective, the development and the instantiation effort data obtained can provide an empirical foundation for the development of cost models for framework development as well as investment analysis models.

Regarding the requirements management issue the method identifies those instantiations that have been costly to realize. If the organisation has adopted the strategy to incorporate the framework instantiation requirements into the next version of the framework, the identification of costly instantiations may point out difficult requirements which probably should not be incorporated in the next framework version. In that way, the method is useful with respect to requirements management issue.

A framework deployment decision should not be taken on the basis of effort consumed or other historical effort information but on achieved quality levels of the framework. Thus, the method does not assist management with the framework deployment decision. The issue of identifying evolution-prone modules is neither supported by the method. A possible approach could be if effort data is collected per module but achieving this granularity of effort data collection in practice seems difficult.

The method is not difficult to apply since software development organisation have procedures to collect effort data, often automated. The important thing is to have a clear and well-defined description of the activities in the framework project and that the staff correctly report their time with respect to activities defined. A minor advantage of the method is that it may help in staffing a framework project with different specialist depending on the relative distribution of the effort.

4.2 Comparison

In this section we present a comparison between the three methods with respect to perceived benefits and costs, table 9.

Table 9. Benefits and costs

Method	Benefit	Cost
Evolution Identification Using Historical Information	Simple metric used	Medium
Stability Assessment	Adaptable metric set Visualize inheritance structure changes Comprehensive	High
Distribution of Development Effort	Already implemented in most organization	Low

Applying the distribution of development effort method in an organisation is cheap since time reporting procedures are generally present in the organisation. The other two methods require purchase or development of data collection tools. In addition, depending on the maturity of the organisation the introduction of data collection activi-

ties can be relatively costly, but since only product metrics are collected this should be manageable. The benefit of using the Evolution Identification method is that it is relying on only one simple metric which is easy to collect. The stability assessment method is the most costly to apply but offers a number of benefits compared to the other two methods. The metric set used in the method provides a more comprehensive view of the framework compared to the evolution identification method since it address six different framework stability aspects. Another benefit is the possibility to change the metric set used in the method. It is also possible, as in our case, to select a metric set which provides information about inheritance structure changes or cover the metric used in the evolution identification method. To summarize, there exists differences in costs between the three methods but the advantages of using the most expensive method, stability assessment, are justified by the benefits it provides. In addition, the collected data is a superset of the data required by the evolution identification method. Thus, choosing the stability assessment method automatically facilitates the use of the evolution identification method as well.

5. Related Work

In this section we present related work to ours which addresses framework evolution in general as well as economical aspects of object-oriented frameworks, evolution identification and framework assessments.

The work by Roberts and Johnson [25] present a pattern language that describes typical steps for the evolution of an object-oriented framework. Common steps in the evolution are development of a white-box framework (extensive use of the inheritance mechanism), the transition to a black-box framework (extensive use of composition), development of a library of pluggable objects etc. These steps give a coarse-grained description of a framework's evolution. However, they do not explicitly address and describe where and how the framework evolves when it has reached a certain state.

Regarding economical aspects of object-oriented framework development and customization we are only aware of two references. Moser and Nierstrasz [23] discuss the productivity of the framework

user. Their study compares the function point approach with their own System Meter approach for estimating framework customization productivity. The idea behind the System Meter approach is to distinguish between the internal and external sizes of objects, and thus to measure only complexity that has a real effect on the size of the system to be implemented, i.e. incremental functionality. Their study reports that the effect of framework reuse does not increase the software engineer's productivity using the System Meter approach. Compared to our study, Moser and Nierstrasz do not discuss the costs for framework development and customization nor do the authors discuss the distribution of costs within and between framework versions but only the productivity aspect.

Rösel [26] is the only reference that presents some quantitative statements about framework development and reuse. Rösel's experience is that the cost of a framework may be 50 percent of the total cost for the first three applications. Once these three or more applications are implemented the framework will most likely have paid for itself. He also remarks that to reach this pay-off level significant commitment and investment is required. Compared to our study, Rösel gives no explicit quantitative support for his claim and the topic is just briefly discussed.

The issue of identification of evolution-prone modules is, of course, discussed in the work by Gall et al. [10] where historical information about modules and programs is used as a means to identify structural shortcomings of a large telecommunication switching system. Our work uses a similar approach but on a smaller and object-oriented system, 300-600 classes, and we use historical information about modules and classes, which are entities of smaller granularity than in the work Gall et al.

Recent work by Lindvall and Sandahl [16] shows that software developers are not so good at predicting from a requirements specification how many and which classes that will be changed. Their empirical study shows that only between 30-40 percent of the classes actually changed were predicted to be changed by the developers. This can be seen as an argument for using more objective approaches for identifying change-prone and evolution-prone parts of an object-oriented software system as the one used in our study.

Assessments of frameworks has been addressed by Bansiya [2] and Erni and Lewerentz [8]. Bansiya's assessment approach is simi-

lar to ours since we have used it as a basis. One difference is that we have added an additional metric which has been used for validating one of the original metrics. In addition, we have added a set of framework stability indicators which can be used for indicating framework stability. We have also applied our method on three frameworks, both commercial and proprietary ones.

Erni and Lewerentz [8] describe a metrics-based approach that supports incremental development of a framework. By measuring a set of metrics (which requires full source code access) an assessment of the design quality of the framework can be performed. If the metric data is within acceptable limits, the framework is considered to be good enough from a metrics perspective otherwise another design iteration has to done. The approach is intended to be used during the design iterations, before a framework is released and does not consider long term evolution of a framework as is the case with the stability assessment method.

6. Conclusion

Object-oriented framework technology has become common technology in object-oriented software development. An object-oriented application framework is a reusable asset that constitutes the basis for a number of applications in the same domain. As with all software, frameworks tend to evolve. Once the framework has been deployed, new versions of a framework cause high maintenance cost for the products built with the framework. This fact in combination with the high costs of developing and evolving an application framework make it important to have controlled and predictable evolution of the framework's functionality and costs.

We have discussed three methods

- Evolution Identification Using Historical Information
- Stability Assessment
- Distribution of Development Effort

which have been applied to between one to three different frameworks, both in the proprietary and commercial domain. The methods provide management with information which will make it possible to

make better and well-informed decisions about the framework's evolution, especially with respect to the following issues; identification of evolution-prone modules, framework deployment, change impact analysis, benchmarking and requirements management.

References

[1] G. Andert, Object Frameworks in the Taligent OS, *Proceedings of Compcon 94*, IEEE CS Press, Los Alamitos, California, 1994.

[2] J. Bansiya, Evaluating Application Framework Architecture Structural and Functional Stability, accepted for publication in *Object-Oriented Application Frameworks: Problems & Perspectives*, M. E. Fayad, D. C. Schmidt, R. E. Johnson (eds.), Wiley & Sons, http://indus.cs.uah.edu/research-papers.htm, February 1998

[3] J. Bansiya, Assessment of Application Framework Maturity Using Design Metrics, Accepted for publication in *ACM Computing Surveys -Symposia on Object-Oriented Application Frameworks*, February 1998, http://indus.cs.uah.edu/research-papers.htm

[4] J. Bansiya, *A Hierarchical Model for Quality Assessment of Object-Oriented Design*, Ph.D Dissertation, University of Alabama, 1997

[5] J. Bosch, P. Molin, M. Mattsson, P. Bengtsson. Problems & Experiences, In *Object-Oriented Application Frameworks*, M. E. Fayad, D. C. Schmidt, R. E. Johnson (eds.), Wiley & Sons, 1998.

[6] J. Bosch, Design of an Object-Oriented Framework for Measurement Systems, in *Object-Oriented Application Frameworks*, M. Fayad, D. Schmidt, R. Johnson (eds.), John Wiley, 1998.

[7] S. R. Chidamber, C. F. Kemerer, A Metrics Suite for Object-Oriented Design, *IEEE Transactions on Software Engineering*, Vol. 20, No, 6, pp. 476-493, June 1994.

[8] K. Erni, C. Lewerentz, Applying design metrics to object-oriented frameworks, *Proceedings of the Metrics Symposium*, 1996.

[9] Fayad, M. E.; Schmidt, D. C.; and Johnson, R. E., (Eds.) Object-Oriented Application Frameworks: Problems & Perspectives , John Wiley & Sons Inc., New York NY, 1999

[10] H. Gall, M. Jazayeri, R. G. Klösch, G. Trausmuth, Software Evolution Observations Based on Product Release History, Proceedings of the Conference on Software Maintenance, 1997, pp. 160-166.

[11] E. Gamma, *Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools*, (in German), Springer-Verlag, 1992

[12] D. Geary, *Graphic Java 1.1: Mastering the AWT*, Prentice Hall, 1997.

[13] H. Huni, R. Johnson, R. Engel, A Framework for Network Protocol Software, *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages and Applications*, Austin, USA, 1995.

[14] R.E. Johnson, V.F. Russo, Reusing Object-Oriented Design, *Technical Report UIUCDCS 91-1696*, University of Illinois, 1991.

[15] W. Li, S. Henry, Object-Oriented Metrics that Predict Maintainability, *Journal of Systems and Software*, Vol. 23, pp. 111-122, 1993.

[16] Lindvall, M. and Sandahl, K. (1998). How Well do Experienced Software Developers Predict Software Change? *Journal of Systems and Software*, 43(1):19 – 27.

[17] M. Mattsson, *Object-Oriented Frameworks - A survey of methodological issues*, Licentiate Thesis, LU-CS-TR: 96-167, Department of Computer Science, Lund University, Sweden, 1996.

[18] M.Mattsson, Effort Distribution in a Six Year Industrial Application Framework Project, *Proceedings of the International Conference on Software Maintenance - 1999*, pp. 326-333, 1999

[19] M. Mattsson, J. Bosch, Observations on the Evolution of an Industrial OO Framework, *Proceedings of the International Conference on Software Maintenance - 1999*, pp. 139-145, 1999

[20] M. Mattsson, J. Bosch, Characterizing Stability in Evolving Frameworks, I*n Proceedings of the 29th International Conference on Technology of Object-Oriented Languages and Systems*, TOOLS EUROPE '99, Nancy, France, pp. 118-130, June 7-10, 1999

[21] M. Mattsson, J. Bosch, Stability Assessment of Evolving Industrial Object-Oriented Frameworks, Accepted with reviosons for the *Journal of Software Maintenance*, 1999

[22] Peter Molin and Lennart Ohlsson, Points & Deviations -A pattern language for fire alarm systems, *Proceedings of the 3rd International Conference on Pattern Languages for Programming*, Monticello, USA, 1996.

[23] S. Moser, O. Nierstrasz, The Effect of Object-Oriented Frameworks on Developer Productivity, *IEEE Computer,* pp. 45-51, September 1996.

[24] T. Neward, *Core Owl 5.0 : Owl Internals for Advanced Programmers*, Manning Publications Company; 1997

[25] D. Roberts, R. Johnson, Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks, *Proceedings of the Third Conference on Pattern Languages and Programming*, 1996.

[26] A. Rösel, Experiences with the Evolution of an Application Family Architecture, *Proceedings of the Second International ESPRIT ARES Workshop, Las Palmas, Spain, pp. 39-48*

[27] G. Shepherd, S. Wingo, *MFC Internals: Inside the MFC Architecture*, Addison-Wesley, 1994.

[28] S. Sparks, K. Benner, C. Faris, Managing Object-Oriented Framework Reuse, *IEEE Computer,* pp. 53-61, September 1996.