Research Report 9/97

Object-Oriented Frameworks - Problems & Experiences

by

**Jan Bosch, Peter Molin,
Michael Mattsson, PerOlof Bengtsson**

Department of
Computer Science and Business Administration
University of Karlskrona/Ronneby
S-372 25 Ronneby
Sweden

ISSN 1103-1581
ISRN HKR-RES—97/9—SE

Object-Oriented Frameworks -
Problems & Experiences
by Jan Bosch, Peter Molin, Michael Mattsson, PerOlof Bengtsson

ISSN 1103-1581
ISRN HKR-RES—97/9—SE

Printed by Psilander Grafiska, Karlskrona 1997

Object-Oriented Frameworks - Problems & Experiences

Jan Bosch   Peter Molin   Michael Mattsson   PerOlof Bengtsson
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
E-mail: ARCS@ide.hk-r.se[1]
WWW: http://www.ide.hk-r.se/~ARCS

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 their usage in real projects. The authors have been involved in the design, maintenance and usage of several object-oriented frameworks and based on the experiences from these projects, a number of problems related to frameworks are described. The problems are organised according to four categories, i.e. framework development, usage, composition and maintenance. For each category, the most relevant problems and experiences are presented. This paper may help software engineers to avoid the described problems, whereas researchers may try to address these topics in their 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, among others, 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 lead to the development of *object-oriented frameworks*, i.e. 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.

After 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, operating systems within computer science 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 universities and even at the governmental level. For instance, the EU-sponsored Esprit project REBOOT [Karlsson 95] had a considerable impact on the object-oriented thinking and development in the organisations involved in the project and later caused the development of a number of object-oriented frameworks, e.g. [Dagermo & Knutsson 96].

1. Personal e-mail addresses: [Jan.Bosch | Peter.Molin | Michael.Mattsson | PO.Bengtsson]@ide.hk-r.se

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 e.g. [Moser & Nierstrasz 96]. 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 paper 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 paper is an overview and discussion of the obstacles identified during these framework projects. The obstacles have been organised into four categories. The first category, framework development, describes issues related to the initial framework design, i.e. 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, among others, 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.

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

The remainder of this paper is organised as follows. In the next section, the history of object-oriented frameworks is described and a consistent terminology is defined. Subsequently, in section 3, some examples of object-oriented frameworks are discussed in which one or more of the authors were involved; either as a designer or as a user. Section 4 discusses the obstacles in object-oriented framework development and usage that were identified, organised in the four aforementioned categories. The paper is concluded in section 5.

2 Object-Oriented Frameworks

2.1 History of Frameworks

Early examples of the framework concept can be found in literature that has its origins in the Smalltalk environment, e.g. [Goldberg & Robson 89] and Apple Inc. [Schmucker 86]. 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 implementing of Macintosh applications. Frameworks attained more interest when the Interviews [Linton et al. 89] and ET++ [Weinand et al. 89] user interface frameworks were developed and became available. Frameworks are not limited to user interface framework but have been defined for many other domains as well, such as operating systems [Russo 90] and fire-alarm systems [Molin 96b, Molin & Ohlsson 96]. 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 hundred object-oriented frameworks [Andert 94, Cotter & Potel 95]. 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, testing of frameworks, etc. Regarding documentation, patterns have been used for documentation of frameworks [Johnson 92, Huni et al. 95] and for describing the rationale behind design decisions for a framework [Beck & Johnson 94]. Other interesting work of general framework nature is the work about restructuring (refactoring) of frameworks. Since frameworks often undergo several iterations before even the first version is released, the framework design and code changes frequently. In [Opdyke 92], a set of behaviour-preserving transformations, refactorings, are defined that help to remove multiple copies of similar code without changing the behaviour. Refactoring can be used for restructuring inheritance hierarchies and component hierarchies [Johnson & Opdyke 93].

[Roberts & Johnson 96] describe the evolution of a framework as starting from a white-box framework, a framework which is reused mostly by subclassing, and developing into a black-box framework, a framework which mostly is reused through parametrization. The evolution is presented as a pattern language describing the process from the initial design of a framework as a white-box framework, to a black-box framework. The resulting black-box 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 behaviour 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 exist. The probably most referenced definition of a framework is found in [Johnson & Foote 88]:

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 paper, the following notions are used: *core framework design*, *framework internal increment*, *application specific increment*, *object-oriented framework* and *application*.

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* (e.g. 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* [Pree 94]. The core framework design describes the typical software architecture for applications in the domain.

However, the core framework design has to be accompanied with 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 realisations 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 comprise 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 [Roberts & Johnson 96] where the distinction between the framework and a component library is made, our interpretation of a framework includes class libraries.

Some authors categorise frameworks into *white-box* and *black-box* frameworks, e.g. [Johnson & Foote 88], or *calling* and *called* frameworks, e.g. [Sparks et al. 96]. In a white-box (inheritance-based) framework, the framework user is supposed to customize the framework behaviour through subclassing of framework classes. As identified by [Roberts & Johnson 96], 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 parametrize the behaviour. A black-box (parametrized) framework is based on composition. The behaviour of the framework is customized by using different combinations of classes. A parametrized framework requires deep understanding of the stable and flexible aspects of the domain. Due to its predefined flexibility, a black-box framework is often more rigid in the domain it supports. A calling framework is an active entity, pro-actively 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, a framework hardly never is a pure white-box or black-box framework or a pure calling or called framework. In general, a framework has parts that can be parametrized and parts that need to 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. One can recognise the following phases in framework-centred 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*, sometimes also referred to as the *framework instantiation* phase or *application development phase*. The main result of this phase is an application developed reusing one or more frameworks. Here, the framework user has to 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 fulfil the actual applications requirements. These new classes we refer to as the *application-specific increment*. The problem of composing frameworks is further discussed in section 4.3, whereas a model for framework usage is outlined in section 4.2.
- The *framework evolution & maintenance phase*. The framework, 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, etc. These kinds of issues will be discussed in section 4.4.

Activity	**The software engineer's role**	**Software artifacts involved**
Framework Development	Framework Developer	The Core Framework Design Framework Internal Increments
Framework Usage[a]	Framework User[b]	The Core Framework Design Framework Internal Increments Application-Specific Increment
Framework Maintenance & Evolution	Framework Maintainer	The Core Framework Design Framework Increments

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

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 Example Frameworks and Application Domains

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 multi-building 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, we refer to [Molin 96a, Molin & Ohlsson 96].

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 consist 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. We refer to [Bosch 96b] 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 outbound communication. The gateway provides services for processing the billing information, e.g. 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 multi-threading is used heavily. We refer to [Lundberg 96] 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 sub frameworks, i.e. the core resource allocation framework and two supporting frameworks, handling user interface and persistence through relational databases, respectively. The main part of the generic behaviour consists of the general scheme of *allocations*, i.e. one or several *resources* could be allocated by one *allocator* represented by an *allocation*. The framework user specifies the concrete resources for the application, implements some specific behaviour such as editing, storing and displaying. The framework handles the general business rules, e.g. 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, maintenance, etc. The framework consists of seven concept hierarchies organised into three categories, i.e. 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, etc. 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 [Betlem et al. 95] for more details concerning the framework.

4 Problems and Experiences

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, where an application only is concerned 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** [Schäfer et al. 94] aims at describing the domain that is to be covered by the framework. To capture the requirements and identification of concepts one may refer to previously developed applications in the domain, domain experts and 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 underlying the framework. Based on this the top level design of the framework is made. Examples of architectural styles or patterns can be found in [Shaw & Garlan 96] and [Buschmann et al. 96].
- During **framework design** 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 [Deutsch 89]), 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. [Johnson & Russo 91] conclude that the only way to find out if something is reusable is to reuse it. For frameworks, this boils down to developing applications that use the framework.
- To evaluate the useability 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, one 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 recognised. Without a clear, complete and correct documentation that describes how to use the framework, a user manual, and design document that describe how the framework works, the framework will be nearly impossible to use by software engineers not involved in the framework design.

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

4.1.1 Domain scope

When deciding to develop a framework for a particular domain, there is a problem of determining the right size of the domain. On one hand, if the domain is too large, the development team has not 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 reasonable time.

On the other hand, a smaller domain uses the known experiences in a much more efficient way. A 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 above, one can deduce that defining the scope of the domain is a decision that has to be carefully balanced. One problem is that, due to the fact that 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 one considers the framework including yet another aspect more useful than if it would lack that (see e.g. [Bosch 96b]).

4.1.2 Framework documentation

The purpose of framework documentation is two-fold. Firstly, there is a need to communicate the information about the framework design and other related information during the framework development. Secondly, there is a need to transfer information on how to use the framework to the framework user.

In the first case, it is convenient to rely on informal communications to spread the knowledge about the developed framework. The often small design group uses different ad-hoc techniques to interchange information among the individuals in the team. The problem is that these techniques are not very efficient and does 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 different kinds of manuals, using different media. The way the documentation is done is usually ad-hoc, making it hard to understand and compare. In [Johnson 92], it is argued 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, white-box or black-box, there is a need for the user to understand the underlying principles or basic architecture of the framework. Otherwise, detailed rules and constraints defined by the framework developers makes 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 there is a large need to elicitate and document them. The problem is how to convey this information in a concise form to the framework user.

A number of methods have been proposed for documenting frameworks. A first example is the cookbook approach, e.g. [Krasner & Pope 88] and [Apple 89] which are example based, and the meta-patterns approach proposed by [Pree 94]. The pattern approaches provide a second example, where patterns are used to describe the architecture of the framework, e.g. [Beck & Johnson 94, Huni et al. 95], or the use of the framework, e.g. [Johnson 92, Lajoie & Keller 94]. Finally, a framework description language (FDL) [Wilson & Wilson 93] can be used that, more formally, describes, e.g. 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.

4.1.3 Business models

Even though it may be well feasible to develop a framework for a particular domain from a technological perspective, 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 build 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 organisations 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 [Kihl & Ströberg 95], but this have 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 [Lim 96]. Since no reliable investment models exist, decisions on framework development are often based on ‘gut-feeling’ with the down side of many frameworks designs never take place since the technical staff is unable to convince management that framework development would be economically liable.

4.1.4 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, e.g. behaviour and attributes are very likely to change in several cases. The analysis of such behaviour and attributes needs to be emphasized in the development model, since it is the nature of frameworks to provide reusable design and implementation that covers the variations, i.e. *hot spots*, in the domain.

During development of the framework an architecture must be selected or developed. The criteria for a framework architecture is 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 taken on whether some parts of the framework should be designed as a sub-framework or if everything should be designed as a single monolithic framework. Also, inter-operability requirements need to be established, e.g. should the framework co-operate with other frameworks, such as user-interface frameworks or persistence frameworks.

In addition to the existing object oriented design methods, [Rumbaugh et al. 91, Booch 94, Jacobson et al. 92], which most often only support inheritance, aggregation and associations, more emphasis is needed for abstract classes, dynamic binding, type parametrization, hot-spots and pre- and post-conditions. Abstract classes and dynamic binding represent the way to define and implement abstract behaviour which is not emphasized in existing design models. Type parametrization needs support in the methods since this is another useful way of supplying pre-defined abstract behaviour which 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 goes and support in the design methods for expressing and designing these are very important. Pre- and post-conditions provide the framework developer with the possibility of specifying the restrictions for usage of parts of the framework.

4.1.5 Verifying abstract behaviour

When developing framework there is a need to verify the result. The current state of practice is to develop test applications using the framework and 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 can not 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.

Secondly, 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, e.g. 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 behaviour. Standard testing procedure does not allow for testing of abstract implementations, such as abstract base classes.

4.1.6 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 [Poulin 94] 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 a two-fold problem. First, the domain is not stable in itself, but evolves constantly (see also section 4.4.3). Secondly, the issue of domain scope and boundaries discussed in section 4.1.1 also complicates the decision on framework release. Deciding whether the framework is sufficiently well documented is hard, since there exists no generally accepted documentation method 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 maintenance and usage of the framework and the instantiated applications. This issue is discussed in more detail in section 4.4.

4.2 Framework Usage

Although we have seen that it often is feasible to produce complex application based on a framework with rather modest development effort, we also have 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 [Mattsson 96] and consist of the following activities:

- **Requirements analysis** aims at collecting and analysing 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 non-functional 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 section 4.3.
- When it is decided what framework(s) to reuse, one can deduce which parts of the application need to be defined for the application itself, i.e. one specifies the required **application-specific increments** (ASIs).
- After the specification of the ASIs, these 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.

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

4.2.1 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 as spiral models and evolutionary models. We believe that the current state of the art in framework-based application development is still in the ad-hoc state 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 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 of such specifications would be to specify the new classes that must be implemented and the classes that requires adaptation by subclassing.

4.2.2 Applicability of the framework

One initial difficulty is to understand the intended domain of the framework and its applicability to 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 general interpret as the domain. This dimension defines the functional boundaries of the domain. Another dimension is the underlying hardware architecture. This dimension addresses such aspects as multi-processor systems vs. single processor systems or distribution aspects. A third dimension is the interoperatability, where aspects such as coexistence with other applications are discussed. Issues in this dimension are, for example, if a particular database engine can or must be used or if there are any restrictions on the graphical user interface. A fourth dimension incorporates non-functional aspects such as performance, fault-tolerance, and capacity. Examples could be a framework intended for batch processing and not for real-time data processing, or a framework that have upper limits of 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 build 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 first of all determine whether it is possible to use the framework at all, and secondly to determine the amount of work needed to implement the required application.

4.2.3 Estimations of the increment

The problem of using frameworks compared to traditional application development is, first of all, that most of the time spent in such a project could well be on understanding the framework and very little on actual coding. From a productivity point of view this may seem like framework-based application is slow compared to a traditional approach. On the other hand, complex applications can be build very fast so real productivity is high. The consequences of traditional estimations techniques based on number of produced lines of code is inadequate in the framework case. [Moser & Nierstrasz 96] discusses 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%/10% rule implying that 10% of the development time is spent on 90% of the application, and 90% is spent on the remaining difficult 10% of the application. The difference depends on whether a specific feature is supported by the tool (framework) or not. 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 even be 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 done. For accurate estimations 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.

4.2.4 Understanding the framework

When a white-box 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 pre-requisite 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 multi-threading 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 could be dynamic memory allocation strategies where it is important that the increment follows 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 understands 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, e.g. the framework documentation, but we refer to section 4.1.2 for a discussion of these issues.

An alternative avoiding the understanding approach, or a complement to it, is to explicitly specify the constraints that the framework put on the application. This approach has, among others, been proposed by [Molin 96a, Mattsson 96].

4.2.5 Verification of the application specific increment

An application based on a complex framework or based on several frameworks could be very difficult to test, therefore it could be advisable to verify the application specific increment beforehand. For large scale applications, it is advisable that the increment is locally certifiable, i.e. tested or verified locally without executing the entire application. Such a verification must be based on a specification of what requirements the increment needs to fulfil. There are two aspects of verification. The first is a functional verification that verifies that the increment implements the expected behaviour 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, among others, been investigated by [Molin 96a] 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.

4.2.6 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 that 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 black-box 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 which 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 post-conditions or behavioural interaction constraints [Meyer 92, Helm et al. 90, Lajoie & Keller 94]. 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 only represents 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 where the framework is in full control.

In this section, the problems related to the composition of reusable components and frameworks with an object-oriented framework are discussed. 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.

4.3.1 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 [Garlan et al. 95] this is referred to as *architectural mismatch*, i.e. the architectural styles based on which the frameworks are designed are different, complicating composition so much that it may be impossible to compose. [Shaw & Garlan 96, Buschmann et al. 96] 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 either require substantial amounts of glue code or redesign of parts of one or both of the frameworks. Lately, one can recognise an increasing interest in the domain of software architecture. Explicitly specifying the architectural style underlying a framework design seems to be the first step towards a solution to this problem.

As an example we use the resource allocation and the gateway billing framework. The resource allocation framework is based on a layered architecture, i.e. the typical 3-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, among others, due to the mismatch in the underlying architectural styles.

4.3.2 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. The situation that may occur is that both frameworks contain a representation (e.g. a class) of the same real-world entity, but modelled from their respective perspectives. This results in the situation that the representations by the frameworks may have modelled both overlapping and exclusive properties. In the application, however, the real-world entity should be modelled by a single object and the two representations should be integrated into one.

The situation is thus that two frameworks exist F_1 and F_2 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, i.e. $P_{C^r{}_{F1}} \subset P_r$ 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, i.e. the application class $C^r{}_A$ inherits from both classes $C^r{}_{F1}$ and $C^r{}_{F2}$ thereby combining the properties from these classes, i.e. $P_{C^r{}_A} = P_{C^r{}_{F1}} \cup P_{C^r{}_{F2}}$.

In practice, however, the composition of properties from the frameworks classes is not as straightforward as presented above, due to the fact that 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, e.g. 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 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 behaviour 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 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.

4.3.3 Composition of entity functionality

A typical example of this problem can be found in the 3-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 and such. Constructing an application class by composing instances from the three frameworks, or by multiply inheriting from classes in the three frameworks will not result in the desired result. Among others, 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 behaviour from the user-interface and database framework classes needs to be superimposed on the object [Bosch 96a]. 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 behaviour for notifying the user-interface and database classes, e.g. using the *Observer* design pattern [Gamma et al. 94]. One could argue that the application domain class should have been extended with such behaviour during design, but, as we 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 non-volatile memory, i.e. an EEPROM. To deal with this, each entity was implemented by two objects, i.e. one application object and one persistence object. These two objects were obviously tightly coupled and had frequent interactions, due to the fact that they both represented parts of one entity.

4.3.4 Hollywood principle

In [Sparks et al. 96], the distinction is made between ‘calling’ and ‘called’ frameworks. Calling frameworks are the active entities in an application, calling 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 behaviour. If we would compose both frameworks in an application, the two control loops would conflict leading to incorrect behaviour. 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 lead 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 synchronisation code. Since a class often cannot be modularly extended with synchronisation code, it requires that all reused classes are 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 that two concurrent threads.

4.3.5 Integrating legacy components

A framework presents, among others, 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 only contains behaviour for internal framework functionality and not much of the behaviour 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 a framework class. It is, however, far from trivial to integrate the 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, one runs into typing conflicts.

This requires the software engineer to create some form of adaptation or bridging. For instance, a class could be defined, inheriting both from 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 [Lundberg & Mattsson 96] and as a solution they propose the use of templates. Their solution, however, requires that the framework is designed using their approach, which is unlikely in the general case.

4.4 Framework Evolution & Maintenance

Development of a framework has to be seen as a long term investment and, as such, it has to be seen as a product that needs to be maintained. In the beginning of a framework development effort often several design iterations are necessary. The reasons for the iteration are, according to [Johnson & Russo 91], that the framework is supposed to be reusable and the only way to prove this is to reuse the framework and identify the short-comings. In [Opdyke 92], a set of behaviour-preserving transformations, refactorings, has been identified that characterizes the code and low-level design changes that may occur in the iterations. Especially, changes/refactorings related to inheritance hierarchies [Opdyke & Johnson 93] and component hierarchies [Johnson & Opdyke 93] are 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 organisation has to decide to either 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 organisation supporting the framework may require the domain must be adapted accordingly. Today it is unclear 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.

4.4.1 Framework change

Imagine the situation where a fault is found in an application using a framework. Where should the error be corrected? It could either be 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 due to the complexity of the framework (i.e difficult to understand how the framework works) or lack of access to the framework's source code. Often, the organisation 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 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 that we introduce new faults in these applications. In addition, the cost of upgrading all existing applications is high.

The first occasion where 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 easiest solved by direct support to the application developers by the framework team [Sparks et al. 96]. The framework developers then collect all the experiences from the first application and iterate the framework design once more to get a more mature 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 this is either due to the fact the domain covered by the framework is incomplete, i.e we have failed to capture the domain in our domain analysis, or due to the fact that the application does not match the intended framework domain, i.e. we have decided to develop an application whose requirements have a bad fit of the domain covered by the framework. In both cases, the framework will generally be changed, forcing the organisation to face the problem of selecting maintenance strategies. This will be further described in the following section.

4.4.2 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 has a bad fit to the framework's 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 through more extensive reuse of the framework. A third consequence is that the organisation is forced to maintain two versions of the framework, the original since there may exist applications based on this version and the redesigned for future applications to be developed. The duration of the period during which two maintenance lines need to be supported depends on the expected life-time on 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 only is 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 life-time. However, this may an acceptable situation if it is expected that no similar applications will be developed in the foreseeable future.

Concluding, the problem of deciding the maintenance strategy is dependent on time-pressure, estimated life-times for the software involved, existing and expected future applications etc. and no clear guidelines exist that support the decision.

4.4.3 Business domain change

The framework is developed in a domain that is closely related to the organisation's business domain. Unfortunately, the business domain is often weakly defined and not stable over time. Especially when the organisation's business domain changes frequently the framework will be more difficult to reuse, and, if not maintained, 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 has to be considered in the investment of the framework development effort.

There are, in principle, three approaches attacking the problem of business doming change. One may define the original framework domain much wider than currently useful, some kind of super-domain, that will capture most future new domain changes. As discussed in section 4.1.1, there are obvious problems of defining such a super-domain. For example, it often is unclear in what direction the super-domain should be expanded to incorporate an existing adjacent business domain (e.g. the burglar alarm domain within a framework for fire alarm systems) or to incorporate a future business domain. Other problems are to obtain funding for a larger framework effort, finding domain expertise for the meta- domain and verification of the super-domain.

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 organisation 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 of 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 one has (hopefully) learned many lessons from the first framework development.

4.4.4 Design iterations

It is a well-known fact that framework development is an iterative process. Traditional software development also requires iterations but iterations are more important and explicit when designing frameworks. The underlying reason, we believe, is that frameworks primarily deal with abstractions, and that 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, to a high degree related to the first problem, is that it is difficult to stop iterating and decide that the framework is ready for release (see also section 4.1.6). These problems are analogous to the problems of testing, which is not surprising since, in principle, a design iteration is a modification activity resulting from testing. However, some important differences exist, since iteration activities in frameworks are much more expensive than the 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.

Concluding, we have identified three issues that are open for improvements. Firstly, quantitative and qualitative guidelines of when to stop framework design. Secondly, methods that can predict the number of iterations and the corresponding effort involved. Finally, it should be investigated whether it is possible to evaluate a framework for a specific application without completely writing a test application.

5 Conclusion

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 paper, we have experienced the advantages of framework-based development when compared to the traditional approach of starting from scratch. In addition to our own positive experiences, also others, e.g. [Moser & Nierstrasz 96], have identified that frameworks reduce the amount of development effort required for application development and can be prosperous investments.

However, as we report in this paper, there still exist a number of problems and hindrances that complicate the development and usage of object-oriented frameworks. 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, 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, estimations of development time and size of application specific code, understanding of the framework, verification of 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 & 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 paper we believe to be relevant both for software engineers and researchers on object-oriented reuse. After reading this paper, practitioners will hopefully be able to avoid problems that some of us experienced. In addition, we believe that this paper could be used as a research agenda for researchers on object-oriented software development or on software reuse in general.

Acknowledgements

The authors wish to thank Christer Lundberg and Lennart Ohlsson for their comments on some topics discussed in this paper.

References

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

[Apple 89]. Apple Computer Inc., *MacAppII Programmer's Guide,* 1989.

[Beck & Johnson 94]. K. Beck, R. Johnson, ‘Patterns Generate Architectures,’ *Proceedings of the 8th European Conference on Object-Oriented Programming*, Bologna, Italy, 1994.

[Betlem *et al.* 95]. B.H.L. Betlem, R.M. van Aggele, J. Bosch, J.E. Rijnsdorp, An Object-Oriented Framework for Process Operation, Technical report, Department of Chemical Technology, University of Twente, 1995.

[Booch 94]. G. Booch, *Object Oriented Analysis and Design with Applications - 2 ed.*, Benjamin/Cummings, 1994.

[Bosch 96a]. J. Bosch, ‘Composition through Superimposition,’ to be published in the proceedings of ECOOP’96 workshops, LNCS series, Springer-Verlag, 1996.

[Bosch 96b]. J. Bosch, ‘Design of an Object-Oriented Framework for Measurement Systems,’ *submitted*, November 1996.

[Buschmann *et al.* 96]. F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M.Stahl, *Pattern-Oriented Software Architecture - A System of Patterns, John Wiley* & Sons, 1996.

[Cotter & Potel 95]. S. Cotter, M. Potel, *Inside Taligent Technology,* Addison-Wesley, 1995.

[Dagermo & Knutsson 96]. P. Dagermo, J. Knuttson, ‘Development of an Object-Oriented Framwork for Vessel Control Systems,’ *Technical Report ESPRIT III/ESSI/DOVER Project #10496,* 1996.

[Deutsch 89]. L.P Deutsch, ‘Design Reuse and Frameworks in the Smalltalk-80 system,’ in *Software Reusability,* T. J. Biggerstaff and A. J. Perlis, editors, Vol II, ACM Press, 1989.

[Gamma *et al.* 94]. E. Gamma, R. Helm, R. Johnson, J.O. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1994.

[Goldberg & Robson 89]. A. Goldberg, D. Robson, *Smalltalk-80: The Language,* Addison-Wesley, 1989.

[Garlan *et al.* 95]. D. Garlan, R. Allen, 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,* April 1995.

[Helm *et al.* 90]. R. Helm, I.M. Holland, D. Gangopadhyay, ‘Contracts:Specifying Behavioural Compositions in Object-Oriented Systems’, *Proceedings of ECOOP/OOPSLA’90,* Ottawa, Canada, 1990.

[Huni *et al.* 95]. 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 Conference*, Austin, USA, 1995.

[Jacobson *et al.* 92]. I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, *Object-oriented software engineering. A use case approach*, Addison-Wesley, 1992.

[Johnson & Foote 88]. R.E. Johnson, B. Foote, ‘Designing Reusable Classes,’ *Journal of Object-Oriented Programming,* Vol 1, No. 2, June 1988.

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

[Johnson 92]. R.E. Johnson, ‘Documenting Frameworks with Patterns,’ *Proceedings of the 7th Conference on Object-Oriented Programming Systems, Languages and Applications,* Vancouver, Canada, 1992.

[Johnson & Opdyke 93]. R.E. Johnson & W.F. Opdyke, ‘Refactoring and Aggregation,’ *Proceedings of ISOTAS ’93: International Symposium on Object Technologies for Advanced Software,* 1993

[Karlsson 95]. E-A. Karlsson, Editor, *Software Reuse - a Holistic Approach*, John Wiley & Sons, 1995.

[Kihl & Ströberg 95]. M. Kihl, P. Ströberg, ‘The Business Value of Software Development with Object-Oriented Frameworks,’ *Master Thesis,* Department of Computer Science and Business Administration, University of Karlskrona/Ronneby, Sweden, May 1995 (in Swedish).

[Krasner & Pope 88]. G. E. Krasner, S. T. Pope, ‘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.

[Lajoie & Keller 94]. R. Lajoie, 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.

[Lim 96]. Wayne Lim, 'Reuse Economics: A comparison of Seventeen Models and Directions for Future Research', *Proceedings of the International Conference on Software Reuse,* April 1996.

[Linton *et al.* 89]. M.A. Linton, J.M. Vlissides, P.R. Calder, 'Composing User Interfaces with Interviews,' *IEEE Computer,* Vol 22., No. 2, February 1989.

[Lundberg 96]. L. Lundberg, '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.

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

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

[Meyer 92]. B. Meyer , 'Applying Design by Contract,' *IEEE Computer,* October 1992.

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

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

[Molin & Ohlsson 96]. 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.

[Moser & Nierstrasz 96]. S. Moser, O. Nierstrasz, 'The Effect of Object-Oriented Frameworks on Developer Productivity,' *IEEE Computer,* pp. 45-51, September 1996.

[Opdyke 92]. W.F. Opdyke, 'Refactoring Object-Oriented Frameworks,' *PhD thesis,* University of Illinois at Urbana-Champaign, 1992

[Opdyke & Johnson 93]. W.F Opdyke, R.E Johnson, 'Creating Abstract SuperClasses by Refactoring,' *Proceedings of CSC'93: The ACM 1993 Computer Science Conference,* February 1993.

[Poulin 94]. Jeff Poulin, 'Measuring Software Reusability', *Proceedings International Conference on Software Reuse,* November 1994.

[Pree 94]. W. Pree, '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.

[Rumbaugh *et al.* 91]. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-oriented modeling and design*, Prentice Hall, 1991.

[Roberts & Johnson 96]. 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.

[Russo 90]. V. F. Russo, 'An Object-Oriented Operating System,' *PhD thesis,* University of Illinois at Urbana-Champaign, October 1990.

[Shaw & Garlan 96]. M. Shaw, D. Garlan, *Software Architecture - Perspectives on an Emerging Discipline,* Prentice Hall, 1996.

[Schmucker 86]. K.J. Schmucker, *Object-Oriented Programming for the Macintosh*, Hayden Book Company, 1986.

[Schäfer *et al.* 94]. W. Schäfer, R. Prieto-Diaz, M. Matsumoto, *Software Reusability,* Ellis-Horwood Ltd., 1994.

[Sparks *et al.* 96]. S. Sparks, K. Benner, C. Faris, 'Managing Object-Oriented Framework Reuse,' *IEEE Computer,* pp. 53-61, September 1996.

[Weinand *et al.* 89]. A. Weinand, E. Gamma, R. Marty, 'Design and Implementation of ET++, a Seamless Object-Oriented Application Framework,' *Structured Programming,* Vol. 10, No. 2, July 1989.

[Wilson & Wilson 93]. D. A. Wilson, 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, 1993.